Thirteenth Real-Time Linux Workshop

Organizing committee

Local staff at Czech Technical University Prague
Michal Sojka (Faculty of Electrical Engineering, Department of Control Engineering)
Pavel Piša (Faculty of Electrical Engineering, Department of Control Engineering)
Petr Hodač (SUT - Středisko un*xových technologií, UN*X Technologies Center)

Real-Time Linux Foundation Working Group and OSADL
Prof. Nicholas Mc Guire, (Lanzhou University, China)
Andreas Platschek (OpenTech, Austria)
Dr. Carsten Emde (OSADL, Germany)

Program committee

Roberto Bucher, RTAI Maintainer, Switzerland
Alfons Crespo, University Valencia, Spain
Carsten Emde, OSADL, Germany
Oliver Fendt, Siemens, Corporate Technology, Germany
Gerhard Fohler, Technische Universität Kaiserslautern, Germany
Thomas Gleixner, Linutronix, Germany
Nicholas Mc Guire, Lanzhou University, China
Hermann Härting, TU Dresden, Germany
Zdeněk Hanzálek, Czech Technical University, Prague
Paul E. McKenney, IBM
Jan Kissza, Siemens, Germany
Miguel Masmano, Universidad Politecnica de Valencia, Spain
Odhiameo Okeeh, University of Nairobi, Kenya
Pavel Piša, Czech Technical University, Prague
Andreas Platschek, OpenTech, Austria
Zhou Qingguo, Lanzhou University, China
Ismael Ripoll, University Valencia, Spain
Georg Schiesser, OpenTech, Austria
Stefan Schönegger, Bernecker + Rainer, Austria
Michal Sojka, Czech Technical University, Prague
Martin Terbuc, University of Maribor, Slovenia
Mathias Weber, Roche Diagnostics Ltd., Switzerland
Bernhard Zagar, Johannes Kepler University, Austria
Peter Zijlstra, Red Hat, Netherlands

Prague 2011
Preface


The event is still driven by the simple goal: bring together developers and users, present new developments, discuss ‘real’ user demand and get to know those anonymous people that only exist as e-mail folders on your mailing-list archive, and last but not least, encourage the spirit of a community.

Free Libre Open-Source Software is a fast growing technology pool and we can observe this well in the breath of development presented at this year’s Real-Time Linux Workshop. Not only has FLOSS reached traditional automation and control, but it is increasingly reaching into technical areas that were almost unthinkable for “non-commercial” entities - safety critical systems. This development is underpinned by developments in the FLOSS tools for formal and semi-formal verification. With other words, FLOSS is covering the entire area from educational material, traditional automation and control, robotics to aerospace and automotive industries - while no single workshop can ever claim to cover it all - we do hope to have collected a representative snapshot of this sprawling community.

Thank you very much for attending the Real Time Linux Workshop. We hope that your expectations are met during this workshop, as developer, as user or as newcomer to real time Linux.

The organizing committee
Acknowledgements

No Real Time Linux community, no Real Time Linux users, no Real Time Linux Workshop. Therefore, our thanks go to the Real Time Linux community for the work done in Open Source software development as an international cooperation.

All authors and attendees, thanks a lot for your contribution in any respect.

In particular, we want to express our thanks to the sponsors of the 13th Real-Time Linux Workshop:

Last but not least, thanks to everybody having contributed to this workshop and not explicitly mentioned above.

OSADL – Real-Time Linux Foundation Working Group
Contents

Real-Time Linux in Education .................................................. 1

Alberto Guiggiani, Michele Basso, Massimo Vassalli and Francesco Difato ........ 1

A Nonlinear Model-Based Control realized with an Open Framework for Educational Purposes
Klaus Weichinger ................................................................. 7

Real-Time Linux Applications .................................................. 17

The Witch Navigator A Software GNSS Receiver Built on Real-Time Linux
Petr Kacmarik, Pavel Kvar, Ondrej Jakubov and Frantisek Vejrazka ................. 17

Case Study: Challenges and Benefits in Integrating Real Time patch in PowerPc Based Media System
Manikandan Ramachandran and Aviral Pandey .................................... 29

Hard real-time Control and Coordination of Robot Tasks using Lua
Markus Klotzbuecher and Herman Bruyninckx .................................. 37

Platform independent remote control and data exchange with real-time targets on the example of a rheometer system
Martin Leonhartsberger and Bernhard G. Zagar ................................ 45

Using GNU/Linux and other Free Software for Remote Data Collection, Analysis and Control of Silos in Mexico
Don W. Carr, Juan Villalvazo Naranjo, Rubén Ruelas, Benjamin Ojeda Magaña .... 53

A Status Report on REACT, a Control Engine that runs on top of GNU/Linux for Creating SCADA and DCS Systems
Don W. Carr, Rubén Ruelas, Benjamin Ojeda Magaña, Adriana Corona Nakamara . 57

Development of an optical profile measurement system under RTAI Linux using a CD pickup head
Gregor Gerstorfer and Bernhard G. Zagar ..................................... 61

Process Data Connection Channels in uLan Network for Home Automation and Other Distributed Applications
Pavel Piša, Petr Smolík, František Vacek, Martin Bohaček, Jan Štefan and Pavel Němeček .......................... 67

Real-Time Linux Infrastructure and Tools ..................................... 75

Application of RT-Preempt Linux and Sercos III for Real-time Simulation
Michael Abel, Luis Contreras and Prof. Peter Klemm ........................... 75

Lachesis: a testsuite for Linux based real-time systems
Andrea Claudi and Aldo Franco Dragoni ...................................... 81

Generic User-Level PCI Drivers
Hannes Weisbach, Björn Döbel and Adam Lackorzynski ......................... 91

COMEDI and UIO drivers for PCI Multifunction Data Acquisition and Generic I/O Cards and Their QEMU Virtual Hardware Equivalents
Pavel Piša and Rostislav Lisový ............................................... 101

A Framework for Component-Based Real-Time Control Applications
Stefan Richter, Michael Wahler and Atul Kumar ............................... 107

Alberto Guiggiani  
Università di Firenze, Dipartimento di Sistemi e Informatica  
Via di S. Marta 3, Florence, Italy  
guiggiani@gmail.com

Michele Basso  
Università di Firenze, Dipartimento di Sistemi e Informatica  
Via di S. Marta 3, Florence, Italy  
basso@dsi.unifi.it

Massimo Vassalli  
National Research Council of Italy, Institute of Biophysics  
Via De Marini 6, Genoa, Italy  
massimo.vassalli@cnr.it

Francesco Difato  
Italian Institute of Technology (IIT), Dept. of Neuroscience and Brain Technologies  
Via Morego 30, Genoa, Italy  
francesco.difato@iit.it

Abstract

In this paper we present the Realtime Suite, a project that aims to provide all the tools and guides needed to set up a Linux-RTAI real-time machine within a CACSD environment. This is addressed in particular to researchers and students approaching the world of real-time control applications for the first time, or people searching for an open-source alternative to commercial solutions.

1 Introduction

When building a real-time machine, open-source software can constitute a suitable alternative to commercial solutions in terms of functionality, performance and costs. On the other hand, the major drawbacks of this choice are the long amount of time and the advanced computer skills needed to make all the pieces work together. These steps represent an obstacle that might discourage less-experienced people, like students for example, to enter the world of Linux-based real-time applications. With Realtime Suite[1] our aim is to provide the tools and documentation needed to easily set up a working real-time machine for signal acquisition and processing within a Computer Aided Control System Design (CACSD) framework. In order to achieve this, we started by collecting all the pieces required to configure a complete software chain: an Ubuntu operating system with RTAI[2] modified kernel; Comedi[3] to interface with DAQ boards; Scicoslab[4] with RTAI-Lib[5][6] palette to generate and build the real-time target and, finally, QRTAI-Lab[7] and RTAI-XML[8] to monitor the target locally or remotely, respec-
tively. The next step was to edit source codes with the objective to avoid conflicts and compile errors. Lastly, we packed everything in the so-called Realtime Suite alongside with documentation with simple step-by-step instructions and examples. In addition to that, we configured the suite on a Virtual Machine (VM) that works out-of-the-box, useful for testing purposes.

In order to verify the efficacy of the proposed approach in developing an actual real-time application, a Realtime Suite based system has been installed[9] to realize a feedback control for nanometer-precision specimen tracking in an optical tweezer system with piezoelectric actuators.

The paper is organized as follows. In Section 2 an overview of the software included in the Realtime Suite package will be presented. Section 3 will cover the remaining two components of Realtime Suite: a step-by-step tutorial and the preconfigured virtual machine. Section 4 will focus on results of performance tests to evaluate achievable sampling rates and jitter, while in the last section an application will be proposed in which a Realtime Suite based machine has been used to control specimen position in an optical tweezers setup.

## 2 An Open-source Rapid Controller Prototyping Environment

In this section will follow a brief description of the software needed to build a Linux-based machine for signal acquisition and conditioning. The proposed configuration is based on open-source projects and packed in one single archive, available for download in the Realtime Suite website [1]. Some of the source files included in the package has been edited in order to assure compatibility between each software version and obtain a fluent installation procedure. A schematic diagram of functional connections between the various components of the real-time machine is shown in figure 1.

### 2.1 RTAI and Comedi

RTAI[2], Real Time Application Interface, is a Linux extension that allows the execution of tasks with strict temporal constraints, enabling hard real-time (HRT) control algorithm implementations. Development started back in 1999 with the work of Paolo Mantegazza at Politecnico di Milano. The Suite includes RTAI version 3.8.1 alongside with Linux Kernel 2.6.32 patched from LinuxCNC[10].

Comedi[3] is a set of drivers provided as Linux kernel extensions that enable communication with a broad range of commercial data acquisition boards. A collection of libraries is provided, with APIs which allow the real-time target to interface with the device.

### 2.2 Scicoslab and RTAI-Lib

Scicoslab[4] is a scientific software package born in 2005 from the BUILD4 distribution of Scilab, when the development of the latter left the INRIA (Institut National de Recherche on Informatique et on Automatique). It provides a wide range of functions for mathematical and engineering applications, and comes with the CACSD software Scicos. With Scicos [11] control algorithms can be designed by means of a graphical UI, connecting blocks from available palettes. The RTAI-Lib palette, installed in the machine setup phase, adds various block functional to hard real-time control targets. They can be divided in four groups: blocks providing interface with Comedi devices, mailboxes for communications between multiple targets running on the same ma-
chine, semaphores for task synchronization, and input/output blocks, like signal generators, scopes and meters. The RTAI-Lib palette blocks are shown in figure 2.

2.3 QRTAILab

QRTAILab [7] is a RTAI-Linux Graphical User Interface useful to manage real-time targets running on the same machine. It creates a virtual oscilloscope to monitor signals and allows online modification of the target parameters. It was developed starting from the source code of xrtailab, a software part of the RTAI-Lab package, using the Qt libraries. With respect to xrtailab, QRTAILab is much lighter[12] CPU-wise when connecting to complex targets.

2.4 Remote interface: RTAI-XML

One of the major concerns that becomes evident when designing real-time control system architectures is the intrinsic duality between hard real-time (HRT) and soft real-time (SRT) components. While the first requires the programmer to focus on timing constrains, latencies, and sampling rates, SRT components like human-machine interfaces (HMI) require flexibility, user-friendliness and efficient data handling. In order to separate those two worlds, a web services approach can be taken. Web services[13] allow two pieces of software to communicate through a network defining a standard object access protocol. Here only the communication language is shared, leaving freedom of implementation.

RTAI-XML[8][14] brings a web services approach to the world of Linux real-time control applications. The Realtime Suite includes the RTAI-XML server component that are compiled on the RT machine through the last steps of the tutorial. This server components acts as an intermediary between the real-time targets and a remote procedure call framework. Using XML, it bridges the target signals and parameters over the network, making them accessible from remote clients like jRTAILab[15], a Java implementation of xrtailab, or any other application-oriented client as the one presented in Section 5.

3 Setup Tutorial and Realtime Suite VM

The Realtime Suite project was born with the objective to guide less experienced people into the world of Linux real-time applications. To achieve this, the software package described in the previous section comes with a step-by-step tutorial that accompanies the user in all the phases needed to build a working real-time control machine, from the initial system setup to compilation and execution of the first real-time target. The majority of the steps involves running console commands to compile and install the various components, but everything is explained and viable for users without advanced programming skills. The tutorial is a revised version of the one proposed by Bucher, Mannori, Netter[16].

In addition to the software package and its related tutorial, a virtual machine has been released. It is based on Linux Ubuntu 10.04 and includes all the software of the Realtime Suite compiled and ready-to-use. It is available through the RTAI-XML project website[8] in the Open Virtual Machine format (.ova) and can be executed on a wide range of host machines with the open-source virtualization software VirtualBox[17]. Due to the limitations of a virtualized system it cannot be used in substitution of a physical RTAI machine in actual signal acquisition and conditioning, but can constitute and handy tool in the design phases of Scicos control algorithms or while testing remote RTAI-XML clients.

4 Performance Tests

This section shows results of two kinds of experimental tests performed on a machine configured with Realtime Suite. The machine was a commercial personal computer with the following specifications:

- CPU Intel Core2Duo 6300 @ 1.86 GHz
- System RAM 1 GB
- Video Card ATI Radeon HD 4350
- HDD WD Caviar Blue SATA @ 7200 RPM
- National Instruments DAQ board NI-PCI 6229

4.1 Jitter

The first test aims at evaluating precision of the sampling tick, in respect to sampling rate and task priority. In order to calculate it, two separate real-time targets are in execution: the first with high priority and high sampling rate (2.5 KHz), the second with low priority and low sampling rate (125 Hz). Each one gets sampling effective timestamps and compares them with expected timestamps, calculated by adding task period to a counter. The absolute value of their difference is the instantaneous jitter. The two tasks are kept running until the maximum jitter reaches a stable value. Results are shown
in table 1.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Samp. Rate [Hz]</th>
<th>Max Jitter [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>2500</td>
<td>19425</td>
</tr>
<tr>
<td>Low</td>
<td>125</td>
<td>23445</td>
</tr>
</tbody>
</table>

**TABLE 1: Jitter Test**

### 4.2 Sampling Rates

Sampling rate test is executed to establish maximum reachable sampling rates in respect to number of input and output channels open on the data acquisition board. This is done by running a single target with a fixed number of input/output Comedi blocks and gradually increasing sampling rates until output waveform, measured with an oscilloscope, stops being equivalent to the one generated inside the block diagram. The test is repeated for different numbers of I/O channels as summarized in figure 3.

**FIGURE 3: Sampling Rate Tests**

Looking at the chart, it is clear how the major impact on maximum sampling rates is given by the number of input channels. Anyway, sampling rates in the magnitude of some KHz, suitable for a wide range of control applications, are easily handled even with many open channels. This is a remarkable result for a machine built with such limited and cheap hardware components.

### 5 Application - Optical Tweezers Control

Optical tweezers are scientific instruments that exploit the property of light to exert forces on matter to optically trap a small particle (usually a silica or glass bead with diameter in the range of micrometers) near the focus of a concentrated laser beam, without mechanical contact[18]. For small displacements of the trapped bead, few hundreds of nanometers, from the laser focus centre, the particle suffers a force actuated by the light beam, in the pN order, attracting it towards the center of the trap. This light-force has the important property to show a behavior similar to a spring for displacement of the bead of about 200 nm, therefore the force-displacement relationship can be described by the Hookes Law[19]. In the past two decades optical tweezers found applications for nanometers and micrometers scale experiments in many scientific areas, spacing from physics to chemistry and biology. The application shown in this paper involves studies on neuron cells during differentiation, and relies heavily on a real-time control machine built with the Realtime Suite.

**FIGURE 4: Optical Tweezers scheme**

Setup is shown in figure 4, and has been described in details in [9]. A laser beam is used (a) to trap a microsphere. The interference fringes generated by the interaction of the bead with the laser are projected to a four-quadrant photodiode(b). This interferometric measurement allows measuring the position of the sphere in respect to the center of the beam with sub-nanometer and sub-millisecond resolutions. Photodiode signals are acquired by the RTAI machine equipped with a National Instruments DAQ PCI board and Comedi drivers, and sampled at 2 KHz rate. On the machine a control target designed with Scicoslab implements a proportional/integral feedback(c) comparing bead displacement with an external reference position given by the user. Controller output pilots a Physik Instrumente 3-axis piezoelectric positioning system (d). Thanks to RTAI-XML, all signals and feedback parameters are sent through the network to another machine where a custom C# interface is running (e). This interface provides a graphical front-end for parameter editing, signals visualization and measurements saving. Moreover, automatically handles recoveries with a DC-motor translation stage to extend piezo-
electric stage range.

![FIGURE 5: Section of the C# interface](image)

In figure 5 is shown the section of C# interface that provides a graphical front-end to edit on-the-fly parameters, e.g. proportional and integral gains, of the control algorithm running on the real-time machine.

This application shows the functionalities of a control machine built with Realtime Suite. We were able to acquire multiple analog channels (nine in this application) sampled with a bandwidth of two KHz, and to condition by a custom control algorithms, with the use of limited hardware (a commercial PC and a DAQ board). In addition to that, we could separate implementation of the HMI soft real-time components from business logic ones thanks to RTAI-XML.

### 6 Conclusions

With Realtime Suite, we have configured a ready-to-use software package useful for researchers/users approaching real-time applications for the first time. Everything is based on open-source projects supported by active developers and communities. At the cost of a personal computer with a supported data acquisition board, and a few hours of work, it is possible to build a real-time machine capable of running custom control targets, sampling signals with a bandwidth of a few kHz. Flexibility in control architecture design is added by the inclusion of the RTAI-XML project, which allows to separate the HRT components (signal acquisition, control algorithms...) from the SRT components (user interface, data manipulation), to develop appropriate strategies for interfacing two distinct worlds.

### References


A Nonlinear Model-Based Control realized with an Open Framework for Educational Purposes

Klaus Weichinger
BIOE Open Hardware Automation System Developer
3300 Greinsfurth, Austria
snaky.1@gmx.at - http://bioe.sourceforge.net

Abstract

Today, nonlinear model-based control methods are an essential part in different control applications. To provide a complete open framework for educational purposes this contribution extends common open source software (Scilab/Scicos, Maxima and rt-preempt Linux real-time system) with a low-cost do-it-yourself open hardware interface and a web-based monitoring system embedded into Scicos blocks.

The simple concept of the open hardware interface called Bioe (Basic Input Output Elements) allows the real-time application to interact with general analog and digital signals as well as to more complex devices (e.g. resistive touch panels, RC servos, I²C acceleration sensors). Furthermore, a prototype of a web-based monitoring system using Ajax is treated. It consists of a HTTP web server embedded into a Scicos block so that existing Scicos code generation packages for rt-preempt Linux can be used without modifications.

To demonstrate the applicability and usability of the proposed framework a nonlinear model-based control law for a mechatronic multi-input multi-output system is derived with the concept of input/output linearization and realized with the proposed open framework.

1 Introduction

Rapid Control Prototyping (RCP) is still associated with cost-intensive hardware and software investments. Such proprietary RCP frameworks are used in industrial applications because they are well known from education.

Mercifully, the real-time capability of the Linux kernel due to patches like Rtai [1] or rt-preempt [2] has highly increased and the usability of cost-saving open source software (OSS) for computer-aided control system design (CACSD) has improved within the last years. A fully usable RCP framework for industrial and educational applications is provided by Rtai-Lab which consists of a Linux distribution with a Rtai patched kernel, Scilab/Scicos [3] or ScicosLab [4] and the Comedi drivers collection [5] for a variety of data acquisition boards. Furthermore, the software xRtaiLab allows to scope and record signals and to edit system parameters. There also exist other real-time Linux projects like RTLlinux [6] and Xenomai [7], but the comparison of real-time approaches is not the topic of this paper.

This contribution picks up two aspects that should complete the existing OSS [2, 8, 4, 9] to a cost-and time-saving open RCP framework for rt-preempt Linux real-time systems. The first aspect is discussed in section 2 and concerns the hardware to interface a RCP system with the target system. The presented solution is a low-cost and do-it-yourself system called Basic Input Output Elements (Bioe, [10]) with a simple but very flexible interface. The second aspect treats a web-based monitoring approach that implements a HTTP web server within a Scicos block and uses an Ajax web-application to scope signals and edit parameters. The web-based concept, first results of the prototype and further details are topic of section 3.

In section 4 the proposed open RCP framework is used to implement a nonlinear model-based control law for a mechatronic system. The multiple-
input multiple-output (MIMO) mechatronic system consists of the well known mass-spring system with viscous friction that is actuated with a double-acting hydraulic piston (DAP). Beside the position control of the mass the sum-pressure of the DAP has to be stabilized at a constant value. The method of input/output exact linearization \[11, 12\] is used to derive the control law and a hardware-in-loop (HIL) simulation is used to test the control law and to demonstrate the usage of the BIOE system and the web-based monitoring system.

Usability and reliability of an open RCP framework are basic requirements so that open hard- and software can be used for educational purposes, a very important precondition to introduce open RCP frameworks into industrial applications.

2 Open Hardware - BIOE

The BIOE system was developed within a diploma thesis \[13\] and is a simple but effective piece of hardware to interface a PC with signals of a physical system. It consists of small modules that are connected parallel with the BIOE bus cable to the parallel port interface (Line Print Terminal, LPT) of the PC as shown in figure 1.

![FIGURE 1: A BIOE mock-up](image)

Each module can be addressed with a 4bit dip-switch and so up to 16 BIOE modules can be connected to the bus cable. An adapter board is used for signal amplification and to protect the LPT against damages. This interface is used for communication because of its real-time performance and simplicity. In addition, the LPT interface is still available on some embedded and desktop systems or can be retrofitted with e.g. PCI Express cards.

A BIOE module is equipped with an AT-Mega16 microprocessor. This 8bit microprocessor contains the functionality to deal with different types of signals and to provide the information via the BIOE bus by the use of 16 end-points (EP0, EP1, ..., EP15). Each end-point consists of a 16bit receive and 16bit transmit register \(RX_{adr,ep}\) and \(TX_{adr,ep}\) with the BIOE address \(adr \in \{0, \ldots, 15\}\) and the end-point number \(ep \in \{0, \ldots, 15\}\). These end-points are the common interface between the real-time application and the physical signal. The end-points are accessed with transactions. During a transaction a 16bit value is written from the PC to the register \(RX_{adr,ep}\) and is read from the BIOE module register \(TX_{adr,ep}\) back to the PC simultaneously.

![FIGURE 2: UML sequence diagram from the BIOE module firmware](image)

In addition, the end-points EP0 and EP1 are used to activate the required interface type with the device type number (DTN). This approach reduces the software effort for the real-time task and so only two Scicos blocks are required to use the whole functionality of BIOE (see section 2.2). Furthermore, each BIOE module contains all interface types that
are realized for BioE and each interface type can be activated with an unique DTN. After the initialization of the interface the corresponding interface function is processed and the informations are converted and exchanged between the end-points and the hardware interface as illustrated in figure 2.

The table 1 gives an overview of the provided interfaces:

<table>
<thead>
<tr>
<th>DTN</th>
<th>Interface Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>16 digital outputs</td>
</tr>
<tr>
<td>002</td>
<td>16 digital inputs</td>
</tr>
<tr>
<td>003</td>
<td>8 digital inputs, 8 digital outputs</td>
</tr>
<tr>
<td>050</td>
<td>two 10bit PWM, four 10bit ADC</td>
</tr>
<tr>
<td>090</td>
<td>square signal generator</td>
</tr>
<tr>
<td>091</td>
<td>square signal frequency measurement</td>
</tr>
<tr>
<td>100</td>
<td>incremental decoder</td>
</tr>
<tr>
<td>101</td>
<td>incremental decoder with HCTL2022</td>
</tr>
<tr>
<td>105</td>
<td>16x2 LCD driver, four push buttons</td>
</tr>
<tr>
<td>106</td>
<td>ADC, PWM, RC-Servomotor and incremental decoder</td>
</tr>
<tr>
<td>107</td>
<td>UART interface</td>
</tr>
<tr>
<td>120</td>
<td>RC5 infrared receiver</td>
</tr>
<tr>
<td>127</td>
<td>4 wire resistive touch interface</td>
</tr>
<tr>
<td>128</td>
<td>WII-Nunchuck interface (I²C)</td>
</tr>
<tr>
<td>129</td>
<td>WII-Remote IR camera interface (I²C)</td>
</tr>
<tr>
<td>230</td>
<td>electrical network simulator</td>
</tr>
<tr>
<td>231</td>
<td>torsional oscillator simulator</td>
</tr>
<tr>
<td>250</td>
<td>cycle time measurement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIN</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$V_{cc}$</td>
<td>+5 V supply</td>
</tr>
<tr>
<td>2</td>
<td>GND</td>
<td>ground supply</td>
</tr>
<tr>
<td>3</td>
<td>CS</td>
<td>Chip-Select (by Master)</td>
</tr>
<tr>
<td>4</td>
<td>CLK</td>
<td>Clock (by Master)</td>
</tr>
<tr>
<td>5</td>
<td>DO₀</td>
<td>Bit 0 Master $\rightarrow$ Slave</td>
</tr>
<tr>
<td>6</td>
<td>DO₁</td>
<td>Bit 1 Master $\rightarrow$ Slave</td>
</tr>
<tr>
<td>7</td>
<td>DO₂</td>
<td>Bit 2 Master $\rightarrow$ Slave</td>
</tr>
<tr>
<td>8</td>
<td>DO₃</td>
<td>Bit 3 Master $\rightarrow$ Slave</td>
</tr>
<tr>
<td>9</td>
<td>DI₀</td>
<td>Bit 0 Slave $\rightarrow$ Master</td>
</tr>
<tr>
<td>10</td>
<td>DI₁</td>
<td>Bit 1 Slave $\rightarrow$ Master</td>
</tr>
<tr>
<td>11</td>
<td>DI₂</td>
<td>Bit 2 Slave $\rightarrow$ Master</td>
</tr>
<tr>
<td>12</td>
<td>DI₃</td>
<td>Bit 3 Slave $\rightarrow$ Master</td>
</tr>
<tr>
<td>13</td>
<td>NC</td>
<td>not in use</td>
</tr>
<tr>
<td>14</td>
<td>NC</td>
<td>not in use</td>
</tr>
</tbody>
</table>

TABLE 1: List of available interface types

If a new interface type is required the implementation is done in the BioE module software. The concept to keep hardware related functionality on the BioE modules avoids any modification of the BioE software on the PC and reduces the efforts to keep the PC software updated. Hence, if the BioE communication via the end-points is ensured all available DTN interfaces can be used. In the following sections further topics concerning the BioE system are discussed.

2.1 BIOE Bus and Performance

The BioE bus is a 14 pole flat cable (see table 2) that connects the modules with the LPT adapter board. This bus is a compromise of low-cost cables and connectors, communication speed and the microprocessor performance. For the communication with transactions a 4 bit parallel bus is used and each transaction contains the device address, the end-point number, the register values and a very simple check-sum to detect transmission errors.

Performance measurements done with different PC’s return a transaction time $T_{\text{trans}}$ between 25 $\mu$s and 50 $\mu$s (time where CS is high; see figure 3). This parameter depends on the PC’s hardware, especially the type of parallel port (parallel port directly on the motherboard, a PCIe extension module, ...).

The following example demonstrates the determination of the smallest possible sample time of a control realized with BioE.

**Example 1** A PID control of a system with one input $n_i = 1$ and one output $n_o = 1$ should be realized with BioE. This fictive example illustrates how to determine the smallest sample time for the control task. For discrete time systems it is assumed that the time delay $T_{io}$ between reading the inputs and setting the outputs is zero. Nevertheless, due to finite calculation speed there is a delay. If a sample time $T_s$ is used so that the relation $T_{io} \leq 0.1 T_s$ is ensured...
then the effects of the timing error can be neglected. In this example a measurement of the BioE bus resulted in $T_{\text{trans}} = 30 \mu s$.

$$T_s = 10 T_{\text{trans}} (n_i + n_o) \quad (1)$$

The equation (1) delivers the minimum sample time of $T_s = 0.6 \text{ ms}$ for this PID control example.

### 2.2 Toolchain

The BioE toolchain consists of a set of Scicos blocks, a command-line tool and a tool with a graphical user interface. A CLisp API is also available. This toolchain uses the BioE C library which provides the required communication functions and contains the platform abstraction layer.

For the development and testing of BioE modules and new interface functions the command-line tool bioedude and the graphical user interface bioegui (figure 4) can be used to configure the modules and to access the end-points directly. These tools are also very useful to test sensors and actuators of a new mock-up. The toolchain can be compiled and used on WIN32 and Linux based platforms.

**FIGURE 4:** Screenshot of bioegui (uses FLTK, a cross-platform C++ GUI toolkit)

Only two Scicos blocks (figure 5) are required to include the BioE system into a Scicos model. The first block `bioe_dev` does not have inputs and outputs and it initializes the interface type of a BioE module. The second block `bioe_ep` provides the end-point access. This block can be configured so that it reads or writes an end-point triggered by an external event.

**FIGURE 5:** Scicos blocks (left side: device block; right side: end-point block)

To configure the parallel port the BioE API requires the base-address of the parallel port (e.g. 888 dec), the channel number (e.g. 1) to define the chip-select number, the speed-parameter (e.g. 0) to trim the communication speed and the pause-parameter (e.g. 9) to avoid CPU stress of the BioE module in consequence of a permanent BioE bus communication. Optionally, a fifth parameter is used to define a setting number that loads the port settings from the file `bioesettings.conf` if available. This feature will be required if a compiled BioE application has to be adapted to a different hardware platform (e.g. another parallel port base-address).

### 2.3 An Example Interface

**FIGURE 6:** Illustration of a complex BioE interface

All interfaces are described similar to DTN106 illustrated in figure 6. The concept of BioE allows such simple documentation so that the principle of *What You See Is What You Get* (WYSIWYG) fits very well.

### 3 Web-Based Monitoring

This section presents the prototype of a web-based monitoring system for rt-preempt Linux real-time applications created with Scicos. The idea was
to implement a HTTP server within the real-time application, to start the server as non-real-time thread and to exchange signals and parameters between the real-time application and the HTTP server (see section 3.1). A REST-style architecture [14] is used for AJAX based communication in which the signals are exchanged as XML formatted byte stream. A web application or an other kind of application uses this interface to establish a connection with the real-time application to scope signals and to modify parameters. For the client side communication and a basic web-based monitoring system see section 3.2.

3.1 Server Side Component

The RTXMLSERVER illustrated in figure 7 was realized within one Scicos block. During the initialization of this Scicos block a new POSIX thread is created (see listing 1).

![Diagram of the RTXMLSERVER](image)

**FIGURE 7: Simplified Structure Diagram of the RTXMLSERVER**

This thread processes the data-exchange between a remote client monitoring system and the real-time task. For the communication between the real-time thread and the server thread a object called RTSignalManager was implemented. The Scicos signal blocks for the RTXMLSERVER use this manager to register their signals and parameters and during the operation the signal value with time-stamp is passed via thread-safe FIFOs to the RTXMLSERVER. At the moment the RTXMLSERVER provides an access via HTTP requests (a reduced HTTP server is implemented according RFC-2616 [15]; TCP and UDP servers are planed).

```c
static pthread_t thrd;
static void *server_task(void *p)
{
    struct sched_param param;
    param.sched_priority = 10;
    if(sched_setscheduler(0, SCHED_FIFO, &param)==-1)
    {
        exit(-1);
    }
    // ... process the server
}

void rtxmlserver_fnc(scicos_block *block , int flag)
{
    if (flag==1) { /* set output */ }
    if (flag==2) { /* get input */ }
    if (flag==5) { /* termination */ }
    if (flag==4)
        { /* initialisation */
            // ... block operations
            pthread_create(&thrd, NULL, server_task, NULL);
        }
}
```

**LISTING 1: Schematic structure of the RTXMLSERVER Scicos block**

The HTTP server supports GET requests to access the file system and to load a web-page. In addition, a PUT-Request with the URI /rtxml.pipe passes the RTXML-Request within the HTTP request message-body to the RTXML-Request-Handler. The handler parses the XML byte stream and does the corresponding actions similar to SAX [16]. The XML formatted RTXML-Response is sent back to the client within the message-body of the PUT-Response. The whole description of the RTXML-Requests and RTXML-Responses is not intended within this contribution but sub-section 3.3 should give an idea about the structure of the XML formatted messages.

The concept to start a non-real-time thread within a Scicos block (see listing 1) can be applied for other purposes like image processing. In this case, the function to get the image from a camera and to do the image processing can be done within a non-real-time task. The results are passed to the real-time thread. This approach reduces the effort to start the application and to share the data with inter-process communication.

3.2 Client Side Application

The prototype of a client side monitoring system shown in figure 8 is an AJAX based web-application using JavaScript and jQuery (see [17]). AJAX (with jQuery) and the HTTP request method PUT are
used to establish and configure a connection with the Rtxmlserver and to exchange the signals and parameters. This functionality is encapsulated within the JavaScript Rtxmlclient class that prepares and stores the data in arrays. These arrays can be used for a visualization purposes, e.g., realized with a JavaScript plotting library to illustrate the signals continuously without refreshing the whole web-page. The library jQuery UI [17] was used to improve the look and feel of the visualization and to get a behavior comparable with a native application.

WEB-BROWSER  
HTTP-REQUEST  
RTXMLCLIENT  
JavaScript-Class using Ajax  
Web-Browser  
jQuery Web-Application  
HTML-Web Page

FIGURE 8: Simplified Structure Diagram of the web-based application using Rtxmlclient

Figure 9 shows the screenshot of the standard interface. It allows the configuration of the connection. If the connection to the real-time application is established, the three signal scopes and the parameter list are automatically configured. This standard interface consists of a HTML file, some CSS and JavaScript files. A custom interface can be created with a text editor and without the need of further compilers or a complex toolchain. This approach allows to prepare a demonstration system mock-up with a specialized interface for presentation purposes or to create laboratory setups where the students can configure the control-law parameters and test the performance of the control-law.

The first prototype works very well and can be used for applications, but if a high sample-rate and more signals are transmitted it is necessary to reduce the number of transmitted data-points. Otherwise, there is not enough CPU power to parse the XML formatted context with JavaScript. The reduction of data-points has no significant effect on the quality of the visualization. To provide data records with all sampled informations for offline analysis the Rtxmlserver can record all data-points into a local *.csv file and this file can be downloaded with the web browser.

FIGURE 9: Screenshot of a web-based monitoring example

3.3 About the RTXML-Request and RTXML-Response

The RTXML-Request in listing 2 contains most of the available functions. With the first XML tag <C>...</C> the client provides the connection number so that the Rtxmlserver knows that it is the same client. Then some action commands are sent to the Rtxmlserver and at the end the parameter with the id 6 is set with the value 2.44.

LISTING 2: An example of a RTXML-Request

The RTXML-Responses have a similar format (see listing 3). The Rtxmlserver collects all signals of one sample time, creates a list that contains the time (double precision), signal id and signal value (double precision), converts this list to a HEX-ASCII string and places it with XML tags into the RTXML-Response.

The main idea was to keep all information in double precision so that there is no information lost due
to truncation. The HEX-ASCII string yields to a 100% increase of the bytes to transmit. In the future the use of Base64 (RFC4648, [18]) coding is planned because this increases the bytes to sent only about 33%. Furthermore, it should be checked if the XML parsing can be improved.

LISTING 3: An example of a RTXML-Response

4 Application Example

To demonstrate the proposed open RCP framework this chapter is devoted to the position and sum-pressure control of a non-linear hydraulic system by the use of exact input-output linearization [11, 12]. The hydraulic system, realized with a PC and Bioe for HIL simulations, is controlled by another PC. Both PCs use Linux with a rt-preempt patched kernel to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML.

The hydraulic system, realized with a PC and Bioe for HIL simulations, is controlled by another PC. Both PCs use Linux with a rt-preempt patched kernel to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML.

The hydraulic system, realized with a PC and Bioe for HIL simulations, is controlled by another PC. Both PCs use Linux with a rt-preempt patched kernel to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML to provide real-time capability. The signals like position, pressure and volume flow are exchanged between the PCs via an analog interface and XML.

With the concept of exact input-output linearization it can be shown that the inputs of the system appear at \( \dot{e}_1 \) and \( \dot{e}_2 \) and in consequence the system is input-state linearizable. To keep these outputs as close as possible to zero the dynamics

\[
0 = \ddot{e}_1 + \alpha_{12} \dot{e}_1 + \alpha_{11} \dot{\dot{e}}_1 + \alpha_{10} e_1 \\
0 = \ddot{e}_2 + \alpha_{20} \dot{e}_2
\]

are chosen with a set of parameters \( \alpha_{10}, \alpha_{11}, \alpha_{12} \) and \( \alpha_{20} \) so that the dynamics (3) and (4) are asymptotically stable and have the required behavior. In the next step the control law

\[
Q_A = Q_A(x, v, p_A, p_B, x_d, \ddot{x}_d, \dot{x}_d, \ddot{e}_d, p_d, \dot{p}_d) \\
Q_B = Q_B(x, v, p_A, p_B, x_d, \ddot{x}_d, \dot{x}_d, \ddot{e}_d, p_d, \dot{p}_d)
\]

(5)

can be evaluated by solving the equations (3) and (4) for the volume flows \( Q_A \) and \( Q_B \). The time derivatives \( \ddot{x}_d, \dot{x}_d, \ddot{e}_d, p_d \) can be provided by a trajectory planning system and this allows a feed-forward tracking control of the system (2). Let us assume that the desired trajectories \( x_d(t) \) and \( p_d(t) \) are smooth and changes very slow and so the choice \( \ddot{x}_d = \dot{x}_d = \ddot{e}_d = \dot{p}_d = 0 \) is justifiable. Consider that the control law (5) depends on the piston velocity \( v \) which can not be measured. Hence, a velocity and disturbance observer is designed in the next step.

4.2 Observer Design

To implement the control law (5) a velocity observer can be used. The realized HIL mock-up (see
A Nonlinear Model-Based Control realized with an Open Framework for Educational Purposes

The following observer design (see, e.g., [12]) treats the general system

\[ \dot{x} = v \]
\[ \dot{v} = \frac{1}{m} (-c_x - d v + F + F_d) \]
\[ \dot{F}_d = 0 \]

with the position \( x \), the velocity \( v \) and the disturbance force \( F_d \). To get the equations for the DAP system \( F = A p_A - \alpha A p_B \) has to be used. The position \( x \) and the force \( F \) can be determined by measurements and calculations and so a reduced observer is designed to estimate the velocity \( v \) and the disturbance force \( F_d \). In the first step the following state transformation

\[ w_1 = v + k_v x \quad w_2 = F_d + k_{F_d} x \]

is introduced and results in the linear and time invariant system

\[ \dot{w} = A_{obs} w + u \]

with the new state vector \( w = [w_1 \ w_2]^T \), the matrix

\[ A_{obs} = \begin{bmatrix} k_v - \frac{d}{m} & \frac{1}{m} \\ \frac{d}{m} & -k_{F_d} \end{bmatrix} \]

and the input

\[ u = \begin{bmatrix} (d k_v - k_{F_d} - c - k_v^2 m) \frac{x}{m} + \frac{F}{m} \\ -k_v k_{F_d} x \end{bmatrix} \]

With a specific choice of parameters \( k_v \) and \( k_{F_d} \) the matrix \( A_{obs} \) becomes a Hurwitz matrix and then the standard approach of a trivial observer for the system (6) can be applied to get an estimation \( \hat{w} \) of \( w \), where \( \hat{w} \) indicates the estimated value of \( w \).

\[ \dot{\hat{w}} = A_{obs} \hat{w} + u \]

Now it is easy to prove that the estimation error for the velocity \( v \) and the disturbance force \( F_d \)

\[ e_v = v - \hat{v} = w_1 - \hat{w}_1 \]
\[ e_{F_d} = F_d - \hat{F}_d = w_2 - \hat{w}_2 \]

has the asymptotically stable dynamic

\[ \dot{e} = A_{obs} e \]

with \( e = [e_v \ e_{F_d}]^T \). Nevertheless, the proof of the stability of the observer does not allow for conclusions concerning the stability of the overall system because of the nonlinear system (2).

### 4.3 Implementation

To realize the HIL simulation Ubuntu 10.04 LTS [22] with the rt-preempt patched 2.6.31-11-rt kernel from the Lucid repository was installed. ScicosLab 4.4.1 [4] with the rt-preempt code generation package [9] is used for the simulation and real-time code generation. The Bioe and RTXMLSERVER blocks were installed into the SCICOSLAB directory structure. A performance measurement of a dummy simulation with a cycle time of \( T_s = 2 \) ms and a Bioe with DTN250 (see table 1) was performed on an Intel(R) Pentium(R) M 1.4 GHz notebook. During the test duration of two hours the maximum \( T_s,max \) = 2.08 ms and the minimum \( T_s,min \) = 1.928 ms of the cycle time were detected. It should be noted that the measured time variations cover the rt-preempt patched Linux notebook, the Bioe bus and the Bioe element software.

In addition, the rt-preempt code generation package [9] was modified so that a real-time scaling parameter can be used to slow-down or speed-up the HIL simulation. This feature can be very useful in the case of complex systems (e.g. distributed parameter systems) were the calculation of the dynamics can not be performed in real-time. In the case of the DAP system control a real-time scaling factor of three is used to keep the time of the Bioe transactions and the settling time of the analog interface circuit small compared to the cycle time. In addition, RC low-pass filters with \( R = 1 \) k\( \Omega \) and \( C = 1 \) \( \mu \)F were used to convert the digital PWM signals \( f_{PWM} = 15.6 \) kHz into analog signals.

For the numerical simulation and HIL simulation Scicos blocks of the systems (2) and (5) are required. For this job a MAXIMA toolbox was developed that allows to export a system

\[ \dot{x} = f(x, u, p) \]  \hspace{1cm} (7)
\[ y = g(x, u, p) \]  \hspace{1cm} (8)

into a ready to use Scicos block. A Scicos block normally consists of an interface-function and a computational-function (see, e.g. [23]). The interface-function is written in the Scilab language and provides an interface to edit the system parameters \( p \) and the initial values of the state \( x \). The computational-function can be written in Scilab or C and contains the set of ordinary equations.
$f(x, u, p)$ with the system input $u$ and the equations $g(x, u, p)$ for the system output $y$.

Such a code-generation package is an important component of a complete RCP framework because it avoids the repeated implementation of equations, the search for typing errors and it is easy to keep the analytic calculations and the simulation code synchronized. The created blocks for the DAP system and the control law can be used for numerical simulations and the real-time code generation with Scicos.

To obtain realistic results the numerical simulation in figure 11 takes effects like quantization, signal ranges and noise into account. The numerical results fit very well with the HIL measurements and they are discussed in the following section.

### 4.4 Results

Following system parameters were used for the numerical and HIL simulation: $m = 1$ kg, $d = 1 \frac{N s}{m}$, $c = 1 \frac{N s}{m}$, $A = 1 \cdot 10^{-4}$ m$^2$, $\alpha = 0.7$, $V_{A0} = 5 \cdot 10^{-4}$ m$^3$, $V_{B0} = 3.5 \cdot 10^{-6}$ m$^3$, $E_f = 1.6 \cdot 10^5 \frac{N}{m}$. Some of the parameter values are not realistic because they were chosen so that bad influences caused by noise, quantization, offsets and signal limitations due to the low-cost analog interface will be reduced. The eigenvalues of the observer dynamic matrix $A_{obs}$ are placed with the coefficients $k_v = -3$ and $k_{F_d} = -4$ to $-2$. Finally, the poles of the error dynamic (3) are defined with $\alpha_{12} = 90$, $\alpha_{11} = 2700$, $\alpha_{10} = 2700$ at $-30$ and the pole of the error dynamic (4) is aligned with $\alpha_{20} = 10$ to $-10$.

The results are summarized in figure 12. The first two plots illustrates the tracking behavior of the piston position $x$ and the sum-pressure $p_{sum}$ control. The desired trajectories $x_d(t)$ and $p_d(t)$ are also changed dramatically in order to obtain the step response of the system. In the third plot the volume-flow $Q_A$ into the DAP are compared and the fourth plot compares the real piston velocity (with noise) and the estimated velocity from the numerical simulation.

**FIGURE 11:** Numerical simulation of the DAP control realized with Scicos ($T_p = 1$ ms, $T_s = 5$ ms)

**FIGURE 12:** Comparison of the numerical simulation results and the HIL measurements
5 Conclusions and Perspectives

The HIL simulation of an industrial motivated control demonstrated that rt-preempt patched Linux kernels, ScicosLab, Maxima, the presented open hardware Bioe and the web-based monitoring system prototype build a complete open rapid control prototyping framework which can be used for educational purposes. Due to the web-based approach there is no additional software except a modern web-browser required to interact with the real-time system. Furthermore, the web interface can be modified to the needs of the application by editing the web pages with a single text editor.

Improvements of the web-based monitoring system and the release of a ready to use version for the community with more human interface demos are planned. An other point of interest is to use this framework to realize a distributed parameter system benchmark example and to compare the results with measurements. Therefore, more complex Bioe interfaces will be used and the realization of 2D/3D visualizations are planned.

References


The Witch Navigator – A Software GNSS Receiver Built on Real-Time Linux

Petr Kacmarik
Department of Radio Engineering K13137, CTU FEE Prague
Technicka 2, 166 27 Praha 6, Czech Republic
petr.kacmarik@fel.cvut.cz

Pavel Kovar
Department of Radio Engineering K13137, CTU FEE Prague
Technicka 2, 166 27 Praha 6, Czech Republic
kovar@fel.cvut.cz

Ondrej Jakubov
Department of Radio Engineering K13137, CTU FEE Prague
Technicka 2, 166 27 Praha 6, Czech Republic
jakubon2@fel.cvut.cz

Frantisek Vejrazka
Department of Radio Engineering K13137, CTU FEE Prague
Technicka 2, 166 27 Praha 6, Czech Republic
vejrazka@fel.cvut.cz

Abstract

The Witch Navigator (WNav) is an open source project of GNSS (Global Navigation Satellite System) receiver whose hardware is implemented as an ExpressCard hosted in PC with Linux OS.

The employment of PC offers a possibility of an easy implementation of signal processing algorithms since almost no restrictions are introduced by a specific embedded platform (concerning memory requirements or real data type and its arithmetic). As a consequence, the WNav is especially suitable for researchers or students because the signal processing algorithms can there be implemented in asimilar manner as in high-level simulations. Furthermore, developers can rely on the wide and well known collection of development tools for Linux on x86 architecture. Unlike similar projects, WNav has capability to achieve performance comparable to professional GNSS receivers.

The WNav receiver is equipped with two front ends which can process any civil GNSS signals on two frequencies simultaneously. The whole receiver task is distributed between the device driver, user space real-time process and other axillary processes. The real-time needs are satisfied with RT_PREEMPT kernel patch.

The paper describes the whole conception of WNav with focus on the kernel part (device driver) and the real-time user space process, provides information about the processes synchronization and presents the achieved performance.

The first obvious milestone is to develop the fully functional GPS L1 C/A receiver which justifies the selected conception. The achieved results and experience with this legacy signal are presented in the paper, as well.
1 Introduction

It makes no sense to emphasize the importance of satellite navigation for everyday life. A satellite navigation system with worldwide coverage is termed as GNSS (Global Navigation Satellite System). Nowadays the term GNSS can represent several particular navigation systems. The most important one and well known is the United State GPS (Global Positioning System) since it can be considered as the only fully operable GNSS. For the WNav project, the implementation of the legacy GPS L1 C/A signal processing into WNav is important milestone that verifies the entire WNav project conception. The next GNSSa are EU’s Galileo, Russian’s Glonass and China’s Beidou.

The project of our software GNSS receiver, which is carried out at the Department of Radio Engineering FEE CTU, goes back to 2000. The WNav receiver is its latest contribution. Up to the WNav project beginning we tried several types of architectures. All of those were based on FPGA (Field Programmable Gate Array) which was supported with some processor (x86, PowerPC, MicroBlaze). There was established a simple rule how to divide the whole GNSS signal processing task between the FPGA and the processor: the direct signal manipulation at high sampling rate is performed in the FPGA while relatively slow but logically complicated operations are done in the processor.

At that time, we faced several drawbacks. If some FPGA development board together with PC were used, such solution showed low communication capability between this board and the PC. If the architecture was purely based on FPGA and processor core incorporated into the FPGA the main drawback was low computational performance of such processor core.

The WNav project tried to overcome most of the drawbacks. The utilization of PC (with x86 processor) takes advantage of a widely available architecture with sufficient computation power. The WNav receiver analog and FPGA part are implemented as an ExpressCard peripheral device. The ExpressCard internally relies on the PCI Express (PCIe) standard which ensures sufficient throughput between FPGA and the processor on PC. The ExpressCard now is relatively modern interface and we can expect its future with sufficient computation power. The WNav receiver analog and FPGA part are implemented as an ExpressCard peripheral device. The ExpressCard now is relatively modern interface and we can expect its future with sufficiently high throughput. The WNav processor is based on a modern processor core which will be useful to explain basic principles of satellite navigation.

Before we move to the detailed description of the WNav receiver and its particular components, it will be useful to explain basic principles of satellite navigation. Our goal is to prepare background for the time requirement explanation and also communication explanation between the receiver hardware (WNav device) and processor (PC workstation). More details concerning satellite navigation can be found in textbooks [1, 2].

Consider an unknown user position \((x, y, z)\) and known positions of satellites \((x_i, y_i, z_i)\), \(i \in 1, 2, \ldots\) (the satellite positions are computed from parameters of satellite tracks). In principle, the receiver obtains the distance \(d_i\) to the \(i\)-th satellite by measurement of the signal propagation time. If we measure distance \(d_i\) to 3 satellites with known positions \((x_i, y_i, z_i)\) we can obtain user position \((x, y, z)\) by solving the following system of nonlinear equations

\[
d_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}
\] (1)

The real situation is complicated since the receiver clock and satellite clocks are not synchronized. As a
consequence, the receiver is not capable of direct distance measurement but rather so called pseudorange $\rho_i$ which differs from true distance $d_i$ by an unknown bias $b$. The system of equations has then following form
\[
\rho_i - b = \sqrt{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2}
\] (2)
Since there are 4 unknown parameters (3 position coordinates and bias $b$) the receiver needs to perform minimally 4 measurements to find the solution.

How is the time measurement accomplished from the signal? The receiver generates identical signals as satellites. These locally generated signals are called signal replicas. The signal replicas are kept synchronized with the received signals. The time (delay) information is then carried in signal replica parameters needed for their generation so it is available in the receiver.

Let’s consider the following simplified model of the received signal
\[
s_i(t) = A d(t - \tau_i) c_i(t - \tau_i) \exp(j\omega_o t + \phi_0) + n(t),
\] (3)
where $A$ is a signal amplitude, $d(t)$ represents a navigation message, $c_i(t)$ is a pseudorandom code (PRN code) and $n(t)$ is an additive noise. The parameter $\tau_i$ represents the signal delay, $\omega_o$ is a frequency offset, $\phi_0$ arbitrary carrier phase in $t = 0$. The corresponding signal replica has the form
\[
r_i(t) = c_i(t - \tilde{\tau}_i) \exp(-j\hat{\omega}_o i t),
\] (4)
where $\tilde{\tau}_i$ is an estimation of the received signal delay. The $\tilde{\tau}_i$ is the key parameter needed for pseudorange formation $\rho_i$.

The specific correlation property of $c_i(t)$ enables to keep $s_i(t)$ and $r_i(t)$ synchronized, i.e. the estimation error $\tau_i - \tilde{\tau}_i$ is kept small.

In fact, the parameter $\tau_i$ can carry only a fractional part of the pseudorange $\rho_i$ due to $c_i(t)$ periodicity. Since $c_i(t)$ has period of 1ms the $\tau_i$ can only be measured in the range of 0 to 1 ms which corresponds to 0 to 300 m in distance. The pseudorange $\rho_i$ is extended using the bits of navigation message $d(t)$ and so called Z-count (time mark imprinted in $d(t)$).

The signal tracking is a stage of signal processing where a locally generated replica is kept synchronized with a received signal. The receiver usually utilizes cooperation of two feed-back systems for this task. For code and carrier synchronization the DLL and PLL feed-back systems are employed, respectively.

The simplified block diagram of the signal tracking is shown in Fig. 1. The key parts of both DLL and PLL are their detectors. The detector is a block which output is proportional to an error of tracking parameter, i.e. to $\tau - \tilde{\tau}$ for DLL or to $\varphi - \hat{\varphi}$ for PLL. The detector output drives (through the loop filter) particular NCO in the signal replica generator in other to minimize the error. The contemporary GNSS receiver has several such DLL/PLL blocks, each of them tracks one satellite signal.

Now we move towards the real implementation of the signal tracking as utilized in the WNav receiver. The modified block diagram of the algorithm is shown in Fig. 2.

The detector in GNSS receiver consists of several correlators (WNav employs just Early and Late correlators in one DLL/PLL block) and discriminator block. The correlator is a block which computes the mutual energy of the $s_i(t)$ and $r_i(t)$ over specified interval (given by period of $c_i(t)$, i.e. 1 ms in GPS L1 case). This interval is denoted as the integration time and, in WNav project, the instants of the Early and Late integration ends are denoted as Early and Late PRN TIC (E PRN TIC and L PRN TIC), respectively. The next block, discriminator, is usually non-linear memory-less block. Note that the DLL and PLL detectors have common correlators while discriminator blocks differ. The DLL and PLL detectors are marked as the blue and red areas, respectively.

FIGURE 1: Simplified block diagram of GNSS signal tracking – DLL and PLL structure
The interface between the FPGA and the PC workstation divides the entire tracking algorithm into a high frequency part processed in the FPGA and a slow part processed in the PC. This interface is outlined with a gray line in the figure. The interface output/input registers are marked as gray boxes in the figure.

The correlators perform signal down-sampling. While the correlators treat the sampled signals at their inputs (at 20 MHz rate in WNav case), their outputs are issued at integration time rate, i.e. at 1 kHz. The relatively slow correlator outputs are read from appropriate registers (Early/Late correlator output registers) and the rest of the loops are implemented as software in the PC. The local replica generator, since treat sampled signal, has to be implemented in the FPGA. The replica generator is driven through the NCO code/carrier control registers. The signal delay can be read from the code measurement register (and also from the carrier measurement register in case of precise carrier phase measurement).

The Early/Late correlator output registers issue new values at every E/L PRN TIC event (approximately every 1 ms). These values have to be sent to PC workstation and new NCO code/carrier registers have to be received. The receiver has several such DLL/PLL blocks, and this type of register transfer has to be accomplished for all of them. Since each DLL/PLL block tracks different satellite signal, the E/L PRN TIC events of particular blocks are not synchronized. To make the transfer feasible, the WNav utilize a resampling conception: registers of all DLL/PLL blocks are sampled at a slightly higher rate than E/L PRN TIC and at this time all register values are interchanged between the FPGA and the PC. The software on the PC side then does recognition whether new correlator output values were received as a consequence of E/L PRN TIC of a particular DLL/PLL block. The resampling runs at 800 µs rate and in the WNav project is denoted as TIC event. We will also discuss the TIC event later since it is an important event which drives the FPGA-PC communication.

Signal tracking can not be an initial stage of signal processing. The DLL/PLL supposes that the parameter error is small enough and then the detector can produce a meaningful output which drives the NCO in the right way. The stage which has to precede the signal tracking is denoted as signal acquisition. The task of the acquisition is to provide a coarse estimate of a signal parameter to seamless transition to the tracking stage.

The WNav receiver relies on parallel acquisition algorithm. The FPGA part therefore contains snapshot memory, where the sampled complex envelope of the received signal is stored. Since the acquisition algorithm is completely implemented in the PC, the contents of the snapshot memory have to be also transferred to the PC. If we consider In-phase (I) and Quadrature (Q) signal components, eight bit samples at 20 MHz each, we obtain $2 \times 20 \times 10^6 \times 800 \times 10^{-6} = 32000 = 0x7d00$ bytes as a size of the snapshot memory which has to be entirely transferred to PC at every TIC.

## 3 Description of the WNav Receiver

In this section, we will describe particular parts of the WNav receiver. We will start with the receiver hardware. Next, the device driver will be described and finally we will turn our attention to the software (processes) in user space. Here, the provided information introduces WNav as a Linux project. The signal processing and GNSS perspective of the WNav receiver can be found in [3, 4].
3.1 Receiver Hardware

The WNav hardware consists of a peripheral device plugged into the PC workstation. The device is implemented as an ExpressCard/54 (L-shape). Two different WNav device prototypes can be seen in Fig. 3.

**FIGURE 3: Two different versions of the WNav ExpressCard device prototypes**

The WNav device consists of an analog part and FPGA part. The FPGA part is responsible for digital signal processing and communication via PCI Express (PCIe).

![Functional block diagram of the WNav ExpressCard peripheral device.](image)

**FIGURE 4: Functional block diagram of the WNav ExpressCard peripheral device.**

The device is equipped with two RF (Radio Frequency) inputs with MMCX connector, thus, signals from two antennas can be processed simultaneously. The analog receiver part utilizes direct conversion concept (MAX 2120), so the complex envelope (I&Q components) are fed into A/D converter (MAX 1192). The 8-bit samples at sampling rate of 20 MHz are used for each component. The next signal processing is accomplished in the FPGA. The WNav is now built on Xilinx Spartan 6 FPGA (XC6SLX45T).

The key elements of the FPGA part are DLL/PLL correlator blocks as described above (see Fig. 2). These blocks are organized into groups of six and, on the higher level, there are four such groups in WNav. Thus the WNav receiver has capability to track $6 \times 4 = 24$ satellite signals simultaneously.

Except the correlator blocks, the FPGA part contains the other blocks as snapshot memory for signal acquisition purpose, and an I2C block which can control the direct-conversion tuners.

All input and output registers and snapshot memory are arranged in such a way that they can be accessible through I/O memory mapped mechanism from the PC side. The device offers one I/O memory region which is common for reading and writing operations. But, the read and write operations with identical address access different memory cells in the device (generally, a value which was written to the device can not be read back from the device at the same address). The arrangement of the input and output registers into I/O memory space as viewed from the PC side can be seen in Fig. 6 and Fig. 7. To ensure register visibility from the PC side, the FPGA part contains the communication block which forms and processes TLP (Transaction Layer Packets) packets and transforms them to the I/O memory operations.

The memory region for the read operation is large due to the snapshot memory size. To meet strict time requirement the transfer from the device to the PC is accomplished with DMA and the communication block is equipped with a simple DMA controller.

The communication with the device is synchronized with the TIC event. The TIC event occurs every $800 \mu s$ and at this instant, the new values can be read and written through I/O memory. The TIC event is propagated to the PC side by MSI interrupt (Message Signaled Interrupt) which is generated at the end of DMA transfer. The time relation of the TIC event and the interrupt is depicted in Fig. 5.

The correctly plugged WNav device can be seen in a list of PCI devices:

```bash
$ lspci -v
...
18:00.0
  RAM memory: Xilinx Corporation Zomojo Z1
  Subsystem: Xilinx Corporation Zomojo Z1
  Physical Slot: 1
```
Flags: bus master, fast devsel, latency 0, IRQ 44
Memory at e4000000 (32-bit, non-prefetchable) [size=1M]
Capabilities: [40] Power Management version 3
Capabilities: [48] MSI: Enable Count=1/1
Maskable- 64bit+
Capabilities: [58] Express Endpoint, MSI 00
Capabilities: [100] Device Serial Number
00-00-00-00-00-00-00-00
Kernel driver in use: wnav

3.2 Device Driver – Kernel Module

The device driver was written based on information in [5, 6]. Other up to date information was obtained using a Linux identifier search server [7].

The WNav device driver is implemented as a character device driver. When module is loaded, the plugged WNav card is accessible through the device file /dev/wnav0. The kernel message, when the device was plugged, is shown here (there was one WNav device detected):

wnav: wnav_module_init() BEGIN
wnav: wnav_pci_probe() BEGIN
wnav: /dev/wnav0 created for device
wnav 0000:18:00.0: enabling device (0000 -> 0002)
wnav 0000:18:00.0: PCI INT A -> GSI 18 (level, low) -> IRQ 18
wnav: resource start: 0x0000000e000000
wnav: resource end: 0x0000000e40fffff
wnav: resource flags: 0x00000000040200
wnav: resource length:0x00000001000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: resource start: 0x000000e4000000
wnav: resource end: 0x000000e40fffff
wnav: resource flags: 0x00000000000000
wnav: resource length:0x00000000000000
wnav: probe function ends with success
wnav: --> struct wnav_dev ptr: 0xf44f4000
wnav: wnav_pci_probe() END
wnav: wnav_module_init() BEGIN
wnav: module init ends with success
wnav: Number of recognized WNav devices: 1
wnav: --> &dev[0] 0xf44f4000
wnav: --> &dev[1] Ox (null)
wnav: --> &dev[7] Ox (null)
wnav: wnav_module_init() END

Internally, the module data are stored into two structures. On the top, there is a structure struct wnav_dri, which gathers common data for entire driver. The second structure, struct wnav_dev, gathers data for each particular device (for one plugged WNav card) into a system; it is supposed, that there can be more WNav cards plugged in one PC. The struct wnav_dri contains an array of pointers to struct wnav_dev as a one of its items.

Most of struct wnav_dev items are filled in wnav_pci_probe() function call invoked after the device plugging. The important items are addresses for accessing I/O memory of the WNav device. The hardware address baddr_hw is obtained from pci_resource_start(), and is mapped using pcim_iomap() to obtain virtual address baddr_vir, which is used for access from the driver side. The next two addresses are related to the DMA transfer. There is dma_hw address which has to be sent into the device (the DMA controller in the FPGA needs this) and dma_vir address, which is used for access the DMA region from driver side. Both of them are obtained as a result of pci_alloc_consistent() call.

The device driver counterpart in the user space is a RT process wnav_core, which is mainly responsible for channel services, i.e. the closing feedback of the DLLs/PLLs. The driver implements following system calls: open(), close(), read() and write(). The RT process calls read() and write() periodically and between these two calls the channel services are accomplished (more wnav_core details will be provided in 3.3.1). The time relation of the FPGA, driver and RT process is depicted in Fig. 5. The behavior of read() and write() system calls is depended on the device status stored in variable stat (item in struct wnav_dev). The stat can be one of the following: WN_READ, WN_WRITE and WN_DONE.

We describe driver function according Fig. 5. Consider that we are in instant of TIC. The RT process is sleeping now (in waiting queue), since it called read() and status was WN_DONE. However, after the TIC new values are available through I/O memory. But it is not supposed that the I/O memory would be accessible now with functions as ioread32() or iowrite32(). Instead, the DMA transfer is initiated after the TIC event. Entire read block, as shown in Fig. 6, is then available in the kernel space (driver). The transfer end is signalized by the MSI interrupt. The interrupt handler changes the status from WN_DONE to WN_READ. Since the status in WN_READ is the condition for RT process wake up, the RT process is removed from the waiting queue and can now continue in reading. The read block is then copied into the user space with copy_to_user() function. Except the data from the FPGA (output registers, snapshot memory block), TMARK and FFLAG blocks are also added. The TMARK and
FFLAG contain timing information and fault flags from previous read() & write() cycle. This is a way, how to make available these useful data for performance debugging in user space (we will mention both of them later). When the entire read block is transferred into the user space, the status is changed from WN_READ to WN_WRITE. Next possible attempt of read ends with an error. Then, the RT process does channel services and prepares data for writing. The arrangement of the write block can be seen in Fig. 7. Since the write block is significantly smaller than the read block, no DMA transfer mechanism is implemented due to simplicity. The write() system call is implemented into two stages. First, the write block is copied into the kernel space with copy_from_user(). Next, depending of write block contents, the data are copied from the kernel space to the device with iowrite32(). When all data are written, the status is changed from WN_WRITE to WN_DONE. Next possible attempt of write ends with an error. When the RT process calls read() in this time, the RT process is put into waiting queue due to the status in WN_DONE.

FIGURE 5: Time relations during the data transfer, cooperation of the WNav hardware, device driver (kernel module) and user space RT process

The previous paragraph describes the desired situation, when all the PC tasks after the MSI interrupt are finished before the next TIC. Despite utilization of Real Time Linux kernel (RT_PREEMPT) [8], such strict time requirement can not be ensured in all cases and conditions. The strategy of WNav driver is to announce the situation when time constraints were broken. Further, the driver offers time measurement of particular drive statuses. This information is available in user space due to mentioned TMARK and FFLAG blocks.

The breaking of time constraints is gathered in struct faultflags, its instance FFLAG is an item in struct wanv_dev. The FFLAG contains error counters or cycle slip counters, i.e. if all items remain in zeros the time constraints are not broken. The driver checks that in two successive interrupt handler calls the TIC counters differ exactly by one. Next, the TIC event can be detected in the driver by reading the TIC counter directly from I/O memory (not by reading the DMA memory region). The driver checks that in particular driver status (interrupt handler, reading and writing) the TIC counter read from I/O memory does not differ with the TIC counter obtained from the DMA memory region.

To analyze time consumptions in particular statuses the driver stores time markers (using get_cycles() function). The markers form items of struct time_mark, its instance TMARK is then a item in struct wanv_dev. The driver stores markers at the beginning and end of the interrupt handler, at the beginning and end of read() system call, at the time when the RT process wakes up and at the beginning and end of write() system call.
3.3 User Space Processes

Our goal now is to describe the processes which run in the user space context. We describe just coarse conception. The detailed information is available on the project homepage [12] where the source code documented with Doxygen [11] tool is placed.

The WNav project, in the first approach, consists of three main user space processes. We already mentioned the RT process labeled as \texttt{wnav\_core} which is mainly responsible for the channel services. The second one is a user interface and offers a look inside of WNav but also provides WNav’s control facility. The process is labeled as \texttt{wnav\_monitor}. The third one is a process responsible for position, velocity and time (PVT) estimation and is labeled as
wnav_pvt. All of these three processes can be run separately. Of course, to allow a meaningful operation of wnav_monitor or wnav_pvt, the wnav_core has to be also running.

The Inter-Process Communication (IPC) in the WNav project is based on shared memory. As a synchronization object of shared resources (items in shared memory) we simply rely on integer variables which are treated atomically (we used gcc built-in atomic functions). A shared memory top structure struct wnav_shamem can be used as an outline, which data are shared among the processes:

```c
struct wnav_shamem
{
    /* --- IO memory --- */
    struct acq_mem acq;
    struct rd_restbuff rd_rest;
    struct wr_buff wr;
    /* --- --- */
    struct prn gener prn;
    struct wnav_tuner tuner;
    struct wnav_core wcore;
    /* --- channels (correlators) --- */
    wnav_corr_t corr_all[WN_CORR_NO_GRP][WN_CORR_IN_GRP];
    /* --- monitor --- */
    struct wnc_llist llist;
    /* --- pvt process --- */
    struct pvt process pvt;
};
```

### 3.3.1 RT Process: wnav_core

The RT process is responsible for several tasks which are accomplished in infinite loop. We describe them in next paragraphs.

First, the data are read from the device file /dev/wnav0. This task is covered in a function do_read(). The reading is accomplished into two steps (there are two read() system calls in do_read()), see Fig. 6. In the first step, the block of snapshot memory is read. Such block contains signal samples over 800 µs (interval between two successive TICs). The block is stored as one element of an array in struct acq_mem. This array organizes successive snapshot memory blocks to result into a region with signal samples over long interval equals to several multiples of 800 µs. The second step of reading get the code and carrier measurements, correlator outputs but also debugging and performance related information from kernel driver. These data are stored in struct rd_restbuff.

Next task of the RT process is connected with the receiver control through the user interface. This task is accomplished in function wnc_llist_perfrmv(). In the shared memory, a queue of tasks (struct wnc_llist) is implemented. This queue is filled from the user interface. Here, in the RT process, the task from queue is performed, and if the task is finished, the it is removed from the queue.

The following work of the RT process is the channel services. The signal processing related data (filter status, channel status, accumulated correlator outputs, etc.) are stored in structure wnav_corr_t. The array of wnav_corr_t with identical organization as DLL/PLL blocks in FPGA (i.e. $4 \times 6$) is an item of the structure struct wnav_shamem. The signal processing task is driven according the channel status, item chst in wnav_corr_t. Based on the available correlator outputs from struct rd_restbuff and data in wnav_corr_t, new values for code and carrier NCOs are prepared.

Another important task of the RT process is an export of code measurement for pseudorange forming. These data are stored in struct pvt_share and then are read by PVT process.

The final task is to write the data back to the FPGA through the device file /dev/wnav0. The task is accomplished in a function do_write(). It may be divided into two parts, see Fig. 7. If just new NCO values and possible one register value have to be written, the writing is accomplished with one write() system call. All needed data are prepared in struct wr_buff. In such case, when the PRN code has to be written in addition, the write is divided into two steps. In the first one, the header and NCO values are written and then a segment of PRN code is written.

### 3.3.2 Process of User Interface: wnav_monitor

The user interface is based on Ncurses library [10], see screenshot in Fig. 8.

The first task of the user interface is to make possible a look inside the receiver. The displayed information is organized into pages. In the screenshot, the channel status related data are displayed. There are other pages like an acquisition page, tuner status page, fault flag page and help page.

The second task of the user interface is a receiver control. The user can control the receiver by typing the commands into a command line (see the last line in the screenshot). The command consists of its identification (string) and argument lists, e.g. ACQ 0 0 starts the acquisition for the first channel in the first group (channel with coordinates 0, 0), or TUNER 0 1575.42 4.0 10.0 sets
the frequency, bandwidth and gain for the first tuner (with 0 id). Since most tasks need some write and read to/from the FPGA, the monitor process just converts commands into task objects and put them into the queue. The queue is internally implemented with struct wnav_list in the shared memory. The function wnc_list_add() performs a command parsing and putting task into this queue. Such created task objects are then retrieved in wnav_core with wnc_list_perform() function.

FIGURE 8: wnav_monitor process screenshot: page with channel statuses

3.3.3 Position, Velocity and Time Estimation Process: wnav_pvt

Process wnav_pvt performs an estimation of the receiver position, velocity and time. Internally, the algorithm is based on Eq. 2 which is solved using the least square method.

For proper function the process wnav_pvt needs data from the RT process wnav_core. The communication is accomplished using struct pvt_share placed in the shared memory.

The items of struct pvt_share can be seen from next code:

```c
struct pvt_share {
  uint8_t count_m125;
  uint8_t core_idx;
  uint8_t pvt_idx;
  enum praw_stat stat[PVT_RAW_CNT];
  PVT_RAW raw[PVT_RAW_CNT];
};
```

The communication utilizes a circular buffer conception. The size of the buffer is determined with PVT_RAW_CNT. The data for the PVT process are encapsulated in raw array. The structure has two indexes, core_idx and pvt_idx, which point to places where wnav_core and wnav_pvt shall perform write and read, respectively. The items in raw array are protected with a lock (or status) stat. The lock stat indicates whether the corresponding item in raw array is free, locked for writing by wnav_core, contains consistent data or is locked for reading by wnav_pvt.

The data export from wnav_core is controlled by count_m125. This is TIC counter modulo 125 and data are exported at its overflow, i.e. each $125 \times 800 \mu s = 100 \text{ms}$. Thus, 100 ms is an interval between two successive position, velocity and time estimation of the WNav receiver.

<table>
<thead>
<tr>
<th></th>
<th>PC workstation 1</th>
<th>PC workstation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC workstation type</td>
<td>Intel(R) Core(TM)2 CPU</td>
<td>Genuine Intel(R) CPU</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1866.669 MHz</td>
<td>1833.337 MHz</td>
</tr>
<tr>
<td>CPU cores</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>cache L1/L2</td>
<td>32/4096 KB</td>
<td>64/2048 KB</td>
</tr>
<tr>
<td>bogoMIPS</td>
<td>3732.89</td>
<td>3657.46</td>
</tr>
<tr>
<td>address size: physical/virtual</td>
<td>36/48</td>
<td>32/32</td>
</tr>
<tr>
<td>system memory</td>
<td>2G</td>
<td>1G</td>
</tr>
<tr>
<td>distribution</td>
<td>Fedora 12 (Constantine)</td>
<td>Fedora 15 (Lovelock)</td>
</tr>
<tr>
<td>kernel</td>
<td>2.6.33.7-rt29 SMP PREEMPT RT</td>
<td>3.0.1-rt11 SMP PREEMPT RT</td>
</tr>
<tr>
<td>hardware platform</td>
<td>x86-64</td>
<td>i386</td>
</tr>
</tbody>
</table>

TABLE 1: Parameters of PC workstations used for WNav receiver testing
4 WNav Receiver Testing

The WNav receiver was intensively tested on two different PC platforms. The first of them was a desktop PC (gaming computer), further labeled as PC 1, the second of them was a laptop, further labeled as PC 2. The PC parameters are enumerated in Tab 1. On both PC platforms, we were capable to put the WNav receiver into operation.

The test configuration for PC 1 can be seen in Fig. 9. We used Spirent simulator GSS6560[13] as a GPS L1 C/A signal source. During the testing the important receiver parameters were logged for later analysis and visualization. See Fig. 10, where the correlator outputs of one channel are depicted. The data for the figure were acquired from wnav_core process.

FIGURE 9: The WNav receiver tested with a signal from Spirent simulator

The time constraints in the receiver were also monitored using build-in facility in the kernel driver. However, breaking of most time constraints is clearly visible since such situation issues in a malfunction of the receiver. The synchronization of all tracked signals is then lost and the receiver has to be restarted.

While the first PC platform (PC 1) worked directly after using custom kernel with RT_PREEMPT patch, it was not true in case of PC 2. To force the WNav receiver to operation on PC 2, we had to do a lot of operation system and kernel related experiments and tunings.

For PC 2 we took into consideration all available recommendation related to RT_PREEMPT patch, mainly from [8, 9]. We did some experiments with kernel parameters, tried to reduce PC load by removing unnecessary demons and service and changed the interrupt thread and RT process real time priority and CPU affinity.

Unfortunately, we have not yet gathered enough information to reliable answer which tuning mechanism or parameters have key impact on the WNav receiver. The kernel and operation system tunings for WNav are challenges for the future. Clearly, it will always be a trade off between the PC hardware performance and amount of work needed for kernel and operation system tunings.

FIGURE 10: Visualization of Early and Late correlator outputs; 1) Early correlator outputs during the test, 2) Late correlator outputs: detail with synchronized carrier phase, 3) Early power and Late power signals: detail immediately after a signal detection

5 Conclusion

In this paper, the Witch Navigator GNSS software receiver was described. The receiver hardware (device) is implemented as an ExpressCard intended for PC workstation running Linux kernel with RT_PREEMPT patch. The first obvious milestone of the project is an implementation of GPS L1 C/A capable algorithms which was already accomplished. The conducted experiments prove, that the key receiver elements (ExpressCard, FPGA, x86, Linux kernel) have enough capability not only for GNSS demanding signal processing tasks but also the capability to meet strict time constraints needed for DLL/PLL controlling.

The paper brings details concerning receiver implementation as a software project in the Linux operation system. The communication with the device, device driver, user space processes and their IPC were described.
The WNav project is developed completely with free tools. This applies to both hardware (PCB design, FPGA programming) and software. The WNav project is open source project. Its source code, documentation and other related materials will be available on the project’s homepage [12].

Further project development has two obvious directions. The first of them is the project development in terms of GNSS, i.e. to introduce algorithms for new GNSS signals and systems (now, just GPS L1 C/A has been implemented). The second of them is a project development in terms of software implementation, i.e. improving an ineffective implementation of the algorithms, gathering the information how to configure the kernel and operation system, maintaining the code, keeping up to date documentations etc. We hope that in both developing directions we will utilize the feedback of other potential users of the WNav receiver.

Acknowledgments

The authors would like to thank the Spirent Communication for lending the GPS simulator GSS6560, which was used for the verification and testing of the final tracking and PVT algorithms.

References


Case Study: Challenges and Benefits in Integrating Real Time patch in PowerPc Based Media System

Manikandan Ramachandran
Infosys Technologies Limited
Electronic City, Hosur Road, Bengaluru 560100, India
manikandan@infosys.com

Aviral Pandey
Motorola Mobility
2450 Walsh Avenue, Santa Clara, CA 95051, USA
aviral.pandey@motorola.com

Abstract

Media systems generally have many CPU intensive as well as time critical processes. Vanilla Linux 2.6 with preemption enabled does provide solution to this kind of system. However, if the system is interrupt intensive, as ours, then vanilla Linux 2.6 performance is not expected to be good; as by definition interrupt preempts all higher priority tasks. RT patch seems to address exact issue by providing an option to handle interrupts in process context, but the solution doesn’t seem to fit customized Linux. Quite a bit of architecture changes had to be made to reap the benefits of RT patch.

This paper describes about various challenges faced in integrating Ingo’s RT patch on a customized PowerPc Linux package. And explains how those challenges were overcome. It describes how LTTng can be used to identify the bottlenecks and finally concludes by comparing performance of the application that was run on vanilla and RT Linux.
1 Introduction

Linux operating system is evolving fast as a de-facto embedded operating system. Looking back the community has made tremendous progress from being stuck with big-o-lock to a really preemptible kernel in 2.6, and to fairly predictable kernel with RT patch. However, still there are few challenges in using Linux for commercial systems that require real-time capabilities.

This paper takes one of such case and walks through the challenges in integrating RT patch along with slew of other patches. After all the effort in integrating, the real performance of the system is not up to the expectation. The paper dwells deeper into a performance issue and proposes a solution to fix that issue.

2 Product Overview

The system identified for this study is a head-end media device. The device is a real time video splicer and statistical remultiplexer that can groom hundreds of video services. The device has 2 PowerPc 7448, built on a Marvell communication controller board [MV64460]. Real MPEG related operation is done by a proprietary application that is highly CPU intensive and it has the highest priority among all processes. In short, proprietary application “almost” expects hard real time capability from the operating system. The figure below gives intended hierarchy of CPU resource to be utilized by various processes/interrupts.

3 Problem Statement

The most important goal of the product software design is that whenever there is an interrupt from video hardware, kernel has to schedule video processing tasks with almost zero latency. In theory assigning high real time priority should take care of this requirement; however in few instances it has been demonstrated that Linux kernel has failed to honor priority of tasks because of various other kernel dependencies.

Linux Trace Tool next generation or LTTng is an open source tool that helps to find performance bottle neck on Linux based system. Using LTTng, following system bottlenecks were identified:

3.1 Multiple IDE interrupts

The media product has two compact flashes each of which can be accessed through kernel’s IDE driver. Any access to compact flash generates lot of interrupts that could preempt the high priority application process and cause degradation in system performance. This issue was kind of mitigated by deferring IDE request by 1 millisecond, think of it like yielding CPU to other tasks before scheduling another IDE request. This feature ensures that there is only one IDE interrupt during 1 millisecond window. This solution seems to work; however in certain condition, when there is excessive IDE access, LTTng shows that more than 1 IDE interrupt could occur in 1 millisecond window.

This scenario happens when video hardware interrupts and IDE interrupts are handled one after the other.

Figure 2[Refer Appendix A for legend] is a capture of LTTng viewer. "Marker 1" and "Marker
2". shows that there are more than 1 IDE interrupts while application process is woken up by video interrupt. "Marker 3" just shows the events that occurred around "Marker 1".

3.2 Softirqs preempts high priority process

Linux kernel is driven by timer interrupts. Whenever there is timer interrupt, scheduler is woken up and deferred tasks are executed from softirq context. The occurrences of timer interrupt can be configured. In our kernel we have configured that to be 1 millisecond. So every 1 millisecond, "softirq" daemon is woken up and it preempts all other high priority tasks. Usually "softirq" daemon doesn’t hog CPU [it would take 30-40 microseconds], but if there are many kernel soft timer registered it could hog the CPU.

3.3 Spinlocks and Preemption

In multiprocessor system spinlocks are used to provide mutual exclusion for any critical resource. Spinlocks by its nature disables preemption in CPU of the current process. So if a lower priority process happens to take spinlock before a higher priority tasks scheduled to run, then as long as spinlock is held by the lower priority task, higher priority task is starved for CPU time. This is one scenario, where priority of process fails to get honored.

In our system there are 2 scenarios in which spinlock usage is inevitable:

- Custom drivers handling interrupts or to provide mutual exclusion to critical resource.
- Kernel usage of spinlocks. One common instance is usage of spinlocks in "printk" function.

4 A Case for Real-Time Patch

Real-time patch [3] provides following features that make the kernel more preemptible:

- Makes spinlocks preemptible by default.
- Option to handle IRQs in kernel thread context instead of IRQ context.
- "Softirqs" are handled in kernel threads context.

Our study about system performance issues with vanilla kernel 2.6.33 made a compelling argument for using RT patch. We thought by using RT patch we could address our system performance issues. Based on that thought we made following changes to our system software architecture:

- Apply RT patch to our kernel and make drivers compatible to that.
- All interrupts except video hardware interrupts were made to handle as kernel threads.
- Priority of Application process was made higher than interrupt and softirq threads.

The idea of new design is that whenever there is a video interrupt, media process is woken up with zero or little latency. Another expectation from RT patch is that, the feedback mechanism used in IDE driver to throttle IDE request can be removed and RT patch would inherently take care of preempting low priority tasks including tasks that handle interrupts.

5 Challenges in Integrating RT-patch

The end goal of this exercise was to have a stable real-time Linux kernel and have performance profiling capability by integrating LTTng. At the time of this exercise 2.6.33 was the latest and stable kernel
version so we choose real-time patch version 2.6.33.9 and LTTng version [version]. Patch process went about quite smoothly, but we had following run time issues:

1 **Unable to boot the System in SMP mode:**
   The cause of this issue was found to be with calling "kzalloc" from "setup_cpu". Apparently this issue is not seen in non-real-time kernel. We worked around this issue by statically allocating memory rather than using kzalloc or kmalloc.

2 **Dependency on BSP code:** When used real-time patch over Marvell’s BSP code, we found lot of recursive locks issues. We identified those issues and fixed them.

3 **Handling IRQ in a Thread:** This is one of the toughest challenges that we are dealing with. In first look, after booting real-time kernel one would think that interrupts are handled in Interrupt threads, but we found it in hard way that interrupts continue to get handled in interrupt context unless a few specific changes are made[4].

   This issue is the main point of this paper, and it is discussed in detail in section “Performance Analysis”.

6 **Performance Measure**

   After applying real-time patch we tried to get raw performance measure of our system using "Cyclic testr" [5] and the actual application performance in a controlled test environment. From cyclic test, we noticed that the performance of the system was similar to vanilla kernel. However, with our custom product performance test we found that the performance of the application has deteriorated a lot. Following section describes our test scenario and results.

6.1 **Kernel Performance Measure**

   "Cyclic" test is used to determine the internal worst-case latency of a Linux real-time system."Cyclitest" measures the amount of time that passes between when a timer expires and when the thread actually runs the thread is woken[9].

   We used this tool in both patched real-time kernel and vanilla kernel. The "Cyclic test" test was run with following parameter set:

   - one thread with priority 80
   - Thread to run 10000 tight loops
   - Interval between each loops 10000 micro seconds

   We conducted this test in 2 phases. In first phase, all application was stopped and made sure CPU utilization was less than 1% on both CPU. Then we ran cyclic test first with vanilla kernel then cyclic test was ran on kernel with real-time patch. Results of the test are given in Figure-4.

   ![Cyclic Test with No Load](image1.png)

   **FIGURE 4:** Cyclic Test with No Load

   In second phase, both CPUs of the system were heavily loaded using a script that just creates a tight loop. Multiple instances of this script were run concurrently till the CPU utilization of the system reached 97%. Cyclic test was run in this loaded scenario. Figure 5 gives the result of cyclic test on both kernels.

   ![Cyclic Test with Load](image2.png)

   **FIGURE 5:** Cyclic Test with Load

6.1.1 **Inference**

   From figure 4 we see that the minimum latency for vanilla kernel is 896 microseconds which is less than real-time patched kernel. While maximum latency for real-time patched kernel is more than 1milliseonds which is far worst than vanilla kernel.

   From figure 5, we can infer that latency isn’t bad with loaded system. Both for vanilla and real-patched kernel we see the difference is only about 10 micro seconds.
The conclusion of this test is that we don’t observe huge improvement in performance of the kernel/system with real-time patch.

6.2 Application Performance Measure

The idea behind this performance test is to study how vanilla and real-time patched kernel behaves when a high priority process is made to starve for CPU time. Since real-time patch has option to handle interrupts in process context and makes normal spinlocks preemptible, the expectation and the desire was that the real-time patched kernel always honors processes’ priority irrespective of other processes or kernel state.

In our case, Media application is configured to have higher priority than all other processes, including processes that are supposed to handle interrupts.

6.2.1 Test Case Description

The Media application was configured to run at full capacity, so that CPU resource is solely taken by this process. Executions of other trivial tasks were reduced. The test was then run on 3 different kernels:

- 2.6.22.5 [scheduler based on RB-tree]
- 2.6.33.5 [CFS scheduler]
- 2.6.33.5 + RT patch

![FIGURE 6: CPU0 load comparison](image1)

![FIGURE 7: CPU1 load comparison](image2)

Figure 6 and 7 compares the performance of each kernel while the process was stressed for 30 minutes.

6.2.2 Inference

From Figure 6 and Figure 7 it is apparent that application performance on real-time patch kernel is not better than its performance on vanilla kernel. Only difference between real-time patch kernel and vanilla kernel from a platform perspective is that a patch to throttle IDE access was not ported to real-time kernel.

To make sure IDE access is the cause of the issue, excessive IDE access was made when media application process was fully loaded. Under this scenario, we found that the performance of the application deteriorated further. Ideally a process that handles IDE interrupts should have yielded to high priority task processes like the media application process; however, from this test it looks like media application process seems to be preempted when there is an IDE interrupt.

7 Application Performance Analysis

After making the kernel real-time enabled, the expectation was that the IRQ thread will be preempted by high priority media application process. However, as demonstrated in previous section, we have seen that performance of the application was not good with real-time patched kernel. There could be two possibilities for this kind of behavior:

1. Scheduler is not honoring processes priority.
2. IDE interrupts are still handled in interrupt context like vanilla kernel.

We confirmed "Case 1" is not the case by running "rt-migrate-test" written by Steven Rostedt[6]. This is a simple program that creates multiple threads with various priority and then checks if scheduler honors the priority of each thread. This test passed both in normal scenario and in heavily loaded system with IDE interrupts and media application process.

To check if IDE interrupts are still handled in interrupt context, we patched real-time kernel with LTTng. Then with patched kernel, the media application was stressed while generating multiple IDE interrupts.

![FIGURE 8: Video Process being preempted by interrupts](image)

Figure 8 captures the various process states at the time of the issue. "irq75-idle1" is the handler process for the IDE interrupt: in this case IDE interrupt number is 75. From the viewer, we can interpret that video process was starved for CPU as it was busy handling IDE interrupt "75" in interrupt context rather than in the context of the thread "irq75-idle1". This is the cause of video process starvation.

8 Solution

We started looking into "request_irq" implementation in real-time patched kernel to understand why IDE interrupts were handled in interrupt context not in thread context. From "request_irq" implementation it was clear that irq thread did nothing if handlers for an interrupt were installed through "request_irq" call. In order for interrupt to be handled in thread, one has to use "request_threaded_irq" and split the handling of irq into two parts by using two handler functions. In first part, interrupt will be handled in interrupt context. In this context, handler should disable the source of the interrupt and wake corresponding interrupt thread. The second handler should do the actual handling of the interrupt, which obviously will be done in the thread context.

Having identified the bottleneck, we are in process of converting IDE and other similar CPU intensive interrupt handlers to be run in thread context.

9 Future Work

9.1 Creating thread function for interrupt handling

As discussed in section 8, just having real-time patch is not go enough for making interrupts to be handled in threads. A thread function, which handles deferred interrupt work, should be designed and implemented. In future, we intend to make low priority interrupts that are CPU intensive to be handled in threads.

9.2 High Resolution Timers

We didn’t enable High Resolution Timers. in any of the current kernel configuration as we weren’t sure about the support of this feature in PowerPc based system [8]. In future, we intend to do deep profile of kernel performance with high resolution timers.

10 Conclusion

Effectiveness of real-time patch on a system depends on various factors. In our case, preemption of IRQ handler is the single most important criteria in determining system performance. Although real-time patch creates interrupt threads by default, it really doesn’t mean that interrupts are handled in those threads. This behavior confuses the user. Instead, it would have been better if no interrupt threads were created. If one prefers to have an interrupt thread then he or she may do so by using "request_threaded_irq".

This case study shows that applying real-time patch will not solve all real time requirements of a system. Instead it provides great tools like preemptive spinlocks, threaded interrupt handler etcetera.
To make a system near real time capable, one has to understand their system bottlenecks. LTTng is a great tool which clearly brings out hard to understand system behavior. As demonstrated in this case, LTTng was used extensively to get to bottom of many performance issues. It is highly recommended to use LTTng to understand system performance issues and make few system software architecture changes to take advantage of tools provided by real-time patch. To summarize, real-time patch is not a "cure-all" of all system real-time owes; instead it provides great tools that could be used as the first step to make a system real time capable.

References

[7] Investigating latency effects of the Linux real-time Preemption Patchs (PREEMPT RT) on AMDs GEODE LX Platform

A Appendix

Following is the legend for LTTV output. Source: ”http://ltt.openrapids.net/lttv-doc/user_guide/x81.html”.

FIGURE 9: Legend For LTTV
Hard real-time Control and Coordination of Robot Tasks using Lua

Markus Klotzbuecher
Katholieke Universiteit Leuven
Celestijnenlaan 300B, Leuven, Belgium
markus.klotzbuecher@mech.kuleuven.be

Herman Bruyninckx
Katholieke Universiteit Leuven
Celestijnenlaan 300B, Leuven, Belgium
herman.bruyninckx@mech.kuleuven.be

Abstract

Control and Coordination in industrial robot applications operating under hard real-time constraints is traditionally implemented in languages such as C/C++ or Ada. We present an approach to use Lua, a lightweight and single threaded extension language that has been integrated in the Orocos RTT framework. Using Lua has several advantages: increasing robustness by automatic memory management and preventing pointer related programming errors, supporting inexperienced users by offering a simpler syntax and permitting dynamic changes to running systems. However, to achieve deterministic temporal behavior, the main challenge is dealing with allocation and recuperation of memory. We describe a practical approach to real-time memory management for the use case of Coordination. We carry out several experiments to validate this approach qualitatively and quantitatively and provide robotics engineers the insights and tools to assess the impact of using Lua in their applications.

1 Introduction

This work takes place in the context of component based systems. To construct an application, computational blocks are instantiated and interconnected with anonymous, data-flow based communication. Coordination refers to the process of managing and monitoring these functional computations such that the system behaves as intended. Keeping Coordination separate from Computations increases reusability of the latter blocks as these are not polluted with application specific knowledge. Examples of typical coordination tasks are switching between controllers upon receiving events, reconfiguring computations and dealing with erroneous conditions. A complex example of an robot applications constructed using this paradigm can be found here [1].

To implement coordination we propose to use the Lua [2] extension language. Using an interpreted language for this purpose has several advantages. Firstly, the robustness and hence safety, of the system is increased. This is because scripts, in contrast to C/C++, can not easily crash a process and thereby bring down unrelated computations that are executed in sibling threads. This property is essential for the aspect of coordination, which, as a system level concern, has higher robustness requirements than regular functional computations.

Secondly, the use of a scripting language facilitates less experienced programmers not familiar with C/C++ to construct components. This is important for the robotics domain, where users are often not computer scientists. Moreover, rapid prototyping is encouraged while leaving the option open to convert parts of the code to compiled languages after identification of bottlenecks. At last, the use of an interpreted language permits dynamic changes to a running system such as hot code updates. This is essential for building complex and long running systems that can not afford downtime.


The major challenge of using Lua in a hard real-time context is dealing with allocation and recuperation of memory. Previously we sketched two strategies to address this: either running in a zero-allocation mode and with the garbage collector deactivated or in a mode permitting allocations from a preallocated memory pool using an O(1) allocator and with active but controlled garbage collection [3]. In practice, especially when interacting with C/C++ code it may be inconvenient to entirely avoid collections, hence now we consider it necessary to run the garbage collector.

The rest of this paper is structured as follows. The next section gives an overview of related work. Section 3 describes how we address the issue of memory management in a garbage collected language used for coordination. Section 4 describes four experiments with the two goals of demonstrating the approach and giving an overview of the worst-case timing behavior to be expected. Robustness is discussed in the context of the last experiment, a coordination statechart. We conclude in section 5.

2 Related work

The Orocos RTT framework [4] provides a hard real-time safe scripting language and a simple state machine. While both are much appreciated by the user community, the limited expressivity of the state machine model (e.g. the lack of hierarchical states) and the comparably complex implementation of both scripting language and state machines have been recognized as shortcomings. This work is an effort to address this.

The real-time Java community has broadly addressed the topic of using Java in hard real-time applications [5]. The goal is to use Java as a replacement to C/C++ to build multi-threaded real-time systems. To limit the impact of garbage collection parallel and concurrent collection techniques are used [6]. For our use case of building domain specific coordination languages we chose to avoid this complexity as coordination can be defined without language level concurrency. In return this permits taking advantage of the deterministic behavior of a single threaded scripting language.

The Extensible Embeddable Language (EEL) [7] is a scripting language designed for use in real-time applications such as audio processing or control applications. Hence, it seems an interesting alternative to Lua. Lua was ultimately chosen because of its significantly larger user community.

3 Approach

To achieve deterministic allocations, Lua was configured to use the Two-Level Segregate Fit (TLSF) [8] O(1) memory allocator. This way memory allocations are served from a pre-allocated, fixed pool. Naturally, this raises the issue of how to determine the required pool size such that the interpreter will not run out of memory. We address this in two ways. Firstly, by examining memory management statistics the worst case memory consumption of a particular application can be determined and an appropriate size set. Due to the single threaded nature of Lua a simple coverage test can give high confidence that this value will not be exceeded in subsequent runs. Furthermore, to achieve robust behavior the current memory use is monitored online and appropriate actions are defined for the (unlikely) case of a memory shortage. What actions are appropriate depends on the respective application.

This leads to the second challenge for using Lua in a hard real-time context, namely garbage collection. In previous work [3] we suggested to avoid garbage collection entirely by excluding a set of operations that resulted in allocations. However, in practical applications that transfer data between the scripting language and C/C++ this is not always possible. Consequently the garbage collector can not be disabled for long periods and must be either automatically or manually invoked to prevent running out of memory. For achieving high determinism it is necessary to stop automatic collections and to explicitly invoke incremental collection steps when the respective application permits this. Only this way it can be avoided that an automatic collection takes place at an undesirable time.

The Lua garbage collector is incremental, meaning that it may execute the garbage collection cycle in smaller steps. This is a necessary prerequisite for achieving low garbage collection latencies, although of course no guarantee; ultimately the latency depends on various factors such as the amount of live data, the properties of the live data\(^1\) and the amount of memory to be freed. The control and coordination applications we have in mind generally tend to produce little garbage because the scripting language is primarily used to combine calls to C/C++ code in meaningful ways. Even though, to achieve high

\(^1\)In Lua, for instance, tables are collected atomically. Hence large tables will increase the worst-case duration of an incremental collection step.
robustness the worst-case duration of the collection steps can be monitored to deal robustly with possible timing violations.

The following summarizes the basic approach. First, the desired functionality is implemented and executed with a freely running garbage collector. This serves to determine the maximum memory use from which the necessary memory pool size can be inferred by adding a safety margin (e.g. the maximum use times 2). Next, the program is optimized to stop the garbage collector in critical paths and incremental steps are executed explicitly. The worst case timing of these steps is benchmarked, as is the overall memory consumption. The program is then executed again with the goal to confirm that the explicitly executed garbage collection is sufficient to not run low on memory.

### 4 Experiments

In this section we describe the experiments carried out to assess worst-case latencies and overhead of Lua compared to using C/C++ implementations. All tests are executed using Xenomai [9] (v2.5.6 on Linux-2.6.37) on a Dell Latitude E6410 with an Intel i7 quad core CPU and 8 GiB of RAM, with real-time priorities, current and future memory locked in RAM and under load.² Apart from the cyclictest all tests are implemented using the Orocos RTT [4] framework. The source code is available here [15].

#### 4.1 Lua Cyclictest

The first test is a Lua implementation of the well known cyclictest [10]. This test measures the latency between scheduled and real wake up time of a thread after a request to sleep using `clock_nanosleep(2)`. The test is repeated with different, absolute sleep times. For the Lua version, the test is run with three different garbage collector modes: **Free**, **Off** or **Controlled**. **Free** means the garbage collector is not stopped and hence automatically reclaims memory (the Lua default). **Off** means the allocator is stopped completely³ by calling `collectgarbage('stop')`. **Controlled** means that the collector is stopped and an incremental garbage collection step is executed after computing the wake up time statistics (this way the step does not add to the latency as long as the collection completes before the next wake up).

The purpose of this test is to compare the average and worst case latencies between the Lua and C version and to investigate the impact of the garbage collector in different modes.

#### Results

The following table summarizes the results of the cyclictest experiments. Each field contains two values, the average ("a") and worst case ("w") latency given in microseconds, that were obtained after fifteen minutes of execution.

<table>
<thead>
<tr>
<th>sleep time</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a, w</td>
<td>a, w</td>
<td>a, w</td>
<td>a, w</td>
<td>a, w</td>
</tr>
<tr>
<td>Lua/free</td>
<td>2,41</td>
<td>2,39</td>
<td>3,38</td>
<td>3,45</td>
<td>5,46</td>
</tr>
<tr>
<td>Lua/off</td>
<td>2,38</td>
<td>3,38</td>
<td>3,38</td>
<td>3,43</td>
<td>5,38</td>
</tr>
<tr>
<td>Lua/ctrl</td>
<td>2,38</td>
<td>2,42</td>
<td>3,37</td>
<td>3,36</td>
<td>5,46</td>
</tr>
</tbody>
</table>

Comparing the C cyclictest with the Lua variants as expected indicates that there is an overhead of using the scripting language. The difference between the three garbage collection modes are less visible. The table below shows the average of the worst case latencies in microseconds and expressed as a ratio to the average worst case of C. Note that the average of a worst-case latency is only meaningful for revealing the differences between the four tests, but not in absolute terms. A better approach might be to base the average on the 20% worst-case values.

<table>
<thead>
<tr>
<th>test</th>
<th>WC avg (us)</th>
<th>ratio to C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>35.2</td>
<td>1</td>
</tr>
<tr>
<td>Lua/free</td>
<td>42.0</td>
<td>1.19</td>
</tr>
<tr>
<td>Lua/off</td>
<td>39.2</td>
<td>1.11</td>
</tr>
<tr>
<td>Lua/ctrl</td>
<td>39.8</td>
<td>1.13</td>
</tr>
</tbody>
</table>

The above table shows that a freely running garbage collector will introduce additional overhead in critical paths. Running with the garbage collector off or triggered manually at points where it will not interfere add approximately 11% and 13% respectively compared to the C implementation. Of course the first option is only sustainable for finite periods. 13% of overhead does not seem much for using a scripting language, however it should be noted that this is largely the result of only traversing the boundary to C twice: first for returning from the sleep system call and secondly for requesting the current time.

---

²**ping -f localhost**, and while true; do ls -R /; done.
³This is possible because the allocations are so few that the system does not run out of memory within the duration of the test.
4.2 Event messages round trip

The second experiment measures the timing of time-stamped event messages sent from a requester to a responder component, as shown in Figure 1. The test simulates a simple yet common coordination scenario in which a Coordinator reacts to an incoming event by raising a response event, and serves to measure the overhead of calls into the Lua interpreter. The test is constructed using the Orocos RTT framework and is implemented using event driven ports connected by lock free connections. Both components are deployed in different threads. Three timestamps are recorded: the first before sending the message, the second at the responder side and the third on the requester side after receiving the response. The test is executed using two different responder components implemented in Lua and C++.

![Sequence diagram of event round trip test.](image)

For the Lua responder, this application takes advantage of the fact that the requester component will wait for 500us before sending the next message and executes an incremental garbage collection step after sending each response. If this assumption could not be made, the worst-case garbage collection delay would have to be added to the response time (as is the case for experiment 4.3).

Results The following table summarizes the average ("a") and worst-case ("w") duration of this experiment for the request \(t_2 - t_1\), response \(t_3 - t_2\) and total round trip time \(t_3 - t_1\); all values in microseconds.

<table>
<thead>
<tr>
<th>req</th>
<th>resp</th>
<th>total</th>
<th>Lua/C (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, w</td>
<td>a, w</td>
<td>a, w</td>
<td>a, w</td>
</tr>
<tr>
<td>C 9, 37</td>
<td>7, 18</td>
<td>16, 50</td>
<td>-</td>
</tr>
<tr>
<td>Lua 15, 47</td>
<td>11, 59</td>
<td>26, 106</td>
<td>1.63, 2.12</td>
</tr>
</tbody>
</table>

On average, the time for receiving a response from the Lua component is 1.6 times slower than using the C responder. The worst case is 2.2 times slower. Of the 1 MiB memory pool, a maximum of 34% was used. It is worth noting that for the initial version of this benchmark, the response times were approximately eight times slower. Profiling revealed that this was caused by inefficient access to the timestamp message; switching to a faster foreign function interface yielded the presented results.

4.3 Cartesian Position Tracker

The following two experiments illustrate more practical use cases. The first experiment compares both a Lua and C++ implementation of a so-called “Cartesian position tracker”, typical in robotics, and running at 1KHz, by measuring the duration of the controller update function. In contrast to the previous example the incremental garbage collection step is executed during the controller update and hence contributes to its worst case execution time.

The following listing shows the simplified code of the update function. Note that \texttt{diff} function is a call to the Kinematics and Dynamics Library (KDL) [11] C++ library, hence the controller is not implemented in pure Lua. This is perfectly acceptable, as the goal is not to replace compiled languages but to improve the simplicity and flexibility of using the primitives these offer.

```lua
pos_msr = rtt.Variable("KDL.Frame")
pos_dsr = rtt.Variable("KDL.Frame")
vel_out = rtt.Variable("KDL.Twist")
local vel, rot = vel_out.vel, vel_out.rot
function updateHook()
    if pos_msr:read(pos_msr) == 'NoData' or
       pos_dsr:read(pos_dsr) == 'NoData' then
        return
    end
    diff(pos_msr, pos_dsr, vel_out, 1)
    vel.X = vel.X * K[0]
    vel.Y = vel.Y * K[1]
    rot.X = rot.X * K[3]
    vel_out:write(vel_out)
    luagc.step()
end
```

Note that for Lua versions prior to 5.2 invoking the incremental garbage collector \texttt{(collectgarbage('step'))} restarts automatic collection, hence \texttt{collectgarbage('stop')} must be invoked immediately after the first statement. The custom \texttt{luagc.step} function executes both statements.
**Results** The following table summarizes the results of the worst case execution times in microseconds. The average execution time is approximately 14 times, the worst case duration 7 times slower than the C version. The worst case garbage collection time measured was 29us, of the 1MiB memory pool size a maximum of 34% was in use.

<table>
<thead>
<tr>
<th>type</th>
<th>duration (avg, max)</th>
<th>Lua/C (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>5, 19</td>
<td>-</td>
</tr>
<tr>
<td>Lua</td>
<td>68, 128</td>
<td>13.6, 6.7</td>
</tr>
</tbody>
</table>

In the current implementation the majority of both execution time spent and amount of garbage generated results from the multiplication of the K gains with the output velocity. If performance needed to be optimized, moving this operation to C++ would yield the largest improvement.

### 4.4 Coordination Statechart

The second real-world example is a coordination Statechart that is implemented using the Reduced Finite State Machine (rFSM) domain specific language [12], a lightweight Statechart execution engine implemented in pure Lua. The goal is to coordinate the operation of grasping an object in an uncertain position. The grasping consists of two stages: approaching the object in velocity control mode and switching to force control for the actual grasp operation when contact is made. This statechart is shown in Figure 2.

![Figure 2: Coordinating the grasping of an object.](image)

The real-time constraints of this example depend largely on the approach velocity: if the transition to the grasp state is taken too late, the object might have been knocked over. To avoid the overhead of garbage collection in this hot path, the collector is disabled when entering the approach state and enabled again in grasp after the respective controllers have been enabled.

Besides the actual grasping it is necessary to monitor the memory use to avoid running out of memory. With an appropriately sized memory pool and sufficient garbage collection steps, such a shortage should not occur. Nevertheless, to guarantee robust and safe behavior this condition must be taken into account and the robot put into a safe state. This is shown in Figure 3.

![Figure 3: Dealing with low memory.](image)

As the grasping task can only take place while enough memory is available, it is defined as a substate of operational. The structural priority rule of the Statechart model [13] then guarantees that the transition to mem_low has always higher priority than any transitions in the grasping state machine.

Identifying the required memory pool size has currently to be done by measuring empirically the maximum required memory of a state machine and adding a safety margin. To avoid this, it would be desirable to infer the expected memory use from the state machine description. Predicting the static memory used by the state machine graph is straightforward; also the run-time memory use of the rFSM core is predictable as it depends on few factors such as the longest possible transition and the maximum number of events to be expected within a time step. However, predicting the memory use of the user supplied programs would require a more detailed analysis/simulation, which is currently out of the scope of this work; but in robotics, most user supplied programs are in C/C++ anyway.

### Results

The previously described grasping coordination Statecharts are tested by raising the events that effect the transitions from grasping, approach to grasp. The timing is measured from receiving the e_contact event until completing the entry of the grasp state. After this, the same sequence of events is repeated. The functions for enabling the controller

---

4It consists mainly of traversing and transforming the FSM graph.
are left empty, hence the pure overhead of the FSM execution is measured. Running the test repeatedly for five minutes indicates a worst-case transition duration between approach and grasp of 180us. The memory pool size was set to 1 MiB and the TLSF statistics report a maximum use of 58%. To test the handling of low memory conditions, in a second experiment the collector is not started in the grasp state. As a result no memory is recovered, eventually leading to a low memory condition and a transition to the mem_low state. For this test the worst case maximum memory use was as expected 70%.

This test does not take into account the latencies of transporting an event to the state machine. For example, when using the Orocos RTT event driven ports, the experiments from Section 4.2 can complement this one. Moreover it should be noted that so far no efforts have been put into minimizing rFSM transitions latencies; we expect some improvement by optimizing these in future work.

Robustness considerations As described, basic robustness of coordination state machines is achieved by monitoring of memory and current real-time latencies. However, the system level concern of coordination unfortunately combines the two characteristics of (i) requiring higher robustness than functional computations and (ii) being subject to frequent late modifications during system integration, the latter of course being susceptible to introduce new errors. The combination of scripting language and rFSM model can mitigate this effect in two ways. Firstly the scripting language inherently prevents fatal errors caused by memory corruption, thereby making it impossible to crash the application. Secondly, rFSM statecharts execute Lua user code in safe mode\textsuperscript{5}. This way errors are caught and converted to events that again can be used to stop the robot in a safe way.

5 Using the Lua pcall function

5 Conclusions

We have described how the Lua programming language can be used for hard real-time coordination and control by making use of an O(1) memory allocator, experimentally determining worst-case memory use and manually optimizing garbage collection to not interfere in critical paths. Several experiments are carried out to determine worst-case latencies.

As usual, benchmark results should be judged with caution and mainly serve to remind that appropriate validation should be repeated for each critical use. In particular when real-time allocation and collection is involved, run time validation of real-time constraints must be considered as an integral part of the application.

The major shortcoming of the current approach is that worst-case memory use can be difficult to predict. To deal with this we currently allocate additional safety margins. As the overall memory usage of the Lua language is comparably small, such a measure will be acceptable for many systems, save the very resource constrained.

To conclude, we believe the results demonstrate the feasibility of our approach to use a scripting language for hard real-time control and coordination that permits to significantly improve robustness and safety of a system. The price of these improvements are (i) increased yet bounded worst-case latencies, (ii) computational overhead, as well as (iii) requiring additional precautions such as manual scheduling of garbage collection. In summary, we believe this constitutes a modern and practical approach to building hard real-time systems that shifts the focus from lowest possible latency to sufficient latency while maximizing reliability.

Future work will take place in two directions. On the high level we are investigating how to automatically generate executable domain specific languages from formal descriptions. Implementationwise we intend to investigate if and how the presented worst case timing behavior can be improved by using the luajit [14] implementation, a high performance just-in-time compiler for Lua.

Acknowledgments This research was funded by the European Community under grant agreements FP7-ICT-231940 (Best Practice in Robotics), and FP7-ICT-230902(ROSETTA), and by K.U.Leuven’s Concerted Research Action Global real-time optimal control of autonomous robots and mechatronic systems. The authors also gratefully acknowledge the support by Willow Garage, in the context of the PR2 Beta program.

References


Platform independent remote control and data exchange with real-time targets on the example of a rheometer system

Martin Leonhartsberger
Institute for Measurement Technology, Johannes Kepler University
Altenbergerstrasse 69, 4040 Linz, Austria
martin@4leo.at

Bernhard G. Zagar
Institute for Measurement Technology, Johannes Kepler University
Altenbergerstrasse 69, 4040 Linz, Austria
bernhard.zagar@jku.at

Abstract

Most real-time applications in context to automatic control and data acquisition do require user interfaces to change parameters or to extract data at runtime. Especially in prototyping control processes, it is common to use CACSD programs such as Matlab or Scicos for code generation. While still being in the prototyping stage the customer demands to start evaluation using a software tool. Very often those customers do not possess the ability to work with CACSD tools or other tools as xRTAILab because of missing licenses or lack of knowledge in linux. To close the gap between the real-time target and data evaluation a remote control framework is introduced. Our approach is to use RTAIXML as a XML-RPC server. It is running on the rt-system and has the ability to connect to the CACSD generated code. This server instance can be contacted through a platform independent Java framework which allows quick prototyping of simple to use user interfaces without interfering the control system development. It is possible to transfer scopes and to change parameters on running targets. A successful application is shown on the example of a low cost rheometer.

1 Introduction

Connecting a well adapted end user software to an experimental real-time application can be quite hard. Usually such environments are served through software packages as Matlab [1] and Scicos [2]. A technician is acquiring data and evaluating it with those tools directly. On the other hand, professional industrial appliances use products as for example LabVIEW [3] and its data acquisition hardware to get everything from a single source. A handy user interface is built and the application is on its way. The second approach is to develop all the needed software in house for a series production. Both is very expensive and usually not applicable for projects like the one described in this paper. Often it should be low cost, therefore many professional tool chains can’t be used. It should be possible to produce it in small series and especially for the rheometer it is a requirement to make usage easy. Still it is a prototype with many changes ongoing, so rapid prototyping should be possible. Therefore we suggest a combination of Matlab or Scicos for Computer Aided Control Design (CACSD), an open source real-time target for execution, a webservice provider on the target for data exchange and a java framework to setup an application for remote controlling the whole application. A similar unified approach is described from Roberto Bucher and Silvano Balemi in [4]. With this suggested approach an end user software, independent from most operating systems (it uses Java) can be written. With the framework provided, software development concentrates on building a graphical user interface (GUI) and a state machine changing parameters of the running real-time
target through the framework. With this combination rapid prototyping of the control process is possible parallel to the GUI implementation as long as the state machine interface parameters are designed properly.

2 Used tools

An RTAI [5] patched 2.6.32.11 kernel is used on our reference system. RTAI was started by Paolo Mategazza from the Politecnico di Milano and is introducing a real-time schedule into the kernel which is then able to execute hard real-time code, both in user and kernel space. Precompiled kernels for a quick start can be found for example in [6]. For communication with the data acquisition hardware a COMEDI [7] driver is being used. COMEDI drivers are low level kernel drivers which provide their functionalities through a common interface for different data acquisition hardware. Comedilib as a user space library is used to extend the block modeling capabilities of Simulink [1] and Scicos [2] for code generation in a later step. All tools except Simulink are available as open source.

2.1 RTAI-Lab and RTAI-XML

RTAI-Lab makes a connection between RTAI and CACSD tools with the aim to allow code generation for RTAI out of CACSD tools. To monitor and change parameters and scopes GUI applications are existing. RTAI-XML [8] is a branch of those applications which aims to bring the lab environment x/qrtailab [9] to a remote space. This tool communicates over an XML remote procedure call protocol (XML-RPC) with the target and exchanges data and parameter settings. The protocol is mostly documented in the sources and was implemented into a Java framework during this work. Overview to the available XML-RPC commands:

**Connect:** will initiate a connection to the RT target and negotiate ports.

**Start:** will start the target connected with if it is not already running. Targets could also start automatically on connect.

**Stop:** will stop the target connected to.

**Get Parameters:** will download the target parameters.

**Send Modified Parameters:** requires a (changed) getParameters structure, upload them and set new target parameter values. This is especially needed to change feedback gains or the state.

**Get Signal Structure:** receives a structure of exposed signals on the target. They can be scalar or arrays.

**View Selected Signal:** This will start a continuous transfer of a signal, so not over XML-RPC but with an explicit socket connection.

**Write Signal Data on File:** This will store a signal on the server to request it in a later stage. It is not used in this project, but this may be very useful for unattended recordings.

**Disconnect:** This command will shutdown the session and deallocate the target lock.

2.2 Java Framework and state machine

For remote controlling a target over RTAI-XML a Java Framework was developed during this work.

```
RTAIProgramming
+connect()
+disconnect()
+stop()
+setParameter()
+setParameterByName()
+getBlockByName()
+setParameterList()
+getScopeByName()
+addListener()
+removeListener()
+Hello()
+disconnect()
```

**FIGURE 1:** Java Framework

It enables one to change parameters and export scope data from the target in a programming context, not in real-time of course, but fast enough for visualization. The framework is intended to work with different target generators, for example Simulink and
Scicos. The java classes, designed with the Model View Controller pattern are shortly presented in the UML chart in Fig 1.

Class RTAITarget: This class provides an object for a target. `setParameterByName(..)` allows the change of a single parameter or the whole parameter-set can be updated with `setParameters(....)` at once. `getScopeByName(String sName)` returns an object of type RTAIScope which is being described in the next item. Functions for sending and receiving parameters allow to update the target at certain times when all parameters have been set or can be triggered with the function parameter `updateImmediately` which is available for all "set" functions. Listeners will notify attached objects about changes in the parameters.

Class RTAIScope: This class provides an object of a single scope instance. All the scope history is saved in this object after triggering the socket transfer of the data with `scope.start()`. Listeners will as well update attached objects about new data.

Interface RTAITargetListener: Provides the interface which target listener objects will have to implement.

Interface RTAIScopeListener: Provides the interface which scope listener objects will have to implement.

Interface XMLRPCClientInterface: Provides an interface with all remote procedure calls specified by RTAI-XML. As it is possible to use different XML-RPC implementations (in the corresponding implementation the apache classes have been used) this is done generic to change the implementation if necessary or desired.

Class XMLRPCClientImpl: The actual implementation to the XMLRPCClientInterface.

A possibility to change parameters is now found, the next step is to implement a work flow by using states for example. This could be done by state flow objects in Simulink or by additional C programs directly on the target for example. Still, a client program for evaluating the results would be necessary as well and very often the required workflows are due to change quite often, therefore we suggest to implement a state machine on the client in a program. Every automatic control block and calculation can be switched on and off with a constant in the CACSD model. Initially all processes are switched off. The remote program then starts to switch on or off the parts as it is defined in a state model. We have successfully used this approach to implement different workflows for one device.

To show the usage of the framework, a very small demonstration program has been written. The comments in the code will explain the different steps. The source of the framework itself is not printed due to its large amount of line numbers but will shortly be available as a sourceforge project under the GPL.

2.2.1 A short example

We create a small example which is shown in Fig. 2. The demo target has been inherited of the comedilib demos. It demonstrates the output of a signal to an RTAI scope and to a COMEDI device. After code generation and setup of all necessary components on the real-time target it is now possible to change parameters with a few lines of Java code on a platform-independent remote machine.

```
public class AppMinimal {

    public static void main(String[] args) {
        // create connection to RTAI-XML server
        ClientXmlInterface remoteClient = new ClientXmlRpcRtaiImpl(new String("10.0.0.10"), 29500, new String("STEST"));

        // create a target
        RTAITarget target = new RTAITarget(remoteClient);

        try {
            // Connect to target and start it
            target.connect();
            target.start();
        }

        // close connection to RTAI-XML server
        remoteClient.close();
    }
}
```

**FIGURE 2:** Simple Simulink example
3 Application

Rheologic measurement cycles do take a certain amount of time and the number of rheometer-devices is limited, therefore it is often not possible to carry out a large test series. It is intended to raise the amount of control samples in quality assurance and laboratory environments with new automated low cost measurement devices. Extensive research in implementation and verification was carried out with a feasibility study [10] and three master theses ([11], [12], [13]). We propose a low cost principle which should be accomplished by a new measurement concept, usage of low cost parts and open source products in control system design. A fully operational prototype based on a disc–disc CSR principle was built and is now being evaluated. Optimized for Bitumen (remains of distillation of crude oil with a colloidal structure) which changes viscosity typically in a range of $10^7$ it can nevertheless be used for many other materials.

The rheometer is intellectual property of vialit GmbH in Austria.

3.1 Mechanical design

FIGURE 3: Mechanical setup — top view
The rheometer consists of a rotatory and a static shaft. The rotor is carried by a small metal ball bearing (which is kept centered by a small magnet in the stator). Stabilization occurs through a magnet on the top which is also lifting the rotor to keep friction down. The distance needs to be trimmed to a point, where the magnetic force is stabilizing the rotor but does not actually lift it completely. Only a reduction of the weight on the ball is feasible and gives a positive side effect to reduce friction forces. On top of the rotor a tripod is mounted. Each foot of this tripod has a magnet glued into epoxide resin. Each of these magnets is centered between two coils in a Helmholtz type configuration. The constant magnetic field between those coils causes the magnets to move, the combination is an electrodynamic actor. AC-currents are causing the rotator to oscillate. Both parts, rotor and stator are heated to a selectable temperature.

### 3.2 Electronic design

The following actuators are used:

**Peltier elements:** Two peltier elements (Fig. 4 - 12) mounted on the stator clamp are heating and respectively cooling the stator and the probe.

**Heating filament:** Due to the poor heat transfer stator tempering is not sufficient. A heating filament (Fig. 4 - 7) is inserted into the rotor head to enable the possibility of exact temperature control in the probe. An extensive research on the heat propagation model can be found in [12].

Both, sensors and actuators need amplifying circuits which have been designed in [11] and [12]. Connected to the amplifying stage a Sensoray S526 data acquisition card is used. The card driver is embedded through COMEDI [7]. Connected through the PC/104 interface of the EPIC/CE embedded PC the data acquisition card communicates with the embedded PC.

### 3.3 Control software

Simulink generated targets are providing a good possibility to run control processes in a real-time environment. Though it was not made to implement...
complex structures in form of workflows. Very often and especially in the rheometer case program states and workflows are needed. After a certain measurement point has been recorded and evaluated there is a need to go on to the next one. When moving on parameters have to be changed according to the workflow and its requirements. Using the framework from Section 2.2 a state flow and a GUI (Fig. 6) for an industrial customer was developed.

FIGURE 7: Rheometer state flow (main)

The state machine (running on the client with the GUI) is switching on and off parts of the automatic controllers (running on the embedded target) depending on the actual requirements for the measurement steps. Additionally it is possible to change some of the parameters if necessary. All additional parameter can still be changed by an control engineer using q/x/jRTAI-Lab from the RTAI project [5] parallel to the running GUI.

FIGURE 8: Rheometer state flow (automatic)

4 Conclusions

We have shown tools and methods to fast prototype control applications using a real-time linux system. With this toolsets it is possible to program a platform independent GUI for a control application without loosing flexibility in automatic control development.
5 Acknowledgment

The author gratefully acknowledges the partial financial support for the work presented in this paper by the Austrian Research Promotion Agency and the Austrian COMET program supporting the Austrian Center of Competence in Mechatronics (ACCM) and the company Vialit GmbH, Austria.

References

[10] Low-Cost Rheometer—Konzeptstudie, Bernhard Zagar, 2005, INSTITUTE FOR MEASUREMENT TECHNOLOGY, JKU, LINZ, AUSTRIA
[13] Low Cost Rheometer for dynamic viscosity measurement of bitumen with real-time Linux, Martin Leonhartsberger, 2011, INSTITUTE FOR MEASUREMENT TECHNOLOGY, JKU, LINZ, AUSTRIA
Platform independent remote control and data exchange with real-time targets
Abstract

We have developed a system based on GNU/Linux, Apache, PostgreSQL, Zend Framework, the REACT control engine, and various free software projects to create a system for remote data collection, analysis, and control of grain silos. This system allows grain operators to monitor/manage grain silos from a distance. The system includes a GNU Linux based computer, locally on site at the silo, with the REACT control engine installed, and, a GNU/Linux server in the cloud running the Zend Framework and the PostgreSQL database. Communication with the server is via GET/POST.

1 Introduction

For grain operators to be able to monitor/manage grain silos from a distance, they must have access to grain quality evaluations done locally on-site at the silo, history of outside temperature, relative humidity, and history of temperature measurements inside the silo. In addition, operators must be able to set the grain ventilation parameters remotely so that ventilation can then happen automatically on-site, with no need to have a person locally checking temperature and relative humidity and deciding when the fans should be turned on and off. All of this can be automated except that the grain quality evaluations must be done by an actual human being using laboratory equipment. We can however provide web forms so that the data from these evaluations get into the database on the website the same day that they are carried out. This design also allows the management team to be distributed at any location around the world where there is Internet service. It of course also allows all interested parties such as those that own the grain, or loaned the money for the grain, to verify that the grain is still in the silo and in good condition.
The temperature columns are typically located such that there is a maximum of 5 meters to the next column, or 2.5 meters from the wall to the nearest sensor, such that there will be a temperature measurement no farther than 2.5 meters from every point in the silo. Vertically the temperature sensors have been located every 1.5 meters, but, will switch to every 0.5 meters to be able to more accurately estimate the height of the grain. More on how we estimate the height of the grain later. The number of temperatures measured inside a silo can run into the hundreds, depending on the size of the silo. Every 10 minutes, the outside temperature, relative humidity, all of the temperatures in the columns inside the silo, and the status of the ventilation fans, are logged locally for backup, and, also communicated to the remote web server. At the same time, we check for new control parameters, and download that at the same time, if necessary.

A graph of temperature and relative humidity taken from near Poncitlan, Jalisco, for one of the initial tests, is shown in Figure 1.

3 How the monitoring is done

On site, we use one hardened GNU/Linux computer running our REACT control engine [1],[2],[3] to read the values from all of the data acquisition devices. We are currently using the Technologic Systems TS-7300 single board computer, but, we are also testing the TS-7500 and TS-7553 and plan to switch to one of these for future projects, as they are cheaper and more compact. None of these have moving parts, and will withstand temperature up to 70 degrees Celsius. They all have SD card slots for gigabytes of local storage. All of the data acquisition devices that we use support the MODBUS protocol, and we communicate via RS-232, RS-485, and 900 MHz radios. The GNU/Linux computer is connected to the Internet via Cellular modem, or DSL modem, or the local network, depending on availability.

The temperature and relative humidity are read using a small PCB with micro-controller and interface for a GE Chipcap-D temperature/relative humidity sensor. The GE Chipcap-D is soldered to a small PCB that must be located close to the micro-controller, and protected from the weather. Discussing the optimum location of this system is beyond the scope of this paper.

The temperatures inside of a silo are read using Dallas 1-Wire temperature sensors, and connected via Category 5, UTP cable. There is a microcontroller that can communicate with up to 16 Dallas 1-Wire cable runs, via a 16:1 multiplexor. Each cable run can be up to 50 meters long, and is for one column of temperatures inside the silo. Each column must be hung from the ceiling of the silo and bound together with a 1/4 inch steel cable using shrink-fit tubing. The steel cable is necessary due to the extreme forces that can be generated when the body of grain moves as the silo is being unloaded. The Dallas 1-Wire temperature sensors must be soldered every 0.5 meters or 1.5 meters depending on the accuracy of volume measurements desired.

Finally, to turn the ventilator fans on/off, we need a relay board with two relays, one that is normally open and goes in parallel with the ON button, and one that is normally closed, and goes in series with the OFF button. The ON relay is pulsed to turn the ventilator fan on, and, the OFF relay is pulsed to turn the ventilator fan off. This allows both automatic and manual control of the ventilator fans.

The architecture for an installation using 900 MHz radios is shown below. The SBC7300 is the GNU/Linux hardened computer, the CL4490 is a 900 MHz radio, the X505 is a master controller for ventilation fans that can connect multiple relay boards, the X105 is a slave device in case that more ventilation fans need to be connected, and the W100 is a weather station. The T200 is the PCB for reading up to 16 temperature columns, and can be daisy chained with an RS-485 port.

4 The Server

The server where all of the data is logged can be located anywhere in the world where there is an Internet connection. The actual servers we use for this project are located in Karlsruhe, Germany, and run Debian GNU/Linux, with the Zend Framework, and PostgreSQL database installed. The monitored data at each silo is uploaded every 10
minutes via HTTP/POST, and the ventilation parameters are checked for changes every 10 minutes via HTTP/GET. If we go for more than 20 minutes without communications from a particular site, we mark it as offline. We use only HTTPS for these requests, and, each request must be accompanied by the correct security key for each silo, or the request is rejected. Notice that the ventilation parameters for each silo can be set from anywhere in the world with Internet access, and they will be transferred and applied at the local site within 10 minutes. Thus, the team managing the silos must only log into the server for all management functions of the various silos. Onsite at each silo, we will not require grain conservation experts, only technical people to maintain the equipment. There must be experts on grain certification/quality to visit all of the silos periodically and carry out quality analysis and certify the quality of the grain using portable laboratory equipment.

5 Control algorithms and analysis of the data

The basic algorithm for controlling the ventilation fans is to turn the ventilation fans on when the relative humidity and temperature are in the correct range, and, we are not in the hours when electricity is most expensive, if specified. Obviously, the fans are then turned off when the conditions are not met. Further, when managers determine that the grain is adequately ventilated, and ventilation is no longer necessary, they will be able to disable ventilation entirely via the web page for the silo in question. To start, the managers must put in the range of humidities/temperatures to ventilate the grain. For corn for instance, if you ventilate the grain in the range of 70-75% relative humidity, you will end up with grain that is 13.5 - 14% humidity by weight. We do plan to automate the process further, by letting the manager only put the target humidity of the grain, and the algorithm will then automatically try to hit that target. We should note, that the parameter of the grain that matters more than all others is the humidity. If it is too humid, then there will be mold and other things that will destroy the grain. If it is too dry, the grain will crack and break. After humidity, we then want the grain as cool as possible. If the grain is just above zero degrees Celsius, it can be stored for years. If it is over 30 degrees Celsius, it will only last a matter of months. In hotter regions like the state of Sinaloa, Mexico, when the humidity is in the correct range, the temperature is typically much hotter than we would like, and, thus, the grain stored in Sinaloa must be used in a relatively short period of time.

To estimate how much grain is in a silo, we basically need to know the level of the grain in the silo. From this, using simple geometry, we can calculate how many cubic meters of grain are in each silo. We can then estimate the number of metric tons of grain in the silo using the approximate number of metric tons of grain per cubic meter. So, the question is now can we estimate the level of grain in the silos? The answer is in the temperature readings. In the silo, the temperature above the grain makes wide swings based on the outside temperature, sun hitting the silo, cloud cover, weather conditions in general. Grain, however, is a natural insulator, and, temperatures inside the grain remain very constant.

FIGURE 3: Overall System Architecture

FIGURE 4: Data from one Temperature Column in a Silo
6 Conclusions

We have developed a novel system using GNU/Linux and other free software projects, that allows silos to be managed/monitored from a distance, so that there is no need for grain conservation experts at each site, and, so that all stakeholders have access to the grain quality and also the quantity of grain in each silo. The system allows managers to set the parameters of humidity and temperature when the grain will be ventilated, and then monitor the progress/effectiveness of the ventilation via the quality analysis done periodically on-site, and, disable ventilation when goals have been reached. Because of GNU/Linux and the many other free software projects, we were able to complete this project much quicker and at a lower cost.

References


A Status Report on REACT, a Control Engine that runs on top of GNU/Linux for Creating SCADA and DCS Systems

Don W. Carr, Rubén Ruelas, Benjamin Ojeda Magaña, Adriana Corona Nakamura
Universidad de Guadalajara
José Guadalupe Zuno 48, Col. Los Belenes, C.P. 45101, Zapopan, Jalisco, México
doncarr@gmail.com

Abstract

REACT is a control engine that runs on GNU/Linux, written in C/C++, for creating SCADA and DCS systems, or just simple controllers for a single machine or laboratory equipment. The design was based on the experience of the author on large SCADA systems for natural gas pipelines, natural gas distribution systems, and water distribution systems, software for testing military, industrial, and light vehicle transmissions, and research on control systems in general. REACT was designed as a general purpose control engine to scale similar to how GNU/Linux scales: from tiny embedded devices in the field, all the way up to large computers. Thus, REACT can run as the central software on a server as a SCADA system, and can also run on small hardened devices as the software for an RTU, or DCS controller.

1 Introduction

The development of REACT [1],[2],[3], was started in 2002 for Masters students at the University of Guadalajara, so that students would be able to actually see the software used for real-time systems and take the mystery away. It was first used for a project of two electronics masters students to automate a liquid chromatography instrument, with robotic system for changing samples, for the molecular biology department at the University of Guadalajara. REACT allows the configuration of all the common objects often referred to as point types or tag names that are common for a SCADA or DCS systems: analog input, analog output, discrete input, discrete output, timed discrete output, pulse count input, PID controller, etc. A variety of other object types have been created for data logging, control, and monitoring, etc, for special applications such as laboratory instruments, air conditioning monitoring/control, pump station monitoring/control, silo monitoring/control, etc. There is also a scripting language with the scripts tokenized for faster execution. Scripts can call member functions of all the REACT objects, which is supported through a code generator that automatically generates the code to link the script function to the object.

2 REACT Driver Model

When we started out, the code to communicate with the data acquisition card was compiled directly into REACT, and communicated with the factory driver that was a shared object file (.so). The problem is that it created a dependency on a particular shared object file, and the code had to be commented out for other systems that did not use this code. So, we quickly moved to putting the driver interface code into shared objects that can be loaded on demand using the system API dlopen(), dlvsym(), dlsym(), dlerror(), dlclose(). To support objects, we use this API to load an object factory function that is called to instantiate an object that inherits from our abstract base class iodriver which defines the interface to all I/O drivers. The factory function for each driver must return a pointer to iodriver which we currently have drivers for Modbus RTU, Modbus ASCII, Modbus...
TCP/IP, various PC data acquisition cards, Dallas 1-Wire via the OWFS project, a simple ASCII protocol that we developed, and, some simulators that we developed that load as drivers. This driver model allows us to write a simple simulator that loads as a driver to use during testing, and then switch easily to the real driver in the field. It allows us to keep the code compact for memory constrained embedded systems.

3 REACT Object model

We needed an object model to implement all of the control engine objects, often referred to as point types or tagnames, or just tags. Actually, we identify all control objects by their tagname which serves as a unique ID. As you will see, there is a need to have a unique ID to refer to objects from scrips, displays, etc. We have four basic types of objects: 1) Input objects that receive process values via a device driver, 2) Output objects that send/write process values via a device driver, 3) Control objects that access process values via input objects, and send process values via an output object, and, 4) Objects that only calculate values, do data logging, do user interfaces, etc. Currently, all of the object types must be hard-coded into REACT, but, we are in the process of switching them to be all loaded on demand at load-time, if they are needed for a particular project. We have started by implementing dynamic loading for one object type (analog input).

4 REACT Configuration

Like many projects, we started out storing all configuration in a directory full of delimited text files. However, there are possible corruption problems, and it is tedious to copy all of the files to the target system, and retrieve all of the configuration files, after local changes have been made, for backup. For this reason, we are moving to putting all of the configurations into SQLite, to eliminate the problems of corruption by using transactions, and, thus simplify copying the configurations to a target system, and backing up, since SQLite stores the complete database with all tables in a single file. SQLite is also extremely compact and can itself be loaded when needed, and then unloaded using dlopen(), dlsetx(), dlerror(), dlclose().

Faced with the need to write all of the functions to read/write the configuration for ALL object types, we quickly realized that it was repetitive, and could be done using code generation from a configuration file that named the database field names/types, and the corresponding field names in the objects. We also quickly realized that this same configuration file could be used to convert existing projects with delimited text files, to use database files. Finally, the existing user interface was a text editor to edit the delimited text files, and we would need a new user interface to edit the object configurations/configure REACT. With these same configuration files, if we add a few prompts, and a few other simple things, we can generate either a web based user interface for editing the configuration, or, a text interface, based on curses, to edit remotely via ssh. We have implemented the code generator for converting existing projects, and to read/write the configurations from REACT, and are working on generating a web interface so that we can offer an online configuration editor. For now, we are using the SQLite console application to remotely edit configuration parameters via ssh.

See below, the configuration file for discrete outputs in Figure 1, and then, the user interface generated from this file, in Figure 2.

![FIGURE 1: Config File used to Auto-Generate code for Discrete Outputs](image-url)
Early on, we realized the need for simple scripts so that common users, that may be control experts, but, are not programmers could extend the functionality of REACT by creating, for example, test sequences, control sequences, interlocks, alarm sequences, shut-down sequences, etc. Thus, we developed a parser to tokenize scripts for faster run-time execution, and a code generator to bind scripts to actual C++ object method calls. The code generator works by processing the object header file with a special keyword (SCRIPT_OBJECT) added (in comments) above objects that support script functions, and then another special keyword (SCRIPT_FUNCTION) above each method that can be executed from a script. Using the tagname, at load-time, we can identify the object, verify it has the method with the given name and parameters, and bind to that object/method. Thus, at run-time, when the script reaches the line with this method call, the given method is called, on the given object, with the given parameters.

We also needed to create system functions for things like sleep, wait for condition to be true, wait for user input, write text to output, etc.

These scripts have proven to be very useful and have been used on many different projects. Below is an example of a script to control the level in a tank between a low point and a high point. We basically open the valve when we reach the high point, and close the valve when we reach the low point. The tag hi_level is a discrete input connected to a level sensor that is on/true when the water is at or above the high point. The tag lo_level is a discrete input connected to a level sensor that is on/true when the water is at or above the low point. The tag valve_2 is a discrete output that activates a drain valve that is opened by sending the value true, and closed by sending the value false. The 15 second wait was introduced to avoid the valve opening and closing in rapid succession when a wave was generated.

As an alternative to scripts, we are also working on state diagrams [4],[5] to describe control algorithms/sequences. For state diagrams, we will still use the script commands described here to specify actions that take place on arriving at a state, or leaving a state.

6 Conclusion

We have created REACT, a very useful free software application that runs on GNU/Linux for creating SCADA systems, DCS systems, simple machine/instrument controllers, etc. REACT is extensible through creating new objects written in C/C++, and also, for non-programmers by writing REACT scripts. We make extensive use of code generation for repetitive programming tasks to save time and eliminate errors, and enable new functionality to be added faster. We use dynamically loaded shared objects for drivers and REACT objects, to eliminate dependency conflicts, reduce code size, and allow new drivers/object types to be added without re-compiling REACT.
References


Development of an optical profile measurement system under RTAI Linux using a CD pickup head

Gerstorfer, Gregor
Institute for Measurement Technology, Johannes Kepler University Linz, Austria
Altenbergerstr. 69, 4040 Linz
gregor.gerstorfer@jku.at

Zagar, Bernhard G.
Institute for Measurement Technology, Johannes Kepler University Linz, Austria
Altenbergerstr. 69, 4040 Linz
bernhard.zagar@jku.at

Abstract
The development of a low cost profile measurement system suggests – because of the feature low cost – the use of an open source control system. In this work, the authors combine a commercially available compact disc (CD) pickup head, the National Instruments NI-PCI 6221 data acquisition card, and a PC running RTAI Linux. The setup is used to scan along a CD and measure the distance between tracks. The setup will be used in a lab course by students to get to know rapid control prototyping systems and open source alternatives to expensive industrial systems.

1 Introduction
The words low cost are nowadays omnipresent, so this paper will fulfill these words and explains the use of low cost components in combination with RTAI Linux. The centrepiece of the setup is a commercially available optical pickup head of a standard CD drive. Those pickup heads are very inexpensive but still show a high level of accuracy. This is the reason why the authors use them to scan across a specimen’s surface for measuring its profile.

Several different applications using an optical pickup head as a measurement system can be found in [1], [2] or [3]. In this paper there is a focus on the pickup head itself but also on the data acquisition and control system which is done with RTAI Linux and RTAI–Lab. There are more than just several applications which already run RTAI Linux (can partially be found on the project’s homepage [4]). Here the application is also aiming to motivate students to work with open source software.

Preliminary to the use of a CD pickup head as a sensor system, several investigations have to be carried out. These investigations ought to deliver information about the electrical and mechanical structures and characteristics of those high quality but still low cost pickup heads. Main focus is given to the so called focus error signal, conveying the sought after measurement information of a change in distance to the reflective surface. This will be presented in the first part of this paper. The second part introduces the software and hardware parts of the system. At the end, some measurement results and how the system could be used in a students lab will be presented.

2 The optical system of the CD pickup head
Optical storage systems usually consist of a rotating data storage medium (i.e. a CD) carrying the data which is read out by the mechnano-optical pickup head. The included electronics control the system...
and convert the data. In this section the used pickup head (Mitsumi PXR–550X) from a commercially available CD–drive is investigated.

![Diagram of CD pickup head](image)

**FIGURE 1:** The working principle of a CD pickup head – focused case.

Figure 1 shows the principle of a CD pickup head, which consists of a laser diode operating at a wavelength of 780 nm, a linearly polarizing beam splitter, a collimator, an objective lens and an astigmatic lens in front of the detector array. The emitted laser beam is polarized by the polarizing beam splitter, collimated and reflected at the specimen’s surface after passing the objective lens. This objective lens can be moved by the so called voice coil motors (VCM). The scattered back light takes its way back through the objective lens, passes the beam splitter and is directed via the astigmatic lens to the detector array. The detector array is a four–quadrant photo diode, in Fig. 2 their arrangement can be seen. Because of the characteristics of the astigmatic lens, the beam shape at the detector array is either elliptic or circular, depending on whether the reflection of the beam takes place in the focal plane of the objective lens (circular beam) or if the reflection takes place out of the focal plane (elliptic shape). The orientation (NE–SW or NW–SE) and size of the elliptic beam pattern depends on the location and distance from the focal plane.

![Graphs of FE and RF signals](image)

**FIGURE 2:** Bottom: The FE signal’s characteristic s–curve with corresponding beam shapes at the detector array. Top: The RF signal.

The mentioned four elements of the detector array are named A, B, C and D. Each of them delivers a photocurrent depending on the illuminance in the respective area. These currents are subsequently converted into voltages $A$, $B$, $C$ and $D$. From these voltages, the so–called focus error (FE) signal can be derived:

$$FE = (A + C) - (B + D).$$

(1)

When a specimen with a reflective surface is moved towards the pickup head, the measured FE signal shows a characteristic s–shape with an approximately linear range over an input range of $\pm 3\mu m$, see Fig. 2. The reflection takes place exactly in the focal plane, when the FE signal becomes zero.

Beside the FE signal also the total illumination integrated over all four photodiodes, the RF signal, is of interest:

$$RF = A + B + C + D.$$  

(2)
The RF signal corresponding to the above mentioned FE signal is shown in the upper part of Fig. 2.

3 Profile measurement with a CD pickup head

The fact that the FE signal shows an approximately linear range in its s–curve is used for profile measurement: a specimen is placed exactly in the focal plane and is then moved in any direction within the focal plane to obtain a scan. The variation of the FE signal is then the measure for the profile. Since some surfaces show better reflective properties than others do, and the FE signal’s amplitude depends on the amount of scattered back light, a way to obtain a proportionality factor between FE signal and profile roughness is sought after. Therefore one has to assume, that the reflective properties of a specimen are approximately constant in the investigated area. Hence the voice coil motors are used to ascertain the proportionality between FE signal amplitude and specimen roughness.

For having the voice coil motors as a reference, their characteristics were investigated first (see also [5]). In Fig. 3 the results of the investigation using a laser vibrometer (Polytec OFV–505) measuring the VCM–displacement when different voltages are applied to the VCM are shown. This is a static investigation, dynamic investigations are shown in a previous work [5]. The results yield to a sensitivity of the VCM (only in axial direction) of

$$\frac{z}{u_a} = 1.1 \frac{\text{mm}}{V}$$  \hspace{1cm} (3)

in the linear range between $-0.5 \text{ V} \leq u_a \leq 0.5 \text{ V}$.

At the begin of a profile measurement at first a voltage sweep from $-0.5 \text{ V}$ up to $0.5 \text{ V}$ must be applied to the VCM and the corresponding FE signal has to be acquired so that we can derive the actual sensitivity of the FE signal to the specimen’s roughness according to its reflective properties. Later some measurement results will be presented based on this idea.

4 The setup using RTAI Linux

In this work there is also a focus on the controlling hard- and software because the students in the course should get to know and use open source software.

The structure of the entire system is shown in Fig. 4. An ordinary Desktop PC (Intel P4 2.8 GHz, 512 MB RAM) is used as the Master PC. The Master also contains the data acquisition device NI PCI 6221. Via TCP/IP the Master is connected to the Host PC. The Host is running Windows with an installed Version of MATLAB/Simulink (MATLAB R2010a [6]) and also Ubuntu (Lucid Lynx 10.04 [7]) is running in a Virtual Machine (Oracle VM VirtualBox 4.0.12 [8]).

On the Master PC the Linux distribution Ubuntu (Lucid Lynx 10.04) is installed. A new kernel (version 2.6.32-2, downloaded from [9]) with the applied RTAI patch (version 3.8, downloaded from [4]) is configured to make a realtime system out of it. Several pages in the web provide how–tos [10], [11] which can be used to configure the RTAI kernel. Also the driver interface Comedi (version 0.7.76, downloaded from [12]) is installed and RTAI is modified so that the data acquisition device can be used in realtime tasks.
During the installation of RTAI a folder named MATLAB is created. This folder is copied into the MATLAB directory of the PC where – after a small setup – the realtime task then can be developed in Simulink. Since the configured RTAI kernel contains Comedi support there are several Comedi blocks included. With the realtime workshop (included in Simulink) the code is generated and can be transferred to the Master. Subsequently, the code has to be compiled (the generated code contains a makefile) and can be started as long as the required realtime modules are loaded. The Ubuntu installation in the Host’s virtual machine has the same RTAI patched kernel and loaded modules as the Master. With QR-tailab [13] a connection to the realtime task can be established to view scopes and to change parameters of the realtime task.

In this work the NI PCI 6221 data acquisition card measures the voltages $A$, $B$, $C$ and $D$ (see section 3) as well as the voltage it applies to the VCM $u_a$.

The sampling time amounts to $t_s = 0.1$ ms.

In Fig. 5 a zoomed-in view of the processed FE signal and the corresponding VCM voltage is shown. The linear range of the FE signal crosses zero when the VCM voltage amounts to $u_a = 415$ mV. Furthermore the sensitivity of the FE signal can be computed by considering Eqn. 3 to

$$\frac{z}{FE} = 2.54 \frac{\mu m}{V}.$$  \hspace{1cm} (4)

Subsequently the VCM voltage is fixed to the zero crossing voltage and slightly adjusted so that the FE signal becomes zero. Now the CD is moved in radial direction along the pickup head. This is accomplished by a translation stage (Oriel Encoder Mike 18011) with a velocity of $v = 1 \mu m/s$. The acquired data is shown in Fig. 7, please remark that the plot

5 Measurement example

As a measurement example we first measure the profile of a CD. Hence CDs are easily available, their reflective characteristics are good and because they are one of the few available specimen with a microstructure on it they are used as specimen.

First, the CD is placed near and parallel to the pickup head. Then the objective lens is moved by the voice coil motor in axial direction (a delta voltage is applied to the VCM). This is done for finding the lens’ position where the reflection takes place in the focal plane and for acquiring the data for the calibration of the FE signal (see section 3). In Fig. 5 the screenshot of QRtailab measuring the signals $A$, $B$, $C$ and $D$ and the VCM delta voltage is depicted.
already shows the profile roughness with respect to the travelled distance, both in $\mu$m.

The track pitch is measured and amounts to $1.6\ \mu$m which corresponds to the specification of the CD standard. The profile shows a roughness of maximal $2\ \mu$m.

The setup shows some insufficiencies hence it is hard to position the pickup head and the CD perfectly parallel. Furthermore the calibration of the FE signal is difficult because the reflective characteristics are never constant. Another problem occurred when a controller was implemented. Because the FE signal's linear range is very narrow the controller is not able to position the lens in focus but rather switches from the negative to the positive defocus where the FE signal also goes zero. Nevertheless a micro structure can be measured despite the insufficiencies.

6 Conclusions

The development of a profile measurement system based on an optical pickup head using RTAI Linux was presented. The optical and electronic components of a CD pickup head and its working principle were introduced. The control and the data acquisition from a remote PC delivered measurement results which showed the functionality of the setup. Based on this work students will work in a lab course with open source/rapid control prototyping system. By using the introduced setup the costs for additional software were zero and students should learn to use similar setups for their own projects.

Acknowledgment

The authors gratefully acknowledge the partial financial support of the work presented in this paper by the Austrian Center of Competence in Mechatronics (ACCM).

References

[3] DVD pickup heads for optical measurement applications, Stefan Kostner, Michael J. Vellekoop, 125 (2008), Elektrotechnik und Informationstechnik (e&i)
Process Data Connection Channels in uLan Network for Home Automation and Other Distributed Applications

Pavel Píša\textsuperscript{1,2}
pisa@cmp.felk.cvut.cz
Petr Smolík\textsuperscript{1,3}
petr@smoliku.cz
František Vacek\textsuperscript{1}
fanda.vacek@volny.cz
Martin Boháček\textsuperscript{1}
bohacma8@fel.cvut.cz
Jan Štefan\textsuperscript{1}
honza.stefan@gmail.com
Pavel Němeček\textsuperscript{1}
pavel.nemecek1@gmail.com

\textsuperscript{1} Czech Technical University in Prague, Department of Control Engineering
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic

\textsuperscript{2} PiKRON s.r.o.
Kaňkovského 1235, 182 00 Praha 8, Czech Republic

\textsuperscript{3} AGROSOFT Tábor s.r.o.
Harantova 2213, 390 02 Tábor, Czech Republic

Abstract

The uLan protocol is the multi-master communication protocol aimed on small RS-485 control networks. It provides deterministic media access arbitration and it is open in design from its origin. An open-source implementation of the protocol has already been available for many years. The article focuses on its adaptation for use in distributed home appliances (switches, lights and HVAC components interconnection and control). For resource restricted control nodes, it was a challenging task to implement a flexible and persistent configuration of data and events direct routing between distributed nodes without need for permanent operation of commanding master. Because devices do not have resources to mutually examine their often large objects/properties dictionaries, the mechanism to map properties values into process data messages slots has been implemented. The message slots act as (virtual) wires which are setup by configuration tools running on PC which has enough resources to build and visualize full objects/properties model by examining of connected devices. Examples of developed devices using developed concept are presented at the end of the article together with tools available to help with fast prototyping of new devices and their testing in PC environment. The compilation of embedded devices code as native Linux binaries is quite straightforward because uLAN driver implementation is portable and provides same API when compiled for system-less nodes, GNU/Linux or Windows operating system environment.
1 Introduction

There is a need for a cheap, two wires bus communication between resource constrained MCU based node in many projects and application areas. Many standards exist but most of them require a special MAC hardware to be integrated onto MCU or attached as an additional communication chip. Many technologies have the disadvantage of being proprietary or at least controlled by (sometimes secretly held) patents, even those declared as public standards. The described solution is based on a different approach. It is targeted to standard UART hardware (with multi-drop or stick parity bit support) available on most MCUs and PC serial port interfaces and it has been developed as an open protocol from its beginning.

The article is divided to main parts. The first one describes protocol basic ideas leading to uLAN protocol design and implementation. Description starts from low level frame protocol and describes uLAN Object Interface (uLOI) higher level layer with a brief application example.

The second part focuses on process data exchange based on device properties/variables values mapping into communication channels distributed in publisher-subscriber manner.

2 uLAN Protocol

Origin and Initial Target Applications

The protocol design has been motivated by the need of control and data acquisition networking suitable for next generation of High Pressure Liquid Chromatography (HPLC) instruments sets designed by yet future PiKRON company forming group in 1992. The HPLC chromatography instruments do not require so fast command/data exchange for basic setups, but there are many parameters which have to be setup and should be monitored. The data types range from simple one scalar variable setup (wavelength, flow rate) to gradient time program and detector data stream (one float at 25 Hz in our case). There should be no loss of sampled data but grouping into longer packets is possible (group of 32 samples at the time is used in our case). The requirement has the ability to send some synchronization commands between instruments without latency added by data resending or even polling cycle controlled by a single master (PC). Because the development of new/different instruments and components to the modular system was expected, the protocol higher level layers need to support examination of instrument type and available properties/variables.

uLAN Protocol Overview

The initial design of instruments control electronics has been restricted to Intel-8051 based controllers due to its availability and price. These devices provide only single UART hardware for communication and their computational power is quite low. But they offer multi-drop (9-bit per character) feature which allows to suppress the need to process these received data characters (address bit clear / bit 8 = 0) which are not a part of message targeted to given instrument/network node (module in uLAN terminology). uLAN defines character values 0 · · · 0x64 with address bit set to address target module but only up to 64 masters are considered by media arbitration described later. The value 0 works as the broadcast address. Values from 0x75 · · · 0x7F range have control and data delimiters role. Values above 0x80 are used to release the bus by master after it finishes its mastering role in one message(s) exchange session. The whole range above 0x80 is used for bus release to encode releasing node/module address which allows to enhance fairness of communication channel capacity distribution between nodes. Due to standard UART behavior and need to synchronize on character basis, whole time to transfer character includes start and stop bit in addition to the address indication bit. The whole character transfer takes 11 bit times in uLAN case.

As a physical layer, RS-485 signal levels, wiring and transceivers have been selected. Because multi-master operation has been required (as stated above) some mechanism of media access control/arbitration has to be defined. The one solution is to use token passing (Profibus, BACnet MS/TP). But it requires to keep and update nodes lists in each communication node, initial single token selection and its regeneration after node failure is quite complex. RS-486 signalling does allow reliable collision detection on the wire when transceiver is switched to Tx direction. Switching between Tx and Rx direction and link level stabilization is much slower than available data rates as well. However simulation of dominant/recessive levels is possible by switching between Tx logic zero level and Rx direction when bus termination with quiet level bias to logic one is used.

uLAN deterministic distributed media arbitration has been partially inspired by Philip's I2C design. But to allow full speed data rates during message data phase and because UART hardware allows only control of transceiver Tx/Rx direction (in most
cases assisted by CPU code in ISR) only on whole character time granularity, the arbitration is based on switching between Tx zero and Rx for whole character time (sometimes implemented by break character send). Not like in I2C case, the arbitration needs to finish before target address and data are sent in transceiver fully driven Tx mode. The arbitration sequence is based on self node/module address to ensure unique dominant/recessive sequence for each node.

uLAN is targetted to control applications which require data reception acknowledgement and communication exchanges can be simplified by a direct reply by addressed device during a single arbitration cycle. Direct reply frame follows directly after initial frame end without media arbitration. Master releases the bus after last frame belonging to the given session. This is technique used in many other standards but the advantage of uLAN is mechanism generic enough that there is no need to use specialized command format knowledge on the master’s side of communication and required/expected single message session frames sequence can be prepared and passed to the driver on application level.

The single frame consists of destination address (DAdr) with address bit set, source address (SAdr), command (Com) followed by frame data characters. The end of data is delimited by one of four control characters describing the frame end kind. The simple frame consistency check byte (XorSum) follows. The frame is directly acknowledged if frame end kind specifies that. Then an direct reply frame can follow if indicated by frame end as well.

\[
T_{\text{arbW}} = (L\text{Adr} - Adr - 1) \mod 16 + 4 \cdot T_{\text{chr}}
\]

where \( L\text{Adr} \) is node address of the last node which has won arbitration and now releases the bus, \( Adr \) is the address of given node which prepares for bus use and \( T_{\text{chr}} \) is time to transfer one character. This setup ensures strict cycling of media access priority between nodes with messages prepared in Tx queue when only addresses up to 16 are assigned to nodes. If more nodes are used, the cycling between aliasing nodes is not ensured on deterministic basis but at least helps with some stochastic distribution.

The second phase ensures that node with lower own address wins arbitration when two or more nodes finish the first phase at the same time. The arbitration is based on sending next three dominant characters separated from initial one by precomputed time intervals \( T_{\text{arb},0} \), \( T_{\text{arb},1} \) and \( T_{\text{arb},2} \)

\[
T_{\text{arb},i} = ((Adr \text{ shr}(2 \cdot i)) \mod 4 + 1) \cdot T_{\text{chr}}
\]

If the activity from other node is detected during inactive interval time, the node abandons arbitration and restarts from the first phase. Direct binary coding and sending of own address as sequence of dominant recessive character intervals have not been selected because precise timing would be a problem through ISR responses. The addition of one dominant start bit and recessive stop bit around each arbitration bit would result in even longer phase two sequence \((3 \cdot T_{\text{chr}} \cdot 8 = 24 \cdot T_{\text{chr}})\) length.
bounded by next ranges
\[
T_{arbAll} \in (4 + 3 \cdot 2, 20 + 3 \cdot 5) \cdot 11 \cdot T_b (3)
\]
\[
T_{arbAll} \in (10, 20 + 35) \cdot 11 \cdot T_b (4)
\]

The whole time of one message arbitration cycle consisting of single frame and reception acknowledgment represents time interval \( T_{arbAll} + (3 + l_d + 2 + 4 + 1) \cdot 11 \cdot T_b \) where \( l_d \) is number of data bytes. If network with only 10 nodes with addresses \( 1 \cdots 11 \) is considered, the arbitration overhead is much lower due to shorter times of the second phase for modes assigned by lower addresses and because maximal length of the first phase applies only in case when same node requests bus repeatedly (see equation 1). The average message transfer time is more favorable for this case, if full Tx saturation from all nodes is supposed. The first phase time is \( 9 \times 5 \cdot 11 T_b \) and \( 1 \times 13 \cdot 11 T_b \) for this case. The second phase from 2 contributions evaluates to 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11 character times. The average arbitration time \( T_{arb} \) settles on \( (9.6 + 5.8) \cdot 11 \cdot T_b \) and whole message time is \( (l_d + 25.4) \cdot 11 \cdot T_b \). In case of quite common (for our HPLC applications) message length of 256 B and communication speed of 19200 Bd it takes 1.6122 s to send 10 messages (one from each station) and overhead caused by arbitration and other control characters represents 10\%. If the whole encoding schema is compared to synchronous communication which does not need any address, start and stop bits, the overhead causes 50\%. But even synchronous communication requires some bit-stuffing in real applications protocols and some media access control. On the other hand, if short messages of 8 bytes each are considered then uLAN protocol makes up much higher overhead about 300\% (550\% if counted on bit level).

The uLAN protocol compared to CAN can offer in case of dedicated or FPGA hardware solution up to 10 times higher transfer rates for bus of the same physical length because arbitration (requiring propagation of dominant/recessive level to whole link and back) is running with 11 times slower timing than actual data bytes. Other advantage is that during data transfer full active push/pull transceiver mode is used which provides better noise immunity and works well even if only single twisted pair of wires is used. CAN typically does not work well without ground interconnection. When compared to token passing networks, uLAN has much simpler (basically none) master node connection to the network and minimal delays are caused by node failure or switched off. The significant disadvantage of very high overhead for small messages can be adjusted by building higher level protocol in the way that multiple variables/properties transfers are grouped into single message.

### Higher Level Layers

There are multiple higher level services built above lower level uLAN messages and frames protocol described earlier.

#### Network Control Messages (uLNCS)

The commands from this group allow to check and change module/node assigned network address, check its identification and production serial number.

#### Dynamic Address Assignment (uLDY)

The mechanism to unveil newly attached nodes from new serial product number appearance, assign them free network address and detect node disconnection or switching off.

#### uLAN Object Interface Layer (uLOI)

The mechanism to retrieve list of device supported readable and writeable variables/properties, their names and data types.

Only very short description of use of the last mechanism fits in this article.

### 3 uLAN Object Interface Layer

The uLOI defines the system how to serialize objects (properties/variables) identification and their values in transferred messages. The service works with asynchronous reply as further master transfer after request sends service/command number to specific uLOI node/module. Multiple queries for objects values and/or their description can be serialized in a single message. The limitation is given only by the maximal length of a request and expected reply messages which is at least 1000 B for actual products. The controlling application can build model representing connected devices and then use this model to access data and control attached modules/instruments. The objects serialization and identification minimizes amount of metadata to minimize communication overhead. Each object in module is identified only by 16 bit Object Identification Number. No type, name nor data length for plain types is included in regular transfers. All these information has to be obtained by controlling application/system in advance through predefined OIDs for this purpose.

The significant advantage of the protocol and current uLAN driver implementation is that message can
be read from incoming queue by parts and OIDs are directly interpreted and the reply message is build again in “driver space” buffers. The second advantage is that reply allows to identify which objects data it contains. This allows to have more data request on the fly from different controlling nodes or applications.

The example of system utilizing many of uLAN services is CHROMuLAN HPLC control system developed by Jindrich Jindrich and PiKRON Ltd.

![FIGURE 3: uLOI in Devices and Corresponding Model Build in CHROMuLAN Application](image)

Many other applications have been developed at PiKRON company or by other uLAN adopters. I.e. Agrossoft Tábor FASTOP and FASTOS systems for automatic batch distribution of feed to pigs, cows and their systems for cow milking data collection. uLAN interconnect the feeding units with RF animal identification with central database in these systems for example.

4 Data Exchange in Home Control Application

The multi-master capability of uLAN, very low cost interconnection with use of a phone line grade cables, free bus topology, non problematic interfacing between many low cost microcontrollers and stable drivers for PC operating systems are features which speaks for spreading of uLAN into other areas as well. uLAN is not intended for high speed communication or hard real-time data exchange but these features are not required for the most tasks of home automation systems. That is why use of uLAN for heating monitoring and control, lights switching and ring-bells has been proposed by team preparing new home automation project at the Department of Control Engineering.

uLOI layer supports devices configuration and their state monitoring by higher level systems. But use of polling cycle by higher level system is significant disadvantage for home automation. The home appliances has to be equipped by system which allows direct communication between nodes in response to the incoming events. This is important not only to short latencies caused by polling cycle but even to allow system to provide at least basic functionality even in the case of higher level control system failure. It would be possible to use uLOI messages for direct data writes or reads to/from one appliance to objects located in other one. However, this would require mutual knowledge of the structure of appliances and require quite complex and memory resource huge OIDs list and types retrieval or made system inflexible by storing other device OIDs into firmware in fixed form.

The generic system for building uLAN Connection Network (uLCN) for processing the data exchange has been designed instead. This mechanism consists of two main specifications. The first there is defined new uLAN protocol level command/service for process data exchange (UL_CMD(PDO). The message of this type contains one or more blocks holding data corresponding to individual virtual “wires” connected between appliances. Each such wire is identified by its Connection ID (CID) and delivers data or events of some type.

uLAN PDO Connection Channels

The subsystem is designed for direct process data (PDO) exchange between devices (nodes/instruments). Every data transfer is identified by connection ID (CID). Design allows to map one or multiple uLOI dictionary objects (properties, variables) as data source or destination for given CID. The mapping is stored directly in devices. The mechanism allows to transfer multiple CID identified data values in single message. Receiver identifies data scope only by CID, no source address or device internal uLOI OID assignment or meta-data format is encoded in PDO messages or directly influence the processing. This allows to connect objects with different OIDs, group multiple objects under
a single CID, use broadcast to distribute data into multiple destination devices or even use more devices as data source for same CID. When device receives PDO message, it processes every CID identified data according to configured mapping. CIDs and their respective data for which no mapping is found are simply skipped. Only data types compatibility between mapped source and destination OIDs is required and sometimes this requirement can be even relaxed to some degree. If destination type is shorter then source, remaining bytes are skipped, counter case is illegal for actual implementation. Predefined constant data can be sent in response to event activation as well.

Command UL_CMD_PDO (0x50) is specified for PDO messages. Message format starts with two reserved bytes for future static extensions and one byte follows, which can be used for dynamic PDO messages header extensions in future. These bytes should be sent as zero for current protocol version. Each data block is preceded by its CID and data length. Maximal individual data block length is 127 bytes for actual implementation and is encoded in single byte. Format allows extension to two bytes in future if needed.

**Control of Data Mapping into Channels**

All configuration/mapping of PDO data source and processing of received PDO messages is done through device objects dictionary (uLOI). Exchanged data and meta-data stored in mapping tables have same format as is used for uLOI layer properties/data access.

The core component are ULOLPICO and ULOLPOCO mapping tables, both with same format structure. They are accessible as regular uLOI arrays of four field structures. Each array entry specifies mapping between CID and object dictionary entries. Simple one to one mappings are specified directly by entry by OID number. Complex mapping can specify offset into block of meta-data byte array instead of direct OID specification. This allows to serialize multiple objects/OIDs data under one CID, add execute command after CID data reception and distribution into uLODOI objects etc. Another possibility is to process the same received data by multiple mappings for the same CID. The special form to embed 3 bytes (OID + single byte) or 4 bytes (OID + 2 bytes) directly into ULOLPICO or ULOLPOCO mapping table entry is also supported.

**Events to Process Messages Mapping**

The ULOLPV2C array specifies, which CID/CIDs identified transfers should be initiated when given event number is activated. One event can be specified multiple times to trigger multiple CID transfers. The ULOLPV2C array entry specifies event number to CID mapping and some flags to nail down CID processing.

**5 Example Applications**

**DAMIC Home Automation Components**

The concept of the uLAN PDO connection channels is used in a components and appliances set which has been developed at the Department of Control Engineering to cover needs of heating, ventilation, air-conditioning (HVAC), light control and other home automation tasks:

![uACT 2i2ct - uLan Actuator and Temperature Sensor](image-url)
The uLAN uLOI, uLCN infrastructure is used on PC hardware which runs Linux or Windows operating systems but even Linux equipped access-point devices or PowerPC based boards are supported by Linux builds of uLAN driver.

**uLAN-Admin**

uLAN-admin is a set of Qt library based components which provide access to devices properties/variables by means of uLOI protocol, allows devices scanning, identification, monitoring and administration. The core component is library "libulproxy". The model of uLAN devices and their OI variables is built by the library in memory and creates abstraction to access uLAN network components over JSON RPC 2.0 interface. Thy library provides a routing of uLAN bus communication through TCP sockets as well. An utility library "libulqttypes" take care about conversion of OI variables values between uLAN bus data format and Qt types. Its primary purpose is to decode/encode byte arrays of uLAN communication to/from QVariant variables. uLAN-admin also contains exemplary application "browser" providing overview of devices on bus, which is based on above described libraries.

**uLAN-genmod**

uLAN GenMod is an application that allows to connect a virtual devices to uLAN bus. Each device is defined by two files. A graphical representation of a virtual device is described by QML (Qt Modeling Language). uLAN description is defined in XDS file (XML description), where are device’s name, address, serial number and device’s object interface. A whole house network and variables interconnection can be configured by uLAN-admin tool through ULOIPICO and ULOIPOCO tables, where is defined what PDO messages and CIDs device receive.

All above listed components can be combined together. The temperature controller uLTH can control a heater equipped by valve controlled by uACT for example. The uLMI can be used to indicate open window and this state can be routed to the uLTH to switch of heating when inhabitant opens doors for ventilation. The designed infrastructure is used in a thermal recuperation and ventilation units (VECO) as well.

One or more computers can be used to monitor and visualize components states and setup parameters and time programs over uLOI protocol and or can participate in PDO uLCN based data exchange.
and send. The application allows save this network configuration. The network configuration is transferable to real devices. The virtual device can control the real devices connected to uLAN bus and vice versa.

**FIGURE 7: uLAN-genmod - Application Main window with Two Devices**

### 6 Conclusion

The uLAN protocol and surrounding infrastructure have been used in many applications for years. They include two generations of HPCL instruments sets (third generation is in preparation now), more agricultural control systems and components, other serious production grade and hobbyists projects (i.e. HISC private house control network based on sole uLOI which components has been designed around the year 2005).

uLAN uLCN/PDO design started in 2008 and its actual version is complete and well tested. The approach is similar to CANopen dictionary and PDO idea but it is more flexible and suitable for wider size data types, generic arrays and inherits underlying uLOI layer flexibility. uLOI layer provides network introspection capabilities much better than many other standards offers. Yet the metadata overhead is kept very small for data exchange after initial device model retrieval phase.

The PDO mapping system has been tested on the CTU developed components for home automation during the DAMIC project. The initial versions of open-sourced management software utilizing Qt library is being developed as well. uLan driver and fully portable interface libraries allows to test even GNU/Linux builds of components and their interaction. The Qt based components builder and dictionary sources generator is in development to help newcomers to test capabilities and speed up new nodes design.

The uLCN/PDO mapping extension and openness of the project make it an excellent candidate for smaller hobbyists home automation projects. The minimal requirements for small nodes (only UART with software parity control) allows to base such designs on a cheap Cortex-M3 or even smaller MCUs. The design of higher communication layers can be utilized even in combination with different link technologies or can serve as an inspiration for other similar projects at least.

uLan project is a live thanks to more companies’ and university members’ participation. The actual version of the code used in multiple real sold products is available from uLAN project SourceForge GIT repository and file releases archives.

### References


Application of RT-Preempt Linux and Sercos III for Real-time Simulation

Michael Abel\textsuperscript{a,b}, Luis Contreras\textsuperscript{a}, Prof. Peter Klemm\textsuperscript{a,b}
\textsuperscript{a}Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), Seidenstr. 36, 70174 Stuttgart, Germany
University of Stuttgart
\textsuperscript{b}Graduate School of Excellence advanced Manufacturing Engineering (GSaME), Nobelstr. 12, 70569 Stuttgart, Germany
University of Stuttgart
\{Michael.Abel, Peter.Klemm\}@isw.uni-stuttgart.de, Luis.Contreras@lpmc.de

Abstract

This paper presents the application of RT-Preempt Linux in a virtual commissioning scenario. In this scenario, a proprietary Programmable Logic Controller (PLC) is connected to a real-time simulation model. The model is located on a separate Linux personal computer which simulates for example the hardware of a production machine. Furthermore, the controller and the simulation computer are connected through the sercos III automation bus. The simulation computer uses a sercos III PCI card as communication hardware in combination with a user space IO (UOI) driver. This allows the execution of the simulation model and the sercos III driver as real-time processes on the simulation computer. The sercos III driver was adapted in order to imitate the bus-interface of a custom sercos III bus-coupler and to provide easy integration into the PLC engineering system. Moreover, variables in the PLC can be coupled to input and output values of the simulation model. With this virtual commissioning method, it is possible to reduce the time to market of a machine, since writing and testing the PLC code for the controller can be done in parallel to the construction of the hardware.

1 Introduction

Due to the constant pressure of the market, manufacturers have to bring new production machines to the market regularly. To reduce the time to market of their machines manufacturers have to find methods to decrease the overall development time. Especially at companies where machines are built for special purposes, every machine is unique and even needs software which has to be developed particularly for one machine. The overall development time of a machine can be decreased dramatically by parallelisation of development tasks. Since most of the development tasks depend on each other, dependencies have to be taken into account. Software components can be written by means of the system specification. Parts of the software can even be tested by software tests before the hardware of the machine is finished. But testing and bug-fixing of functions which depend on the availability of the machine can only be done when the hardware is available. The same problem comes up at the commissioning of the machine, since the whole machine needs to be finished until commissioning can begin. To solve this problem, a virtual commissioning of the machine can be performed. This can be achieved by applying a virtual machine model which simulates the mechatronic parts of the machine. Since production machines are usually controlled by real-time systems, a real-time model for hardware in the loop simulation is well suited for this purpose [1]. Hardware in the loop simulation means that a simulation model calculates the mechatronic behaviour of a machine while its functions are controlled by real control programs.
2 Theory & State of the Art

This chapter gives a short introduction into the technologies and software systems which are used within this project.

2.1 RT-Preempt Linux and User Space IO Drivers

Linux with real-time kernel preemption (RT-Preempt) is an enhancement to the Linux kernel. The aim of this patch is to enable real-time capabilities in the Linux kernel. The RT-Preempt patch allows user space programs to run in real-time [2], [3].

The User Space IO (UIO) driver model enables drivers to run in the user space of a Linux system [4]. UIO drivers are a convenient method to implement drivers for non-standard and rarely used hardware which does not fit into the regular kernel subsystems. The memory of a device is mapped into addresses which are accessible from user space memory segments. To handle interrupts, a user-space thread can be applied. In addition, a small interrupt handler within the kernel space is necessary to wake the thread. With this functionality it is possible to write drivers for special purpose devices without the need to handle complex in-kernel structures. UIO-drivers are often used to handle networking devices for field-buses on systems which are running on RT-Preempt Linux.

2.2 Serial Real-Time Communication System (sercos) III

The automation bus sercos III is an Ethernet based field-bus system which can be used in a wide range of automation applications. Sercos III is standardised by the association sercos International e.V. [5]. In the following the term sercos is used as abbreviation to sercos III. Sercos is based on standard Ethernet and uses Ethernet frames to communicate on the bus. A sercos network consists of a bus master and several slave devices (Figure 1).

![Figure 1: Sercos III ring with master and slave devices.](image)

Each device is equipped with two Ethernet ports. The preferred bus topology is a ring structure, since a ring provides more redundancy than a star topology. Apart from this, a line topology with one or two lines (i.e., broken ring) can be used as well. Sercos uses a sophisticated device model which classifies every bus component into different classes of functionality. According to the device model it is possible to distinguish between servo-drives, IO-devices and other automation hardware.

Furthermore, a parameter model was introduced to describe functional interfaces of field-bus devices. Every device has a set of sercos parameters which characterise the interface of the device. Parameters can be accessed by unique identification numbers (IDN). Furthermore, a parameter contains a description of the parameter as string, several attributes and the data of the parameter with a fixed or variable length.

Sercos uses a start-up phase with five different communication phases (CP) which are usually called CP0 to CP4. When the communication phase has passed the early stages and reaches CP4, real-time communication is active, devices and connections are set up adequately and real-time data can be transmitted. Furthermore, sercos devices can be described in the sercos Device Description Markup Language (SDDML) which is based on the Extended Markup Language (XML).

2.3 Passive sercos III PCI Card

Custom and PC based sercos slaves can be built by equipping PCs for example with sercos III PCI networking cards from the company Automata [6]. The card contains standard Ethernet communication hardware and a FPGA in order to connect it to the PCI bus. To bring the card to operation, a proprietary driver is necessary. This Sercos Slave Driver (SSLV) is written OS independently and contains a hardware abstraction layer which can be ported to other operating systems easily. The card is named "passive" because a driver, which executes the sercos networking stack, and a real-time operating system are necessary to use the card.

This project utilises a port of the SSLV to RT-Preempt Linux. The SSLV is running as UIO-Driver within the user space. To support the user space part of the driver there is also a small kernel module called uio-sercos3 in the mainline kernel. Figure 2 shows a rough overview of the SSLV.
The driver consists of two parts: A small kernel module called uio-sercos3 and the user space application of the SSLV. The user space part is separated in two threads: A UIO interrupt handler thread which is executed with a high real-time priority. And the UserTask, a regular user space part of the SSLV. The UserTask has a lower real-time priority and needs to be executed at least once in a sercos communication cycle. Moreover, a database for IDNs is contained in the SSLV which can be interfaced from the bus and from the UserTask.

2.4 MLP VEP and IndraWorks

In this project a proprietary Programmable Logic Controller (PLC) produced by the company Bosch Rexroth [8] is deployed. The MLP VEP is a PC based PLC which is equipped with several sercos ports and acts as sercos master device on the bus system. Furthermore, it can be programed and configured with the engineering tool IndraWorks. It is able to execute PLC programs written in the five languages specified in IEC 61131-3 [9] and has additional Motion Logic Control (MLC) functionality.

2.5 Virtuos

Virtuos [10],[1] is a simulation software which enables the execution of mechatronic and other models in real-time. Within this project a Virtuos-S variant is deployed which can be executed on a Linux system. To synchronise Virtuos with other programs semaphores are a convenient method. They can trigger the beginning of a simulation step or inform that an simulation step has finished. Other programs can use a Virtuos library that provides access to the input and output ports of the simulation model.

3 Problem Definition

To perform virtual commissioning of a production machine, a real-time simulation model of the hardware is necessary. This virtual machine model can usually be executed by Virtuos on a PC which also executes a PLC or an other type of controller software. Indeed, the project specification demanded the application of a MLP VEP PLC, a proprietary controller which is not able to execute a Virtuos model. Moreover, it provides no standard interfaces to connect it to a virtual machine model. To solve these problems, a new method is desired to connect the PLC to the model. Since the PLC is a proprietary device which needs to be programed by proprietary software there are no simple methods to extend it by custom real-time tasks.

The solution to this problem is to move the simulation model to a PC and let it communicate with the PLC. Since the communication between PLC and simulation model needs to be run in real-time, a field-bus can be used to connect the PLC to the simulation model (Figure 3).

Since the MLP VEP PLC offers direct access to the sercos field-bus, sercos will be applied as field-bus in this project. As this field-bus will also be used later on in the production machine, it does also also simplify the integration of other hardware which will be connected to the PLC afterwards.

4 Approach

Sercos uses a device model which can provide different types of automation devices. But no device has an interface which resembles the complexity of...
Virtuos simulation model. A simulation model produces and consumes a high amount of data in every simulation cycle. In this project it is sufficient to provide exchange of floating point values and integer values, since the simulation consists of a mechatronic model. The interface between the PLC and the simulation was defined as an amount of integer and floating point values. To be able to integrate the simulation model into the bus system the interface of the model was enhanced to resemble the interface of a (very large) bus-coupler (see figure 4).

A bus-coupler is a standard field-bus device that usually couples various kinds of electrical input and output signals to the field-bus. With this solution it is possible to ”hide” the interface of the simulation model behind the interface of a large bus-coupler. This also enables easy integration of the simulation model into the PLC program, since the interface of the simulation model looks like a bus-coupler. Due to the application of the real bus system to connect to the simulation model, the timing which comes to use later is also applied.

5 System Design

This chapter introduces the design of the simulation system. Figure 5 shows the overall system structure.

5.1 Communication Concept

The concept for distributing data within the system is shown in figure 6.

Simulation data is transferred in data packets. A data packet consists of a certain amount of integer values and a certain amount of floating point values. To support the simulation with enough data, packets with 64 32-bit integer values and 32 64-bit floating point values are used. All the values are composed together to data packets of 512 bytes in size. Data packets are composed in the PLC and are put into Ethernet frames which are sent via the field-bus to the simulation PC. In the PC, packets are copied to the address space of the SSLV. The Virtuos-IO (VIO) thread synchronises the execution of the simulation, decomposes the packet into data types and copies them to a memory mapped address space which is shared with the simulation. The data transfer back from the simulation to the PLC works in a similar manner.
5.2 Controlling the Simulation from the PLC

Within the programming system of the PLC, IO ports of devices can be mapped to variables. Variables can be connected to either input or output ports. Afterwards, IO-operations can be done by setting bit-masks in the PLC program. In addition, field-bus devices can be added to the IndraWorks project from a device database. The database can be extended by device descriptions. For sercos devices this can be achieved by using files in the SDDML-Language. Since the bus-coupler which is used in this project is not a standard off-the shelf bus-coupler, a SDDML file was written which describes a very large bus-coupler. The file was added to the device database of IndraWorks to be able to use it within the PLC program.

Moreover, a data structure was created which combines all the IO-data that has to be send or received in one communication cycle. The structure contains a certain amount of 32 bit integer and 64 bit floating point variables (See listing 1).

```c
typedef struct {
    double doubles[32];
    int    ints[64];
} io_type;
```

Listing 2: Specification of a data packet in the SSLV

Since the size of bytes in the structure is equal to the size of bytes of IO-data in the bus-coupler it is possible to connect the complete structure to the IO-configuration at once. To support input and output data, two structures were added to the input and to the output of the device. Consequently, a regular PLC program can be used to perform calculation input and output operations.

5.3 Enhancing the sercos Slave Driver (SSLV)

The SSLV provides two different tasks. On the one hand, it connects to the field-bus system to exchange data with the PLC System. On the other hand it is used to connect to the simulation and exchange data with the simulation. Since both tasks have critical timing behaviour they are executed separately in two threads within the SSLV. To move data from one thread to another they write into global data structures. One thread is responsible for covering sercos communication and the other thread handles the data exchange with Virtuos. As a first step, the sercos interface of the IDN database of the SSLV was enhanced to emulate the bus-interface of a standard off the shelf bus-coupler with just 16 bits of IO-data. This is an error prone process since there is no description which IDNs are retrieved and evaluated by the PLC during the start-up phases. The fields for cyclic real-time data were extended afterwards to the size of 512 bytes as specified in the custom SDDML file of the device. In the final configuration 512 bytes of data are transferred from the PLC to the simulation and the same from the simulation to the PLC in every communication cycle. Data composition and decomposition is also done by the communication thread. Listing 2 shows the specification of a data packet in C source code:

```
typedef struct {
    double doubles[32];
    int    ints[64];
} io_type;
```

FIGURE 7: Execution model of the simulation and the VIO-thread

The execution model of both processes is very similar to those of a traditional producer-consumer model. Two semaphores are deployed: The "start"
semaphore has the purpose to signal the beginning of a simulation step, the "end" semaphore signals the end of a simulation step. The VIO thread is started at $T_N$. When the VIO thread has completed its data transfer to the simulation, the simulation is started. The simulation executes one simulation cycle and signals the end of the cycle to the VIO-thread. Since the exact simulation time varies from application to application, the VIO-thread does not start data transfer immediately but sleeps until the next $T_{N+1}$ to be in time with the other parts of the system.

6 Conclusion

This paper presents how RT-Preempt Linux can be used for real-time simulation and the virtual commissioning of production machines. A proprietary PLC is connected to a simulation PC which executes the real-time simulation model. The automation bus sercos III is used to transfer data in a deterministic manner between PLC and simulation PC. To adapt the simulation model to the field-bus, its interface is hidden behind the interface of a bus-coupler device. For this purpose a sercos III PCI networking card is utilised. The driver of the card is enhanced to emulate the interface of a bus-coupler and to transfer data between the bus and the simulation model. The simulation model is executed by the simulation software Virtuos on the simulation PC. With this setup, PLC programs for controlling production machines, which need run on their (proprietary) and unmodified target hardware can be tested by means of simulated mechatronic hardware. Accordingly, the time to market of a production machine can be reduced by parallelisation of development tasks. As testing of programs which control or depend on mechanical hardware can be tested without the real hardware to be available.

7 Future Work

At current, only one field-bus device can be emulated by the PCI card. This is why a small hardware abstraction layer in the PLC code is necessary to switch from simulated to real hardware. In a follow-up project, the emulation of more than one sercos III devices will be possible. Consequently, the simulation may cover more than one field-bus devices at one. As a result it will, be feasible to switch between simulated hardware and real hardware without the need for any changes in the PLC.

8 Acknowledgement

The authors would like to thank the German Research Foundation (DFG) for financial support of the projects within supporting the Graduate School of Excellence advanced Manufacturing Engineering (GSaME) at the University of Stuttgart.

References


Lachesis: a testsuite for Linux based real-time systems

Andrea Claudi
Università Politecnica delle Marche, Department of Ingegneria dell’Informazione (DII)
Via Brecce Bianche, 60131 Ancona, Italy
a.claudi@univpm.it

Aldo Franco Dragoni
Università Politecnica delle Marche, Department of Ingegneria dell’Informazione (DII)
Via Brecce Bianche, 60131 Ancona, Italy
a.f.dragoni@univpm.it

Abstract
Testing is a key step in software development cycle. Error and bug fixing costs can significantly affect development costs without a full and comprehensive test on the system. Nevertheless, no comprehensive testsuites is able to assess the functionality or the conformance to the real-time operating systems standards of the Linux kernel and of real-time nanokernels that rely on it.

In this paper we propose Lachesis, an automated testsuite derived from the LTP (Linux Test Project) real-time tests. Lachesis is designed with portability and extensibility as main goals, and it can be used to test Linux, PREEMPT_RT, RTAI and Xenomai real-time features and performances. It provides some tests for SCHED_DEADLINE patch, too. Lachesis is now under active development, and more tests are planned to be added in the near future.

1 Introduction

Linux kernel is being increasingly used in a wide variety of contexts, from desktops to smartphones, from laptops to robots. The use of Linux is rapidly growing due to its reliability and robustness.

Thanks to these qualities, Linux is widely used in safety and mission critical systems, too. In these contexts, time is extremely important: these systems must meet their temporal requirements in every failure and fault scenario. A system where the correctness of an operation depends upon the time in which the operation is completed is called a real-time system.

Over the years various solutions have been proposed to adapt the Linux kernel to real-time requirements. Each of them has found extensive applications in production environments. Some of these solutions, like PREEMPT_RT [1] and, more recently, IRMOS [2] and SCHED_DEADLINE [3], are based on a real-time patch to the Linux kernel, which introduces or improves some essential features for a real-time operating system.

Other solutions use a real-time nanokernel in parallel with the Linux kernel. This is made possible through the use of an hardware abstraction layer on top of the hardware devices, that listens for interrupts and dispatches them to the kernel responsible for managing them. RTAI [4] and Xenomai [5] nanokernels, both built on the top of Adeos [6] hardware abstraction layer, belong to this category.

Although first efforts to enhance and introduce real-time features in the Linux kernel are now more than ten years old, nowadays there are no comprehensive testsuites able to assess the functionality or the conformance to the real-time operating systems standards for the Linux kernel and for the nanokernels that rely on it.

Properly testing the Linux kernel is not easy, since the code base is continuously growing and in
rapid evolution. We need more efficient, effective and comprehensive test methods, able to ensure proper software behaviour in the wide range of situations where systems can be deployed. For example, tests are critical in an environment where a malfunctioning system can seriously damage machinery, structures or even human life.

Nowadays there are many automatic testsuites, covering an increasingly wide range of kernel features. Many of these testsuites make it possible to functionally test file systems and network stack, to evaluate efficiency in memory management and in communication between processes, and to assess standards compliance. Very few of them make functional testing on real-time features, and none of them test performances or conformance with real-time features.

1.1 Paper contributions

In this paper we propose Lachesis, an automated testsuite for Linux based real-time systems, derived from the LTP real-time tests. Lachesis main goals are:

- to provide extensive and comprehensive testing of real-time Linux kernel features
- to provide a common test environment for different Linux based real-time systems
- to provide a set of functional, regression, performance and stress test, either developing or porting them from other testsuites
- to design and experiment a series of build tests
- to minimize development time for new tests
- to make the testsuite extensible and portable

Several real-time tests were ported to Lachesis from other testsuites. In this paper we also detail porting procedures and results.

1.2 Paper structure

The rest of this paper is organized as follow: section 2 illustrates a taxonomy for testing methodologies; section 3 briefly introduces the most important testsuites for the Linux kernel and give some reasons why to choose LTP as start point to develop a new real-time testsuite; Section 4 illustrates Lachesis and the tests it includes; Section 5, finally concludes the paper.

2 Taxonomy of testing methodologies

In software engineering, testing is the process of validation, verification and reliability measurement that ensure the software to work as expected and to meet requirements.

The Linux kernel has been tested since its introduction. In the early stage of development tests were ad-hoc and very informal: every developer individually conducted tests on the portion of code he developed, with his own methodologies and techniques; frequently tests came after end-users bug reports, and were aimed at resolving the problem, identifying the section of code causing it.

Over the years kernel grew across many different architectures and platforms. Testing activities became increasingly difficult and costly in terms of time, but remained very critical for kernel reliability, robustness and stability.

For this reason different testing methodologies and techniques for the Linux kernel were experimented and used. Indicatively, these methods can be grouped into seven categories [7].

2.1 Built-in debugging options

This kind of tests must not be done simultaneously with functional and performance tests. It consists in a series of debugging options (CONFIG_DEBUG_* in the Linux kernel) and fault insertion routines that allow the kernel to test itself.

2.2 Build tests

Build tests just compile the kernel code searching for warnings and errors from the compiler. This kind of tests is an extensive problem, for a number of different reasons:

- Different architectures to build for;
- Different configuration options that could be used;
- Different toolchains to build with;

2.3 Static verification tests

This kind of tests is designed to find bugs in the code without having to execute it. Static verification tests
examine the code statically with tools like sparse, LClint [8] (later renamed as splint), and BLAST [9].

2.4 Functional and unit tests

Functional and unit tests are conceived to examine one specific system functionality. The code implementing a feature is tested in isolation, to ensure it meets some requirement for the implemented specific operation. Crashme [10] is an example of this kind of test.

2.5 Regression tests

Regression tests are designed to uncover new errors and bugs in existing functionalities after changes made on software, such as new features introduction or patches correcting old bugs. The goal for this kind of tests is to assure that a change did not introduce new errors.

Regression testing methods are different, but in general consist in rerunning previously ran tests and evaluating system behaviour, checking whether new errors appear or old errors re-emerge.

2.6 Performance tests

Performance tests measure the relative performance of a specific workload on a certain system. They produce data sets and comparisons between tests, allowing to identify performance changes or to confirm that no changes has happened. In this category we can include kernbench, a tool for CPU performance tests; iobench, a tool for disk performance tests; and netperf, a tool to test network performance.

2.7 Stress tests

Stress tests push the system to the limits of its capabilities, trying to identify anomalous behaviours. A test of this kind can be conceived as an highly parallel task, such as a completely parallelized matrix multiplication. A performance test running under heavy memory pressure (such as running with a small physical memory), or in a highly resource-competitive environment (competing with many other tasks to access the CPU, for example) can become a stress test.

3 Automated testsuites

Testing is expensive, both in terms of costs and time. Automation is a good way to reduce economic and human efforts on testing. A number of automated testing environments for the Linux kernel has been proposed, each with its own strengths and weaknesses.

In the following paragraphs the most important test suites for the Linux kernel are presented. Goals and basic concepts which guide their design are stated.

3.1 IBM autobench

IBM autobench is an open source test harness\(^1\) conceived to supports build and system boot tests, along with support for profiling [7]. It is written in a combination of perl and shell scripts, and it is fairly comprehensive.

Autobench can set up a test execution environment, perform various tests on the system, and write logs of statistical data. Tests can be executed in parallel, but test control support is basic and the user have almost no control over the way tests are executed. Error handling includes the success or failure of the tests, but is a very complex activity and must be done explicitly in all cases. In addition the use of different languages limits testsuite’s extensibility and maintainability.

IBM autobench project is inactive since 2004, when the last version was released.

3.2 Autotest

Autotest is an open source test harness capable of running as a standalone client. It is easy to plug it into an existing server harness [11], too. Autotest provides a large number of tests, including functional, stress, performance, regression and kernel build tests. It supports various profilers, too.

Autotest is written in python, which enables it to provide an object oriented and clean design. In this way testsuite is easy to extend and maintain. Including python syntax in job control file, for example, users can take more control on test execution. On the other hand, python is not widely used in the real-time community, and is not suited for real-time tests or applications development.

\(^1\)A test harness is an automated test framework designed to perform a series of tests on a program unit in different operating conditions and load, monitor the behaviour of the system and compare the test results with a given range of good values.
Autotest has built-in error handling support. Tests produce machine parsable logs; their exit status are consistent and a descriptive message of them is provided. A parser is built into the server harness, with the task of summarizing test execution results from different testers, and formatting them in an easy consultation form.

Autotest includes very few tests to examine the Linux kernel from a real-time point of view; almost all of them are functional tests. Moreover, it does not include any compliance test on real-time standards.

### 3.3 Crackerjack

Crackerjack is a testsuite whose main goal is regression testing [12]. It provides:

- automatic assessment of kernel behaviours
- test results storage and analysis
- incompatibilities notification
- test result and expected test result management (register, modify, remove)

Crackerjack integrates a branch tracer for the Linux kernel system calls, but over time has been revised to easy future extension to other operating systems.

It is implemented using Ruby on Rails. This makes it easy to modify it, ensuring a low maintenance cost and simplifying development of new tests. However, as for python, Ruby is not suited for real-time tests or applications development.

Crackerjack integrates a branch tracer for the Linux kernel, called btrax. Btrax is a tool to analyze programs effectiveness. Crackerjack uses btrax to trace the branch executions of the target program, to analyze the trace log file, and to display data about coverage and execution path. btrax makes use of Intel processors' branch trace capabilities, recording how much code was tested.

Crackerjack does not support conformance, performance or stress tests, and does not include any functional test on real-time features.

### 3.4 rt-tests

rt-tests [13] is a popular testsuite developed to test the PREEMPT_RT patch to the Linux kernel. It is developed by Thomas Gleixner and Clark Williams, and it is used in the OSADL lab, across various hardware architectures, for a continuous testing. rt-tests includes ten different tests for real-time features.

* cyclicstest* [14], for example, is a well known test that measures the latency of cyclic timer interrupts. Through command line options, the user can choose to pin the measurement thread to a particular core of a multi-core system, or to run one thread per core. Cyclicstest works by creating one or more user space periodic thread, the period being specified by the user. The accuracy of the measurement is ensured by using different timing mechanism.

Another interesting test is `hackbench`. It is both a benchmark and a stress test for the Linux kernel scheduler. Hackbench creates a specified number of pairs of schedulable entities which communicate via socket. It measures how long it takes for each pair to send data back and forth.

rt-tests is a good and well established suite to test Linux kernel real-time features. However it is conceived primarily to test the PREEMPT_RT patch set, so it's quite difficult to extend it to other Linux based real-time systems. For example, it contains a couple of tests based on a driver for the Linux kernel; in systems such as Xenomai, the use of a driver as this causes a mode change in which a real-time task switch from the Xenomai to the Linux environment.

Thus, the task experiences much longer latencies.

Test results are outputted in a statistical summary, rather than in a boolean "PASS" or "FAIL". Unfortunately rt-tests do not provide any mechanism for collecting the results and present them in a machine parsable form.

### 3.5 LTP - Linux Test Project

LTP (Linux Test Project) is a functional and regression testsuite [15]. It contains more than 3000 test cases to test much of the functionalities of the kernel, and the number of tests is increasingly growing. LTP is written almost entirely in C, except for some shell scripts.

In recent years LTP has been increasingly used by kernel developers and testers and today is almost a de-facto standard to test the Linux kernel [16] [17]. Linux distributors use LTP, too, and contributes enhancements, bug fixes and new tests back to the suite.

LTP excellence is testing Linux kernel basic functionality, generating sufficient stress from the test cases. LTP is able to test and stress filesystems, device drivers, memory management, scheduler, disk I/O, networking, system calls and IPC and provides a good number of scripts to generate heavy load on the system.
It also provides some additional testsuites such as pounder, kdump, open-hpi, open-posix, code coverage [18], and others.

LTP lacks support for profiling, build and boot tests. Even if it contains a complete set of tests, LTP is not a general heavy weight testing client.

LTP also lacks support for machine parsable logs. Test results can be formatted as HTML pages, but they are either “PASS” or “FAIL”, and for tester is more complex to understand the reasons behind failures.

LTP has a particularly interesting real-time testsuite, that provides functional, performance and stress tests on the Linux kernel real-time features. To the best of our knowledge, LTP is one of the few testsuites that provides such a comprehensive and full featured set of tests for Linux real-time functionalities.

4 Lachesis

All analysed testsuites seem to suffer some key failings in relation to testing the Linux kernel real-time features.

Many of them seem to have little consideration for real-time features in the Linux kernel. A great part of them (with the notable exception of LTP and rt-tests) does not offer any real-time functional, performance or stress test.

Usually it is simple to design and develop a new test inside an existing testsuite. However it is very difficult to extend an entire testsuite in order to test some new real-time nanokernels. System calls analysed in tests, in fact, may differ syntactically from one kernel to another maintaining the same functionality. Moreover some nanokernels, such as RTAI and Xenomai, provides some real-time features through additional system calls, for which specific tests should be developed.

Another problem is the lack of machine parsable results. There is no standard way to consistently communicate results to the user: often we have not any detail on the reason that led a test to failure.

Lastly, every testsuites has grown rapidly and chaotically in response to the evolution of the Linux kernel. For this reason they are not easy to understand, maintain and extend.

The lack of a comprehensive testsuite to meet previously exposed needs led us to develop Lachesis.

The ambitious goal of Lachesis is to provide a common test environment for different Linux based real-time systems. Therefore it seems reasonable to start from an existing, accepted and widely used testsuite, to adopt its principles and apply them to a new testsuite, conceived with other goals and priorities.

We choose LTP as a starting point for Lachesis. There are many reasons behind this choice. First, LTP is one of the few testsuites able to provide a set of tests for Linux kernel real-time features, and a large number of testers use it. Second, LTP has a well established and clean architecture. It makes use of two main libraries, liblrttest which provides an API to create, signal, join and destroy real-time tasks, and libstats which provides an API for some basic statistical analysis and some functions to save data for subsequent analysis. Last, LTP provides a logging infrastructure. This is an important and desirable feature for Lachesis, too.

We believe it is of little significance to compare the results of tests to absolute values statically built inside the testsuite. In fact, varying the hardware to test, these values should vary as well. So, unlike LTP, Lachesis provides a boolean pass/no pass output only on functional tests; by contrast, in performance tests it outputs a statistical summary of the results.

4.1 Architecture

Lachesis is designed to analyse a variety of Linux based real-time systems; therefore it provides a straightforward method to build tests for different kernels. During the configuration Lachesis probes the system to determine which nanokernels or real-time patches are present, and instructs the compiler to produce in output different executables for each system to be tested. A set of scripts is provided to execute tests sequentially; launching these scripts, tests are executed one after another and tests results are stored in logs for subsequent analysis.

Liblrttest had to be rewritten to support both RTAI and Xenomai primitives. Lachesis maintains liblrttest API, extending it to provide advanced real-time features, typical of Linux-based nanokernels. Basic real-time features are provided encapsulating real-time specific function into the pre-existing API, thus concealing them from the user. For example, the create_task() primitive was modified to take into account the corresponding primitives for Xenomai and RTAI.

As a result, it is possible to write a single test for a specific real-time feature and use it to test all supported systems, thus increasing testsuite’s porta-
The logging infrastructure of Lachesis is based on LTP features. Libstat is used to provide a set of functions to store data and make statistical calculations on them; other functions are used to write files with reports on data. These files are written in an easy readable form (though they are not yet machine parsable) and stored in an unique subdirectory.

4.2 API

One of the principal goals of Lachesis is to provide a single API to develop tests for a number of different Linux based real-time systems. To do that, it provides an API that can be divided in four sections: tasks management, time management, buffers management and data management.

Tasks management API has been largely changed from the original LTP API, to provide support for testing on RTAI and Xenomai nanokernels. In particular, the `create_thread` primitive has been modified to call the corresponding Xenomai primitive, if Lachesis is requested to compile test for Xenomai nanokernel. RTAI’s approach is quite different and consists in ensuring that certain section of code can be scheduled in accordance to some hard real-time algorithm, calling some specific functions immediately before and after them. We have addressed this particular mechanism defining four macros, to be used inside the tasks to be defined:

- `__RTAI_INIT` defines task descriptor, scheduler and system timer
- `__RTAI_START_TASK` starts hard real-time section
- `__RTAI_STOP_TASK` ends hard real-time section
- `__RTAI_END` deletes the task

In all the other primitives, where approaches do not differ so much from system to system, small changes have proved to be enough. Two primitives to create periodic and deadline tasks were added.

Time management API was extended with many primitives. A `RTIME` structure was defined to standardize time management between the different systems to be tested. Two functions have been defined to make additions and subtractions on it. Nanosleep and busy work primitives have been modified to call specific real-time sleep functions. A primitive was added to end the current period for the calling task.

Buffers management API remained almost unchanged from original API. Few changes were made to support RTAI’s memory management primitives, changing `malloc()` and `free()` to `rt_malloc()` and `rt_free()` where appropriate.

Data management API remained also unchanged from original LTP API, since these functions only deal with data collected from tests previously carried out. We plan to work on this API in the near future, to support XML parsing and to format results adequately.

In addition to the API, we needed to introduce some macros to replace some standard functions with their real-time counterpart, provided by a particular nanokernel. For example, we have a macro to replace `printf()` with `rt_printf()` instances, if Lachesis have to build tests for Xenomai nanokernel.

4.3 Tests included in Lachesis

Below we briefly present all the tests currently included in Lachesis. Some of them were ported from different testsuites, others were developed as a part of this work. In the naming scheme we have tried to adopt the following principle: the first word is the parameter measured by the test, the second indicates the way in which measurement is made. We will indicate when the test is considered to be passed and the category it belongs to.

Tests are supposed to be done in an unloaded system. When a load is necessary, the test itself generates it. It’s worth to say that Lachesis don’t take into account power management features. So it’s up to the user to ensure that tests are running under the same power management conditions. This could be done disabling the power management in the configuration of the Linux kernel.

1) `blocktime_mutex` is a performance test that
measures the time a task waits to lock a mutex. Test creates a task with higher priority, one with lower priority, and some tasks at medium priority; highest priority task tries to lock a mutex shared with all the other tasks. Test is repeated 100 times.

2) func_deadline1 is a functional test we developed, conceived to be used only on SCHED_DEADLINE patched kernels. It creates a task set with $U = 1$, using the UUniFast [20] algorithm not to bias test’s results. Task set is scheduled, and test is passed if no deadline is missed.

3) func_deadline2 is a functional test we developed, conceived to be used only on SCHED_DEADLINE patched kernels. It creates a task set with $U > 1$, using the UUniFast algorithm not to bias test’s results. Task set is scheduled, and test is passed if at least one deadline is missed.

4) func_gettime verifies clock_gettime() behaviour. It creates a certain number of tasks, some of them setted to sleep, some other ready to be scheduled. Test is passed if the total execution time of sleeping tasks is close to zero. This is a functional test.

5) func_mutex creates a number of tasks to walk through an array of mutexes. Each task holds a maximum number of locks at a time. When the last task is finished, it tries to destroy all mutexes. Test is passed if all mutexes can be destroyed, none of them being held by a terminated task. This is a functional test.

6) func_periodic1 creates three groups of periodic tasks, each group with different priorities. Each task makes some computation then sleeps till its next period, for 6000 times. Test is passed if no period is missed. This is a functional test.

7) func_periodic2 is a functional test we developed, and is conceived to be used in kernels that support primitives for periodic scheduling. It creates a task set with $U = 1$, using the UUniFast algorithm not to bias test’s results. Task set is scheduled, and test is passed if no period is missed.

8) func_periodic3 is a functional test we developed, and is conceived to be used in kernels that support primitives for periodic scheduling. It creates a task set with $U > 1$, using the UUniFast algorithm not to bias test’s results. Task set is scheduled, and test is passed if at least one period is missed.

9) func_pi checks whether priority inheritance support is present in the running kernel. This is a functional test.

10) func_preempt verifies that the system is able to ensure a correct preemption between tasks. It creates 26 tasks at different priority, each of them trying to acquire a mutex. Test is passed if all task are appropriately preempted in 1 loop. This is a functional test.

11) func_prio verifies priority ordered wakeup from waiting. It creates a number of tasks with increasing priorities, and a master task; each of them waits on the same mutex. When the master task releases its mutex, any other task can run. Test is passed if tasks wakeup happened in the correct priority order. This is a functional test.

12) func_sched verifies scheduler behaviour using a football analogy. Two kinds of tasks are created: defence tasks and offence tasks. Offence tasks are at lowest priority and tries to increment the value of a shared variable (the ball). Defence tasks have an higher priority and they should block offence tasks, in such a way that they never execute. In this way ball position should never change. The highest priority task (the referee) end the game after 50 seconds. Test is passed if at the end of the test the shared variable is zero. This is a functional test.

13) jitter_sched measures the maximum execution jitter obtained scheduling two different tasks. The execution jitter of a task is the largest difference between the execution times of any of its jobs [19]. The first task measures the time it takes to do a fixed amount of work; it is periodically interrupted by an higher priority task, that simply wakes up and goes back to sleep. Test is repeated 1000 times. This is a performance test.

14) latency_gettimeofday() is a performance test. It measures the time elapsed between two consecutive calls of the gettimeofday() primitive. Test is repeated a million of times, at bulks of ten thousand per time.

15) latency_timer one timer task and many busy tasks have to be scheduled. Busy tasks run at lower priority than timer task; they perform a busy wait, then yield the cpu. Timer task measures the time it takes to return from a nanosleep call. Test is repeated 10000 times, and is passed if the highest priority task latency is not increased by low priority tasks. This is a performance test.

16) latency_kill two tasks with different priority are to be scheduled. Lower priority task sends a kill signal to the higher priority task, that terminates. The test measures the latency between higher priority task start and termination. Test is repeated 10000 times, and we expect a latency under the tens of microseconds. This is a performance test.

17) latency_rdtsc is a performance test. It mea-
sures the average latency between two read of the TSC register, using the \texttt{rdtscll()} primitive. Test is repeated a million of times.

18) \textit{latency\_sched} is conceived to measure the latency involved in periodic scheduling in systems that do not support primitives for periodic scheduling. A task is executed, then goes to sleep for a certain amount of time; at the beginning of the new period the task is rescheduled. We measure the difference between expected start time and effective start time for the task. We expect this difference is under the tens of \(\mu s\) and, additionally, we expect the task does not miss any period. This is a performance test.

19) \textit{latency\_signal} schedules two tasks with the same priority. One task sends a signal, the other receives it. The test measures the time elapsed between sending and receiving the signal. Test is repeated a million of times. We expect a latency under the tens of \(\mu s\). This is a performance test.

20) \textit{stress\_pi} stresses the Priority Inheritance protocol. It creates 3 real-time tasks and 2 non real-time tasks, locking and unlocking a mutex 5000 times per period. We expect real-time tasks to make more progress on the CPU than non real-time tasks. This is a stress test.

5 Conclusions and future work

In this paper we have presented Lachesis, a unified and automated testsuite for Linux based real-time systems. Lachesis tries to meet the need for a software tool to test Linux and Linux-based systems real-time features, having the following qualities:

- supports tests on Linux, RTAI, Xenomai, PRE-EMPT\_RT and SCHED\_DEADLINE real-time features, through a standard test API
- provides a series of functional, performance and stress tests to ensure the functionality of the examined kernels
- provides a series of tests for periodic and deadline tasks
- is easy to use: each feature to be tested is associated to a script, which runs tests and logs the results for every testable system.
- it includes a set of bash scripts that helps to execute tests in the correct order and in the correct conditions.

Several real-time tests were ported to Lachesis from other testsuites, in a simple and straightforward way. In many cases there were no needs to change the code except to add some macro calls at the beginning and at the end of the test’s code.

Our extension to librttest API has made possible to develop some new tests for threads with fixed periods or deadlines. These tests are useful to value jitter and latency in periodic task scheduling. Similar tests can be developed in very short times.

Unfortunately, Lachesis is far from complete. First, its test coverage is very low. Second, tests included in Lachesis are somewhat general to be really useful in development. So Lachesis needs to expand its test coverage with more specific tests, and librttest needs to take into account more low level primitive, to make possible to develop more significant tests.

For this reasons, we believe it’s very important to integrate the testsuite rt-tests in Lachesis. As underlined previously, rt-tests is very specific in respect to Lachesis, and so it’s quite difficult to figure out how to extend these tests to other real-time nanokernels. We expect that a strong extension to librttest API is necessary to reach this goal.

Up to now Lachesis is tested and used only on x86 architecture. Given that we use only high-level kernel primitives, we are quite confident that the testsuite is easily portable on other architectures, with little or no effort. Recently we developed a porting of Xenomai 2.5.5.2 and RTAI 3.8 to a Marvell ARM9 board\footnote{ARM Marvell 88F6281, equipped with a Marvell Feroceon processor, clocked at 1.2 GHz, ARMv5TE instruction set.}, and we plan to use Lachesis to test the functionalities and the performances of these portings.

However, just the variety of Linux based real-time systems that Lachesis is able to test proves that it is portable and easy to use. We plan to exploit this qualities porting Lachesis to other systems, such IRMOS \cite{IRMOS}, SCHED\_SPORADIC \cite{SCHED_SPORADIC}, XtratuM \cite{XtratuM} and PartiKle \cite{PartiKle}, and to other architectures.

Beyond this, we plan to develop some kernel-space tests for real-time nanokernels and to build a system to parse and XML format results. Test results quality can be improved, also, detailing possible reasons behind a test failure.

Lachesis is actually under active development, and can be downloaded from bitbucket.org\footnote{https://bitbucket.org/whispererindarkness/lachesis}. \hfill \hfill
Acknowledgments

We would like to thank Andrea Baldini and Francesco Lucconi for their contributions in the development of the testsuite, and Massimo Lupi for his ideas and advices.

References


Lachesis: a test suite for Linux based real-time systems
Generic User-Level PCI Drivers

Hannes Weisbach, Björn Döbel, Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science, 01062 Dresden
{weisbach,doebel,adam}@tudos.org

Abstract

Linux has become a popular foundation for systems with real-time requirements such as industrial control applications. In order to run such workloads on Linux, the kernel needs to provide certain properties, such as low interrupt latencies. For this purpose, the kernel has been thoroughly examined, tuned, and verified. This examination includes all aspects of the kernel, including the device drivers necessary to run the system.

However, hardware may change and therefore require device driver updates or replacements. Such an update might require reevaluation of the whole kernel because of the tight integration of device drivers into the system and the manyfold ways of potential interactions. This approach is time-consuming and might require revalidation by a third party. To mitigate these costs, we propose to run device drivers in user-space applications. This allows to rely on the unmodified and already analyzed latency characteristics of the kernel when updating drivers, so that only the drivers themselves remain in the need of evaluation.

In this paper, we present the Device Driver Environment (DDE), which uses the UIO framework supplemented by some modifications, which allow running any recent PCI driver from the Linux kernel without modifications in user space. We report on our implementation, discuss problems related to DMA from user space and evaluate the achieved performance.

1 Introduction

Several advantages make the Linux kernel an attractive OS platform for developing systems with real-time capabilities in areas as diverse as industrial control, mobile computing, and factory automation: The kernel supports many popular computing platforms out of the box, which provides a low barrier starting to develop software for it. Being open source, it can be easily adapted to the target platform’s needs. A huge community of developers guarantees steady progress and fast response to problems.

Applying Linux in a real-time environment however leads to additional problems that need to be handled. We imagine a system where a computer controls a safety-critical industrial machine while in parallel providing non-real-time services. For example, it might provide work statistics through a web server running on the same machine. The RT_PREEMPT series of kernel patches [1] aims to provide the ability to do so. However, use of Linux in safety-critical systems would need additional audits and certifications to take place.

The web server in above example makes use of an in-kernel network device driver. Now, if the network driver needs to be upgraded for instance because of a security-related bugfix, the whole kernel or at least parts of it would need to be reaudited. These certifications incur high cost in terms of time and manual labor. They become prohibitively expensive when they need to be repeated every time a part of the system is upgraded.

Running device drivers in user space allows to circumvent recertification of the whole kernel by encapsulating the device driver in a user-level application. If, like in our example, the network is solely used by non-real-time work, it can be completely run outside the real-time domain and doesn’t need to be certified at all.

Linux already comes with UIO, a framework for writing device drivers in user space [5]. However,
these drivers still need to be rewritten from scratch using UIO. In this paper we propose an alternative
technique: Using UIO and other available kernel mechanisms, we implement a Device Driver Envi-
ronment (DDE) – a library providing a kernel-like interface at the user level. This approach allows for
reusing unmodified in-kernel drivers by simply wrapping them with the library and running them at the
user level.

In the following section, we introduce the general idea of the DDE and inspect the UIO framework with
respect to its support of a generic user-level driver layer. We then discuss our implementation of a DDE
for Linux in Section 3. Thereafter, we continue analyzing the special needs of Direct Memory Access
(DMA) from user space in Section 4 and present a solution that requires only minimal kernel support.
In Section 5 we evaluate our DDE implementation with an in-kernel e1000e network device driver run-
ning as user-space application.

2 User-Level Device Drivers
for Linux

Device drivers are known to be one of the single most
important sources of bugs in today’s systems [4].
Combined with the fact that most modern operat-
ing systems run device drivers inside their kernel, it
is not surprising that a majority of system crashes is
caused by device drivers – Swift and colleagues re-
ported in 2003 that 85% of Windows crashes may be
attributed to device driver faults [24].

One way to improve reliability in this area is to separate device drivers from the kernel and run them as independent user-level applications. Doing so isolates drivers from each other and the remaining components and increases the chance that a faulting driver does not take down the rest of the system. Properly isolated drivers may be restarted after a crash as it is done in Minix3 [13]. Performance degra-
dation resulting from moving drivers out of the kernel
into user space is often considered a major disadvan-
tage of this approach. However, it has been proven
that user-level device drivers may achieve the same
performance as if run in the kernel [16]. Further
research showed that existing device drivers can be
automatically retrofitted to run most of their critical
code in user space and only keep performance-critical
paths within the kernel [11].

Our ultimate goal is to provide a Device Driver Environ-
ment, a common runtime library that can be
linked against arbitrary in-kernel device drivers in
order to run them as user-level applications without
modification. In this section we give an overview of
the DDE approach and analyze Linux’ UIO frame-
work regarding its capabilities of supporting generic
user-level device drivers.

2.1 The DDE Approach

Our approach for reusing in-kernel device drivers in
user space is depicted in Figure 1. The source code of
an unmodified native Linux device driver is linked
against a wrapper library, the Device Driver Envi-
ronment. The wrapper provides all functions the
driver expects to be implemented originally by the
Linux kernel. The DDE reimplements these func-
tions solely using mechanisms provided by a device
driver abstraction layer, called DDEKit.

FIGURE 1: DDE Architecture

Only DDE knows about the intricate require-
ments of guest drivers. In turn, the DDEKit provides
abstract driver-related functionality (device discov-
ery and management of device resources, synchro-
nization, threading, etc.) and implements it using
functionality from the underlying host OS. Split-
ting development into these two layers allows to use
a DDEKit for a certain host platform in connec-
tion with different guest DDE implementations as
well as reuse the same guest DDE on a variety of
hosts. This layering has allowed for implementations
of DDE/DDEKit for different guest drivers (Linux,
FreeBSD [10]) as well as different host platforms
(Linux, L4/Fiasco [12], Genode [15], Minix3 [25],
GNU/HURD [7]).

In this paper we focus on implementing a DDE
for Linux PCI device drivers on top of the Linux
kernel. To achieve this goal, it is necessary to under-
stand the facilities at hand to perform device driver-
related tasks from user space. The User-level IO
framework (UIO) appears to be a good starting point
for this.
2.2 UIO Overview

The Linux user-level IO framework (UIO) is an extension to the kernel that allows user-level drivers to access device resources through a file interface and is depicted in Figure 2. The interfacing is performed by the generic \texttt{uio\_core}. In addition to that, UIO relies on a tiny device-specific driver stub, labelled \texttt{uio\_dev} in the figure. During startup, this stub obtains information about the device’s I/O resources and when encountering an interrupt takes care of checking whether the interrupt was raised by the device and handles the device-specific way of acknowledging the interrupt.

![UIO components and their interaction](image)

A user-level driver obtains access to the target device’s resources through a /dev/uioXXX device file. Reading the device returns the number of (interrupt) events that occurred since the last read. Device I/O memory can be accessed by \texttt{mmap}’ing the device. UIO neither supports x86 I/O ports\footnote{Actually, UIO does not need to support I/O ports, because these can be directly accessed by the user application if it is given the right I/O permissions.} nor direct memory access (DMA).

UIO for Generic User-Level Drivers

Our goal is to implement a DDE that allows generic PCI device drivers to be run in user space. This does not fit well with UIO’s dependence on a device-specific stub driver. Unfortunately, there is no generic way to move acknowledgment of an interrupt out of the kernel. Instead, this is often highly device-specific and requires running in kernel mode.

As an exception, the situation improves with PCI devices that adhere to more recent versions of the PCI specification\footnote{\cite{21}} (v2.3 or later). These devices allow generic detection of whether an interrupt is asserted using an interrupt state bit in the PCI config space. Furthermore, it is possible to generically disable interrupt delivery using an interrupt disable bit. This enables the implementation of a generic UIO PCI driver and removes the requirement of a device-specific driver stub.

The lack of support for user-level DMA is another issue that needs to be resolved in order to support arbitrary user-level PCI drivers. In the following sections we present the details of our implementation of a DDE for Linux.

3 A DDE For Linux

As described in Section 2.1, the user-space driver environment consists of two parts: a host-specific DDEKit providing a generic device driver abstraction and a guest-specific DDE that solely relies on the functionality provided by the DDEKit. For our implementation, we can build upon the already existing Linux-specific DDE for the L4/Fiasco microkernel\footnote{\cite{12}}. In addition to that we need to implement a DDEKit for Linux as a host, which we describe in this section.

3.1 Anatomy of a DDEKit

The DDEKit’s task is to provide a generic interface that suits the needs of guest device driver environments. To come up with this interface, we analyzed network, block, and character device drivers in two different kernels (Linux and FreeBSD)\footnote{\cite{10}}, resulting in a list of mechanisms all these drivers and their respective environments rely on.

The most important task of a device driver is managing I/O resources. Therefore, a driver abstraction layer needs to provide means to access and handle interrupts, memory-mapped I/O, and I/O ports. As most of the drivers we are concerned with are PCI device drivers, DDEKit also needs to provide ways to enumerate the system’s PCI devices or at least discover the resources that are attached to a device. Additionally, means for dynamic memory management are crucial when implementing anything but the most simple device driver.

While most device drivers operate single-threaded, threading plays an important role in DDE implementations, because threads can be used to implement tasks such as interrupt handling, Linux Soft-IRQs, as well as deferred activities (work queues). The existence of threading implies that synchroniza-
tion mechanisms such as locks, semaphores, and even condition variables need to be present.

Furthermore, a lot of drivers need a notion of time, which Linux drivers usually obtain by looking at the magic jiffies variable. Hence, DDEKit needs to support this. Apart from these features, in order to be useful, the DDEKit also provides means for printing messages and a link-time mechanism for implementing prioritized init-calls, that is functions that are automatically run during application startup before the program’s main function is executed.

3.2 I/O Ports and Memory

In order to drive PCI devices and handle their resources, DDEKit needs means to discover devices at runtime. This is implemented using libpci [18], which allows scanning the PCI bus from user space. The located devices are then attached to a virtual PCI bus implemented by DDE. At runtime, any calls by the driver to the PCI subsystem use this virtual bus to perform their work.

After the virtual PCI bus is filled with the devices to be driven, information about the provided resources is obtained from /sys/bus/pci/devices/.../resource. Access to I/O ports is later granted by first checking whether the ports requested by the driver match the ones specified by the resource file, and thereafter granting the process port access using the ioperm system call. I/O memory resources are also validated and then made accessible by mmaping the respective sysfs resource files.

3.3 Interrupt Handling

For managing interrupts, DDEKit/Linux makes use of the UIO interrupt handling mechanism, which supports generic interrupt handling through the uio_pci_generic module for all PCI devices supporting the PCI specification v2.3 or higher.

Once the driver requests an IRQ for a device, DDEKit locates the generic UIO driver’s sysfs node (/sys/bus/pci/drivers/.../new_id). It then writes the PCI device’s device and vendor IDs into this file and thereby makes uio_pci_generic become responsible for handling this device’s interrupts.

Thereafter, a new interrupt handler thread is started. This thread performs a blocking read on the UIO file that was generated when attaching uio_pci_generic to the device. Whenever the read returns, at least one interrupt event has occurred and the handler function registered by the driver is executed.

The interrupt handler thread is the only one polling the UIO device file for interrupts. After successful return from the blocking read, the sysfs node for the device’s PCI config space (/sys/class/uio/.../config) is written to disable IRQs while handling the interrupts. In order to avoid interrupt storms in the kernel while the user-level driver is executing its handler, the disabled interrupt is only turned on right before the interrupt thread becomes ready to wait for the next interrupt by reading the UIO device.

3.4 Threads and Synchronization

Threads are a fundamental building block of a DDE, because drivers may use a wide range of facilities that might be executed in parallel: soft-IRQs, kernel threads, and work queues are implemented by spawning a dedicated thread for each such object. Furthermore, threads are used for implementing interrupts as discussed in Section 3.3. However, not all of these activities are actually allowed to execute in parallel. Therefore, means for (blocking) synchronization are needed.

As DDEKit/Linux is implemented to run in Linux user space, we can make use of the full range functions provided by the libpthread API to implement threading as well as synchronization.

3.5 Timing

Linux device drivers use timing in two flavors: first, the jiffies counter is incremented with every clock tick. DDEKit/Linux emulates jiffies as a global variable. During startup, a dedicated jiffies thread is started that uses the libc’s nanosleep to sleep for a while and thereafter adapt the jiffies counter accordingly. For the drivers we experimented with so far, it has proven sufficient to not tick with HZ frequency as the Linux kernel would, but instead only update the jiffies counter every 10th HZ tick. This might be adapted once a driver needs a finer granularity. Furthermore, as device drivers run as independent instances in user space, this can be configured for every device driver separately according to its needs and the jiffies counting overhead can even be completely removed for drivers that don’t need this time source.
The second way Linux drivers use timing is through the `add_timer` group of functions that allows to program deferred events. DDEKit/Linux provides an implementation by spawning a dedicated timer thread for every driver instance. This thread manages a list of pending timers and uses a semaphore to block with a timeout until the next timer occurrence should be triggered. If the blocking semaphore acquisition returns with a timeout, the next pending timer needs to be handled by executing the handler function. Otherwise, an external thread has modified the timer list by either adding or removing a timer. In this case the timer thread recalculates the time to sleep until the next trigger and goes back to sleep.

3.6 Memory Management

Running in user space means that DDEKit/Linux may use LibC’s `malloc` and `free` functions for internal memory management needs. However, this does not suffice for implementing Linux’ memory management functions. Linux’ `kmalloc` is internally already implemented using SLABs or one of their equivalents. Our implementation currently provides a specific SLAB implementation in DDEKit, but we plan to use Linux’ original memory allocator in the future and only back it with page-granularity memory allocations provided from DDEKit.

Additionally, Linux drivers may use the group of `get_free_pages` functions to allocate memory with page granularity. DDEKit/Linux supports page granularity allocations through a function that uses `mmap` in order to allocate page-aligned memory.

A remaining problem is that drivers commonly acquire DMA-able memory in order to allow high amounts of data to be copied without CPU interaction. This is impossible by solely relying on user-level primitives. This means that an implementation of DMA allocation functions such as `dma_alloc_coherent` requires additional thought. We go on to discuss our solution to this problem in the following section.

4 Attacking the DMA Problem

In order for DMA to or from a memory region to work properly, the region needs to meet three criteria:

1. The region’s physical address needs to be available as DMA does not use virtual addresses.
2. It needs to be physically contiguous so that no virtual-to-physical address translations need to be done during the DMA transfer.
3. It needs to be pinned, that is the region or parts of it must not be swapped out during the DMA transfer.

None of these criteria are met by user-level memory allocation routines such as `malloc`, `posix_memalign` or `mmap`, because they work on purely virtual addresses and the underlying kernel is free to map those pages anywhere it wants.

As it is necessary to get kernel support for handling DMA, we implemented a small kernel module providing an interface to the in-kernel DMA API. The module supports two modes: `copy-mode` provides a simple translation layer between user and kernel pages for DMA and `zero-copy mode` facilitates an IOMMU to improve DMA performance.

4.1 Copy-DMA

Our kernel module for supporting DMA from user space closely collaborates with the `uio_core` as shown in Figure 3. The `uio_dma` module is notified by the `uio_core` when a device is bound to it and creates an additional device node `/dev/uio-dma` which user-level drivers can use to obtain DMA-able memory for a specific device.

![Figure 3: Introducing the uio_dma module](attachment:image.png)

The notification is necessary so that the `uio_dma` module has access to the respective UIO PCI device data structure.
A naïve idea would be to simply implement a device driver that allows allocating DMA memory from the kernel and then use it from user space. However, on many platforms it is possible to use any physical memory for DMA. Therefore, many drivers do not explicitly allocate DMA memory upfront, but instead simply start DMA from arbitrary memory regions, even from their stack. This means the DMA allocator driver would have to provide all dynamic memory allocations for the user space driver. Not only would this circumvent the convenience of managing user space memory using libC’s malloc and free, but it would also decrease the possibility of using GDB with a user space driver, because such kernel memory would not be ptraceable.

DDE’s implementation of dma_alloc_coherent performs an mmap on the uio-dma device which in turn allocates DMA-able memory in the kernel and establishes a mapping to user space so that upon return from the system call the driver can use this memory area for DMA.

For the map_single family of functions an ioctl on the uio-dma device is used to send a virtual user address and the DMA direction to the kernel module. The system call returns the physical address the driver can then use to initiate DMA.

If upon a DMA_MAP ioctl the direction indicates that data shall be sent from user space, the kernel module allocates a DMA-able kernel buffer and copies the user data into this buffer before returning the DMA buffer’s physical address. If DMA shall be done from the device into a user buffer, the ioctl only allocates a DMA buffer in the kernel and delays copying data from the DMA buffer to the user buffer until the buffer is unmapped using dma_unmap_single. It is safe to do so, because only after this function call the DMA can safely be assumed to be finished and therefore the user-level driver should not touch the buffer beforehand anyway.

4.2 DMA With Fewer Copies

While the copy-DMA method works without any further support than the one that is already present within the kernel, more recent hardware featuring an IOMMU can be used to get rid of the copying steps between user and kernel buffers.

Copy-DMA uses different kernel- and user-level buffers because it needs to ensure that the kernel buffers that are effectively used for the DMA operation are in fact physically contiguous, which cannot be guaranteed for the user-level buffers.

Using an IOMMU comes to the rescue here. In this case we can use an arbitrary buffer that is virtually contiguous (which includes every buffer allocated using malloc). The uio_dma module upon encountering the DMA ioctl then only needs to run through the list of pages forming the buffer and perform an equally contiguous mapping into the device’s IOVA space. Thereafter, the user-level driver can use the IOVA address returned from the ioctl call and program DMA without needing to care about physical contiguity.

A minor intricacy arises because of the fact that user-level buffers do not always start and end at a page boundary. This means that multiple DMA buffers may share the same page, so that upon unmapping one of the buffers, the uio_dma module cannot safely remove the IOVA mapping as other DMA buffers may still contain the same page. Therefore, uio_dma uses reference counting to detect when a page may really be unmapped.

5 Case Studies

There are three interesting questions concerning user-level device drivers:

1. Does running the driver in user space modify the real-time capabilities of a PREEMPT_RT kernel?
2. How does the user space driver’s performance compare with an in-kernel driver?
3. Which benefits can be gained by running a driver in user space and using existing profiling and debugging tools?

In this section we try to answer these questions using a real-world example. We downloaded the Linux
e1000e network interface driver from the Intel website [6] and compiled it to run in user space.

For all our experiments we used a quad-core Intel Core i7 running at 2.8 GHz with 2 GB of RAM. The operating system was a Linux 3.0.1-rt11 kernel with the PREEMPT_RT option switched on. We tested the e1000e driver with an Intel 82578DC Gb ethernet card.

5.1 Real-Time Operation

To evaluate the influence of running PCI drivers in user space on the system’s real-time behavior, we used the cyclic_test utility provided by OSADL [20]. Figure 5 shows the maximum latencies for several scenarios we tested.

Each group has four bars corresponding to threads running on the 4 CPUs in our test machine. The group labelled no_load shows the latencies for running cyclic_test on the idle system running with idle=poll to mitigate power management effects. For the group labelled hi_load we set each CPU’s load to 100% and reran cyclic_test. Thereafter, we added network load to the system by running the IPerf UDP benchmark [8] between the test machine and a remote PC. The groups with labels *e1000e show the latency for using network through the in-kernel e1000e driver. The groups labelled *user give latencies obtained for running the experiment with the e1000e driver in user space using DDE.

Additionally, we tried to figure out whether turning the machine’s IOMMU on or off makes a difference and therefore reran our experiments with the IOMMU turned on. The results for these experiments are shown in Figure 6.

5.2 UDP/TCP Performance

To evaluate the performance of user-level DDE drivers, we linked our user-level e1000e driver to the lwIP stack [9] and then ran a UDP and TCP throughput benchmark while connected to an external computer. For comparison, we also ran the same benchmark using the in-kernel device driver and the built-in Linux TCP stack.

Figure 7 shows the average and maximum throughputs achieved in these experiments. For UDP it is notable, that even though the network link is a 1Gb NIC, IPerf was only able to saturate 800 MBit/s. Furthermore, user-level and kernel stacks perform equally well. However, the CPU utilization for the user-level driver is higher: the kernel stack ran at about 50% utilization, while the user stack consumed 80%.
FIGURE 7: IPerf throughput results

TCP performance is unfortunately much worse for the user-level TCP stack than with the in-kernel one. We are still investigating these issues and right now attribute this to problems with the lwIP stack.

5.3 Testing, Debugging, and Profiling

So far we showed that user-level device drivers allow for increased isolation at acceptable speed while not influencing the system’s real-time capabilities. Another advantage of running them as user applications is the availability of debugging and profiling tools that ease driver development. In this section we introduce two examples where user-level tools could be applied to kernel code and helped us find problems in our implementation of DDE.

Debugging DDE

While working on the user-level e1000e device driver, we experienced hangs in TCP connectivity and started debugging them. With the help of the GDB debugger we were able to figure out that the problem occurred when the driver was in NAPI polling mode and ran out of its network processing budget.

In the Linux kernel, the driver at this point voluntarily reschedules, giving other kernel activity the chance to run. This was improperly implemented within the DDE as it simply returned from the Soft-IRQ handler. In our case however, it would have been necessary to raise the Soft-IRQ again. This did not happen and so the driver went to sleep until it got woken up by the next interrupt.

Profiling DDE

When we initially ran the e1000e driver in user space, performance was by far not as convincing as in the experiments described in Section 5.2.

Using Valgrind’s [19] Callgrind profiler, we were able to investigate where the performance went. We were caught by surprise by the result: DDE manages a list of virtual-to-physical mappings for all allocated memory. This list is used to implement the virt_to_phys lookup mechanism. This is implemented as a linked list and the assumption was that this would suffice, because there would never be many mappings stored in this list and calls to virt_to_phys would take place rather less frequently.

Callgrind’s output however told us that this function accounted for a huge amount of execution time. With this knowledge we were able to take a closer look at what regions were registered in this list and found out that in many cases, we did not need to store this information at all, thereby reducing the amount of time spent searching the virt-phys mappings.

6 Related Work

Our work relates to the problem of reusing existing device drivers when designing a new operating system. LeVasseur et al. proposed to use per-driver virtual machines to reuse and isolate device drivers [17]. Poess showed that the DDE approach also works for binary device drivers [22], which is not yet implemented in our system. Friebel implemented a DDE for FreeBSD device drivers which uses the split DDEKit/DDE architecture [10]. Building upon this, it should now also be possible to use FreeBSD device drivers within Linux user space. Boyd-Wickizer’s SUD framework [2] also allows running Linux drivers in user space but focusses more on security and isolation and instead of the real-time capabilities of the system.

Our work is also related to Schneider’s device driver validation mechanism [23]. By wrapping drivers in user-level applications, we can use the system’s native analysis and profiling tools in order to observe driver behavior and identify security violations. The RUMP framework for NetBSD has similar goals as our work and allows debugging and developing device drivers for NetBSD in user space [14]. However, it is not intended to be used for actually running drivers at user space in production systems.

Chipounov proposed to perform heavyweight
instruction-level tracing and symbolic execution in order to generate device-specific code that can be dropped into existing per-OS device driver skeletons [3]. While this approach eases device driver reuse, applying it to a real-time kernel has the same drawbacks as native in-kernel drivers in that they still need to be revalidated every time an update is applied.

7 Conclusion

In this paper we presented a Device Driver Environment that allows executing generic Linux in-kernel PCI drivers as user-level applications on top of Linux. This is achieved by implementing the DDE as a wrapper library implementing the facilities expected by in-kernel drivers at user space using off-the-shelf kernel mechanisms such as UIO and sysfs. With the help of a small kernel module our framework also supports DMA from user space.

Using this framework, we were able to run the widely used e1000e network interface driver in user space on a PREEMPT_RT kernel. Experiments using cyclic_test showed that the real-time latencies of the system were not influenced by the fact that the driver was running from user space. Furthermore, it was possible to use common Linux program analysis tools such as the GDB debugger and Valgrind to profile and debug drivers.

The DDEKit for Linux is available for download at http://os.inf.tu-dresden.de/ddekit/.

Acknowledgments

We’d like to thank several people whose hard work within the recent years has made design and implementation of the Device Driver Environment possible. Thank you, Christian Helmuth, Thomas Friebel, and Dirk Vogt. Carsten Weinhold provided valuable hints on improving this paper.

This work was partially supported by the German Research Association (DFG) within the Special Purpose Program 1500, project title ASTEROID.

References


COMEDI and UIO Drivers for PCI Multifunction
Data Acquisition and Generic I/O Cards
and Their QEMU Virtual Hardware Equivalents

Pavel Příša
Czech Technical University in Prague, Department of Control Engineering
Karlov náměstí 13, 121 35 Praha 2, Czech Republic
pisa@cmp.felk.cvut.cz

Rostislav Lisový
Czech Technical University in Prague, Faculty of Electrical Engineering
Karlov náměstí 13, 121 35 Praha 2, Czech Republic
lisovy@gmail.com

Abstract

The article describes implementation of UIO and Comedi drivers for Humusoft MF624 and MF614 data acquisition cards. Basic functions (D/A, A/D converters, digital inputs/outputs) of Humusoft MF624 card were implemented into the Qemu emulator as well which enable to experiment with drivers implementation without physical access to the cards and risk of data lost when drivers are developed and tested on same primary Linux kernel instance. The article can help newcomers in the area to gain knowledge required to implement support for other similar cards and hardware emulation of these cards. The matching real and virtual setup can be used in operating system courses for practical introduction to simple drivers implementation and helps with understanding internal computation world with real world computers interfacing.

1 Introduction

When teaching development of Linux drivers one of the approaches is to explain kernel API and programming paradigms by creating driver which does not require any special hardware – e.g. character driver which returns upper case ASCII text when receiving lower case. Although this approach can be useful, the issues associated with dealing with hardware should be practised as well.

The approach we took in this work eliminates the need of physical access to hardware whereas it provides full feature set of PCI device in form of virtual hardware. This was possible by implementing virtual PCI device into Qemu emulator.

The main reason of choosing DAQ cards for this project was easy interfacing from programmer’s point of view and straightforward testing of proper function of the driver. This can be very helpful for beginners who are not familiar with hardware related topics.

2 Humusoft MF614, MF624

Humusoft MF614 and MF624 are data acquisition (DAQ) cards. Both of these cards use PCI interface to connect to the computer. The main features this cards provide are digital inputs, digital outputs, ADCs, DACs, timers, encoder inputs. Humusoft MF614 is predecessor of MF624 – available functions are quite similar. The main difference is in driver programming – MF614 has only 8-bit wide registers, whereas MF624 has ones 16- or 32-bit wide.
MF624 is available for purchase on manufacturer’s web page. MF614 is no more produced.

3 UIO Driver

Each UIO driver consists of two parts – small kernel module (the need for it is mostly because of device-specific interrupt handling/disabling) and user-space driver logic (as shown in figure 1). The main advantage of this approach is that the most of the development happens in user-space, thus during prototyping the driver (or when using a bad one) the integrity and stability of the kernel will not be disrupted.

![UIO driver structure](image)

**FIGURE 1:** UIO driver structure

Driver `uio_pci_generic`

When dealing with any device compliant to PCI 2.3, it is also possible to use `uio_pci_generic` driver in kernel instead of programming a specific one. This driver makes all memory regions of the device available to user-space.

Binding to the device is done by writing Vendor and Device ID into `/sys/bus/pci/drivers/uio_pci_generic/new_id` file.

Interrupt handler uses Interrupt Disable bit in the PCI command register and Interrupt Status bit in the PCI status register. Because neither of MF614 or MF624 is PCI 2.3 compliant it is not possible to use this driver for them.

Implementing the Kernel Part

In case of writing UIO driver for PCI device, initialization function of the module registers `struct pci_driver` in standard way\(^1\), where the probe function handles initialization of UIO-related structures. The main structure holding all data of particular UIO driver is `struct uio_info`. Its simple initialization (including registration) is shown below:

\begin{verbatim}
1 | /* struct pci_dev *dev */
2 | struct uio_info *info;
3 | info = kzalloc(sizeof(struct uio_info),
4 | GFP_KERNEL);
5 |
6 | info->name = "mf624";
7 | info->version = "0.0.1";
8 |
9 | info->mem[0].name = "PCI chipset";
10 | info->mem[0].addr =
11 | pci_resource_start(dev, 0);
12 | info->mem[0].size =
13 | pci_resource_len(dev, 0);
14 | info->mem[0].memtype = UIO_MEM_PHYS;
15 | info->mem[0].internal_addr =
16 | pci_ioremap_bar(dev, 0);
17 |
18 | info->port[0].name =
19 | "Board programming registers";
20 | info->port[0].porttype = UIO_PORT_X86;
21 | info->port[0].start =
22 | pci_resource_start(dev, 1);
23 | info->port[0].size =
24 | pci_resource_len(dev, 1);
25 |
26 | uio_register_device(&dev->dev, info);
27 | pci_set_drvdata(dev, info);
\end{verbatim}

Structure `uio_mem` is used for enabling memory-mapped I/O regions, whereas structure `uio_port` is used for I/O ports (for each of these structures there is statically allocated array with a size of 5 elements).

Interface to User-space

Communication with kernel part of the UIO driver is possible through `/dev/uioX` file (where `X` is the number of instance of a driver). There are several syscalls possible to be used when interfacing with this file:

- `open()` opens the device, returns file descriptor used for another syscalls.
- `read()` blocks until an interrupt occurs (the value read is number of interrupts seen by the device).
- `mmap()` is used to map memory of the device to user-space. The offset value passed to `mmap()` determines the memory area of a device to map – for `n`-th area offset should be `n*sysconf(_SC_PAGESIZE)`.

\(^1\)For more information about PCI driver development see [1] available online at https://lwn.net/Kernel/LDD3/
irqcontrol() is used for enabling (called with parameter set to (int) 1) or disabling ((int) 0) interrupts.

It is possible to define your own mmap(), open(), release() functions as an option. When there is need to use irqcontrol(), it is necessary to implement this function per device.

Information related to a particular driver instance can be found in /sys/class/uio/uioX directory. Most of the files are read-only. The subdirectory maps contains information about MMIO regions mapped by the driver, subdirectory portio is for I/O port regions.

When using UIO and mmap() with MF624 card (which has 32 or 128 bytes long memory regions) there is an issue with the return value of this syscall – the pointer to the memory is page-size-aligned, so it is necessary to add low bits of physical address (page offset) of each memory region to it. Physical address can be obtained from addr file located in /sys/class/uio/uioX/maps/mapX. Region offset is equal to addr & (sysconf(SC_PAGESIZE) - 1).

4 Comedi Driver

UIO driver is a versatile solution available mainly for uncommon devices. In our case of using DAQ card a special subsystem in Linux kernel designated for DAQ card drivers can be used. It is called Comedi (Linux control and measurement device interface). It provides library functions for user- and kernel-space making development and usage of DAQ devices easier. It consists of three different parts.

Comedi is a part of Linux kernel. It consist of individual device drivers including Comedi driver providing basic set of functions used by device drivers.

Comedilib is a user-space library providing unified interface for another user-space application to devices supported by Comedi.

Kcomedilib is also a part of Linux kernel. It provides the same API as Comedilib, whereas this is used for real-time applications.

Implementing the Driver

Each Comedi driver should be registered to the list of active Comedi drivers. This is done by invoking comple_driver_register() function. The only parameter passed to this function is pointer to struct comple_driver structure. The most important fields of this structure are:

```
const char *driver_name; /* "my_driver" */
struct module *module; /* THIS_MODULE */
int (*attach) (struct comple_device *,
               struct comple_devconfig *);
int (*detach) (struct comple_device *);
```

Unlike the UIO or generic PCI driver, the main initialization function is not probe() (of struct pci_driver) but attach() (of struct comple_driver) which is invoked by Comedi subsystem.

The attach() function is responsible not only for common PCI device initialization but also for initialization of struct comple_device (which is accessible through a pointer passed to attach() function). The most important step is to allocate and initialize each subdevice (in Comedis nomenclature subdevice represents one particular function of the device – e.g. ADC, digital out, etc.) of the DAQ card. Allocation is done by Comedi function alloc_subdevices(struct comple_device *dev, unsigned int num_subdev), each struct comple_subdevice is then accessible in array called subdevices which is part of struct comple_device. Example of initialization of subdevice representing ADC:

```
1 | s = dev->subdevices + 0;
2 | s->type = COMEDI_SUBD_AI;
3 | s->subdev_flags = SDF_READABLE | SDF_GROUND;
4 | s->n_chan = 8;
5 | s->maxdata = (1 << 14) - 1;
6 | s->range_table = &range_bipolar10;
7 | s->len_chanlist = 8;
8 | s->insn_read = mf624_ai_rinsn;
9 | s->insn_config = mf624_ai_cfg;
```

Interface to User-space

After successful compilation and loading of particular Comedi driver, there should be /dev/comediX (where X is number of instance of a driver) file. For communication with this file Comedi library functions are used. For opening device – comple_open(), for reading/writing ADCs/DACs – comple_data_read(), comple_data_write() and for reading/writing digital inputs/outputs – comple_dio_read(), comple_dio_write().
There are already applications using Comedi API\(^2\) – thus in some cases there is no need for implementing user-space application from scratch.

5 Qemu Virtual Hardware

Qemu is an open-source processor emulator. Unlike common virtualization solutions it is able of emulating x86, x86-64, ARM and other widespread processor architectures. For the purposes of this work it was used for implementing virtual Humusoft MF624 DAQ card.

Implementation of Virtual PCI device

When creating new virtual device in Qemu, main hook into Qemu device infrastructure is done by invoking `device_init()` with parameter of pointer to initialization function with prototype of static `void (*)(void)` for registering new PCI device, it is necessary to call `pci_dev_register()` passing parameter of pointer to `PCIDeviceInfo`. The most important fields of this Qemu-specific data type are pointers to `init` and `exit` functions with prototype of `int (*)(PCIDevice *)`.

The PCI device specific initialization consists of:

- Initializing configuration space of PCI device – e.g. setting Vendor and Device IDs, device class, interrupt pin, etc.
- Registration of I/O memory used by the device.
- Creating a function (called when device gets allocated memory from virtual PCI controller) for mapping of physical memory to particular `BARs` (Base Address Registers) of the PCI device.

The very basic (non-compilable) example:

```c
/* File: mf624.c */

#define BAR0_SIZE 0x10000
#define BAR0_MEM_TABLE_INDEX 0

enum CPU_READ_MEMORY_FUNC { mf624_BAR0_read, mf624_BAR0_write, mf624_BAR0_map, mf624_BAR0_exit };

void mf624_map(PCIDevice *pci_dev, int region, pcibus_t addr, pcibus_t sz, int tp)
{
    mf624_state_t *s = DO_UPCAST(mf624_state_t, dev, pci_dev);
    cpu_register_physical_memory(addr + 0x0, BAR0_size, s ->BAR0_mem_table_index);
}

static int pci_mf624_init(PCIDevice *pci_dev)
{
    mf624_state_t *s = DO_UPCAST(mf624_state_t, dev, pci_dev); /* i.e. container_of() */
    uint8_t *pci_conf;
    pci_conf = s->dev.config;
    pci_config_set_vendor_id(pci_conf, PCI_VENDOR_ID_HUMUSOFT);
    /* ... */
    pci_conf[PCI_INTERRUPT_PIN] = 0x1;
    s->BAR0_mem_table_index = cpu_register_io_memory(mf624_BAR0_read, mf624_BAR0_write,
            s, DEVICE_NATIVE_ENDIAN); /* returns unsigned int */
    pci_register_bar(ks->dev, 0, BAR0_size, PCI_BASE_ADDRESS_SPACE_MEMORY, mf624_map);
    return 0;
}

static void reg_dev(void)
{
    pci_qdev_register(&mf624_info);
}

static void mf624_BAR0_read(PCIDevice *pci_dev, uint8_t *addr, uint8_t *data, uint32_t len)
{
    /* read function */
}

static void mf624_BAR0_write(PCIDevice *pci_dev, uint8_t *addr, uint8_t *data, uint32_t len)
{
    /* write function */
}

static void mf624_BAR0_map(PCIDevice *pci_dev, uint8_t *addr, uint8_t *data, uint32_t len)
{
    /* map function */
}

static void mf624_BAR0_exit(PCIDevice *pci_dev)
{
    /* exit function */
}

/*
 * QEMU specific data type
 */

struct PCIDeviceInfo
{
    const char qdev.name[] = "mf624";
    uint8_t qdev.size = sizeof(mf624_state_t);
    int init = mf624_init, exit = mf624_exit;
};

static void reg_dev(void)
{
    pci_qdev_register(&mf624_info);
}

device_init(reg_dev)
```

\(^2\)For basic list of available applications see [http://www.comedi.org/applications.html](http://www.comedi.org/applications.html)
Usage of Virtual MF624

When running any guest operating system in Qemu (with support for MF624 activated) the virtual MF624 device is available in the same way as if it was real hardware – there are no issues with interfacing between guest operating system and virtual device. Interfacing between virtual hardware and real world is handled by TCP/IP connection from MF624 part in Qemu to host operating system. It is used for reading/setting output/input values (as shown in figure 2).

The most fundamental way of communication through this channel is by using telnet application. Example of real communication:

```
$ telnet localhost 55555
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
DA1=9.998779
DOUT=255.000000
DOUT=0.000000
DA1=5.000000
^]  
telnet> Connection closed.
```

As a much more easier way of interfacing, there is also graphical application created just for purposes of communication with virtual MF624 card (see figure 3). It was created using Qt4 graphical toolkit.

![FIGURE 2: Qemu implementing virtual MF624 device](image)

![FIGURE 3: Graphical application used for interfacing between virtual MF624 and real world](image)

6 Conclusion

The outcome of this work creates basic integrated tool for teaching PCI driver development (mostly) for GNU/Linux operating system. Its main advantage is the possibility to train driver development on real hardware without the necessity of having an expensive DAQ device. The other advantage is a safe environment for driver prototyping – where no mistake can damage host operating system.

All the information (including source code) related to topic covered in this article are publicly available on web page rttime.felk.cvut.cz/hw/index.php/Humusoft_MF6xx

References

A Framework for Component-Based Real-Time Control Applications

Stefan Richter
ABB Corporate Research, Industrial Software Systems
Segelhofstr. 1K, Baden-Dättwil, Switzerland
stefan.richter@ch.abb.com

Michael Wahler
ABB Corporate Research, Industrial Software Systems
Segelhofstr. 1K, Baden-Dättwil, Switzerland
michael.wahler@ch.abb.com

Atul Kumar
ABB Corporate Research, Industrial Software Systems
Whitefield Road, Bangalore, India
atulkumar.d@in.abb.com

Abstract

State-of-the-art real-time control systems execute multiple concurrent control applications using operating system mechanisms such as processes, mutexes, or message queues. Such mechanisms leave a high degree of freedom to developers but are often hard to deal with: they incur runtime overhead, e.g., context switches between threads, and often require tedious and costly fine-tuning, e.g., of process and thread priorities. Reuse is often made more difficult by the tight coupling of software to a given hardware or other software.

In this paper, we present a software architecture and execution framework for cyclic control applications that simplifies the construction of real-time control systems while increasing predictability and reducing runtime overhead and coupling. We present the concepts of this framework as well as implementation details of our RTLinux-based prototype.

1 Introduction

Control systems interact with the physical world through sensors and actuators. Since physical processes typically do not wait for the result of some computation, control systems must meet given deadlines and are therefore real-time systems. Traditionally, there was a separate controller for each control application. These controllers are typically running at control cycles of one millisecond, i.e., they acquire new measurements, process them, and take actions once every millisecond. With the continuous improvement of computer technologies and the introduction of digital communication standards such as IEC 61850 [3], there have been strong trends to combine several such logical controllers into one physical device to reduce cost. Still, a certain degree of independence is required to make sure that a faulty logical controller does not affect other controllers on the same physical device.

Executing multiple control applications on the same controller has significantly increased the complexity of the controller’s software. In particular, the concurrent execution of multiple applications requires synchronization between different processes.
The fine-tuning of such synchronization to satisfy all runtime constraints on a given platform can be tedious and error-prone. Furthermore, it is difficult to reuse a given set of processes on a different platform. At runtime, the concurrent execution of multiple processes causes context switches, which may be expensive in comparison to the program logic.

In this paper, we present a component-based software architecture and execution framework for a large class of industrial control systems. These systems comprise one or several periodic real-time tasks (as defined by Liu in Section 3.3.1 [8]). Using a small set of abstraction mechanisms, our approach simplifies system construction and makes large parts of an engineered control system reusable across different platforms.

This introductory section finishes with a motivating example and an overview of related issues. Our component framework and its abstraction mechanisms are presented in Section 2. We give implementation details in Section 3 along with several measurements of the framework’s performance in Section 4. We conclude the paper with a summary, discussion, and outlook in Section 5.

1.1 Motivating Example

Throughout this paper we will use an example from the domain of power systems. In this domain, a typical purpose of an embedded controller is protecting primary equipment, e.g., a transformer. To this end, the controller receives measurements, i.e., voltages or currents, from a sensor and determines from these measurements whether the equipment is in a healthy state. If not, the controller sends a command to a circuit breaker that disconnects the equipment from the network to avoid damage. There are controllers for erroneous situations regarding voltage, current, temperature, and many more.

In our example, we want to run an overcurrent protection and an overvoltage protection on the same device. Both protection functions run at 1 kHz (a typical frequency for this kind of controllers) and rely on measurements that are sampled at 1.6 kHz (a typical sampling frequency in IEC 61850). These measurements need to be resampled by the controllers from 1.6 to 1 kHz to feed consistent data into the protection algorithms. Because of the general setup of the environment, all sensor data from one point in time are sent in one Ethernet packet as defined in IEC 61850-9-2. The structure of the software comprising these functionalities is shown in Figure 1.

![FIGURE 1: Structure of a typical controller software](image)

A typical implementation of such a system would follow the client-server pattern and comprise three processes: overvoltage protection, overcurrent protection, and the sample manager. The latter would be responsible for receiving, parsing, resampling, and storing the data packets temporarily. Ultimately, the protection functions would request resampled data at 1 ms intervals.

Both protection processes would contain one thread each, which would consist of an endless loop that waits for a trigger (either as a direct timer interrupt handler, some IPC mechanism such as a semaphore, or by calling a sleep function) and then run its respective protection algorithm. For obtaining data from the sample manager process, some inter-process communication mechanism would be required.

On the other side, the sample manager would provide two threads for answering to the protection functions and one thread for dealing with incoming network packets. These three threads would have to be synchronized using standard mechanisms such as mutexes to protect access to the sample buffer.

1.2 Issues

This example exhibits several drawbacks of this kind of implementation.

**Predictability** Algorithms for analyzing general thread-based systems, e.g., detecting deadlocks, are hard to implement (see, e.g., [10]). By using specific characteristics of the class of systems under consideration here, predictions could be automated.

**Implementation effort** For every such application, the application engineer has to think thoroughly about the implementation details.

Even though there might be a solution without explicit synchronization mechanisms it is not obvious and possibly not easy.
such as priorities, proper synchronization, possible deadlock scenarios etc. Often, application engineers in the power and automation domain lack formal education in computer science, making the task even harder for them.

**Reusability** While all processes seem to be fairly independent from each other, they do depend on the protocol between sample manager and protection functions. This protocol needs to be implemented by both sides. Further, developers must also decide if the threads are supposed to be run in a light-weight (same process) or in a heavy-weight way (different processes) and must adapt the usage of inter-application communication mechanisms accordingly.

**OS overhead** If there are fewer CPU cores than threads context switches between threads have to occur several times in each cycle, whenever the flow of execution requires one thread to wait for another thread’s output. Because of the short cycle times in many embedded systems, context switches may have a significant impact on the system behavior (see Section 4).

**Communication overhead** The communication must rely on means of inter-process communication, which is considered to be slow (e.g., message passing) or to potentially compromise data integrity (e.g., shared memory).

### 1.3 Related Work

Component-based systems have been proposed to overcome some but not all of these issues. Kopetz [4] describes a component system for real-time systems that addresses predictability and implementation effort. While we use a different, i.e., more fine-grained and restrictive, component model, we follow the proposed concept by using static schedules as advocated also by Locke [9].

Kuz et al. [5] present a component framework on top of the L4 microkernel. It strives for defining operating system functionality (e.g., file system) in a component-based way. In contrast, our framework builds on top of existing real-time operating systems such as RTLinux and focuses on the control applications.

Rastofer and Bellosa [12] aim at separating component functionality from platform mechanisms such as concurrency and synchronization to increase predictability of system properties. Earlier, Bùchi and Weck [2] pointed out that black-box components are not sufficient for analyzing important properties of systems and that white-box components contain too much detail for this task. The decomposition of components into blocks (see Section 2.1) in our approach follows up on their plea for gray-box components.

Reusability has been addressed in a component-based real-time system by Wang et al. [15]. They propose a component-based resource overlay that isolates the underlying resource management from applications to separate the concerns of application designers and component providers. Complementing the work presented in this paper, they focus on the proper allocation of resources to real-time components.

### 2 Component Framework

To address the issues in Section 1.2, we designed a component framework with a runtime concept comprising four structural elements: component, function block, port, and channel. In Section 2.1 and Section 2.2, we describe these concepts in greater detail. Our component framework further encompasses a concept for executing fully deterministic static but replaceable schedules. Application schedules are explained in Section 2.3, their execution is presented in Section 2.4. In Section 2.5, we discuss how the concepts introduced in this section address the aforementioned issues.

#### 2.1 Components and Function Blocks

In our component framework, the example above would consist of three components representing overvoltage protection, overcurrent protection, and sample manager. Conceptually, components separate pieces of software from each other. Their purpose is to provide for the independence required in Section 1.1, hence to ensure that software defects that could affect system stability (e.g., memory safety violations) do not propagate across component boundaries.

Components as such do not directly provide for any executable code; they are in effect comparable to address spaces in typical operating systems. In contrast to a process, which contains threads, a component contains a number of function blocks. A function block (or simply block) is defined as a sequence of instructions with the following properties:

**Sequential** It needs to consist of exactly one stream of instructions that can be executed on one processor core, i.e., it cannot distribute its work
on several cores or threads. The stream of instructions can contain branches and loops as long as the other properties below are satisfied.

**Terminating** Each block needs to guarantee that it finishes its execution within its given deadline for arbitrary inputs if the block is free to run on the CPU.

**Non-Blocking** A block may not depend on any functionality of the underlying platform that could block its execution. This includes in particular synchronization mechanisms, standard I/O operations, and sleep instructions. It does not mean that blocks cannot be used at all to access file systems but it has to be done in a non-blocking way, i.e., by polling.

**Stateless** A block may not keep any state, i.e., the output of a block depends only on its inputs in the same execution cycle. ²

These properties ensure that data flow and control flow are not part of the block’s logic. Instead, block execution is orchestrated by the framework according to given data flow and control flow models. This orchestration is described in Section 2.3.

Compared to a thread as a basic entity in operating systems, a function block does not maintain data across cycles. An important consequence is that the stack associated with the execution of a block is empty after a block finished. Moreover, blocks do not require sophisticated synchronization like threads since they do not rely on blocking mechanisms. In particular, blocks cannot wait for other blocks and thus situations involving deadlocks cannot arise.

In Figure 2, we depict a decomposition of our example from Section 1.1 into components and blocks. The sample manager has a block **Sample Receiver**, which receives samples from the network and one block **Sample Provider** for each protection component that provides the required samples. Each protection component has a block that determines which samples have to be requested from the sample manager and a block containing the actual protection algorithm.

²Of course, some blocks might also depend on the state of the underlying system, e.g., the file system. For the sake of simplicity such blocks and their implications to the system shall be out of scope of this paper.
2.3 Application Schedules

Application schedules describe the execution order and periodicity of the blocks of an application. In particular, they define the control flow between blocks because applications do not change during execution. Such separation of control logics (defined in the blocks) and how individual algorithms operate together (defined in the schedule) allows for easy reuse of blocks in different contexts.

An application schedule is a tree whose leaf nodes are blocks and whose inner nodes (called control nodes) specify details of the execution of their children. Schedule trees can be of arbitrary depth, i.e., control nodes can contain other control nodes. Schedules can be automatically generated from the data flow specified in block diagrams [6] and subsequently be optimized either automatically or manually. The control nodes of our system are:

- **(S)equential** nodes require execution of their child nodes in exactly the sequence specified.
- **(P)arallel** nodes allow for an arbitrary execution order of their child nodes.
- **(A)lternating** nodes alternate through their \( n \) child nodes. In each cycle, the next child gets executed. It has to be specified after how many cycles \( c \geq n \) execution starts again with the first child. This allows for an efficient implementation of multi-rate systems as each child gets executed with a \( c \) times larger period than its parent.
- **(W)atchdog** nodes terminate the execution of their children after a predefined time.

The application schedule for the example of Figure 2 is shown in Figure 4. The root node, which is executed first in each cycle, is a watchdog node that terminates subsequent blocks if they run longer than 850\( \mu \)s. The next block is a sequential node, which first executes the Sample Receiver (SR) block and then a parallel control node. This parallel node specifies that its branches (two sequences of blocks) can be executed in any order.

Sometimes the worst-case execution time of a system cannot be statically verified. In this case, the root node of the schedule should be a watchdog control node with a timeout that is less than the cycle time to ensure the timely termination of the whole schedule. As an example, a system running at 1 kHz should be guarded by a watchdog control node that ends a cycle after 850\( \mu \)s. This leaves a slack time of 150\( \mu \)s in which asynchronous tasks such as an FTP server can be scheduled by the operating system.

Application schedules as presented in this subsection offer several benefits:

- The information about dependencies between blocks is not in the blocks themselves but in the schedule. Thus, blocks can be independently developed, tested, and verified.
- Interaction between blocks is simplified, which makes it easier to reason about the system.
- Blocks can be guaranteed to get their share of CPU time by using time slot guarantees provided by the scheduling mechanisms of modern real-time operating systems. Conversely, blocks can be checked at runtime whether they stick to their time limits using a watchdog timer as described above.
2.4 Execution

Since multiple applications can be executed on the same controller, a system schedule needs to be computed from the individual application schedules. Blocks are executed sequentially in our framework. Therefore, the system schedule defines a total order on the blocks of all applications. This total order must be consistent with the partial orders of the blocks in the individual application schedules.

As an example, assume that the application from Figure 2 is the only application to be scheduled. Since the application schedule of this application (cf. Figure 4) involves parallel control nodes, there are several possible system schedules that satisfy the partial block order. One such system schedule would be the sequence \{SR, CSS, SP, OCP, VSS, SP, OVP\}.

Although system schedules are static, an active schedule can be replaced with another schedule at runtime. In [14] we describe how this is achieved and explain how control software can be updated at runtime using this mechanism.

System schedules are executed by a dispatcher, which executes a sequence of blocks in a cyclic fashion at a given base frequency. To this end, timer interrupts are generated at regular intervals, each of which triggers one execution of the dispatcher. In Figure 5, timer interrupts are displayed as vertical arrows. The dispatcher then calls some or all blocks according to its schedule, which is represented by the gray bars in Figure 5. After the last block has been executed, the dispatcher waits until the next timer interrupt. During this period of time, which we call slack time, other software may run asynchronously, e.g., an FTP server. This software will be preempted by the operating system as soon as the timer interrupt triggers the dispatcher for the execution of the next cycle.

![Timeline of schedule execution](image)

**FIGURE 5:** Timeline of schedule execution

The interval between two timer interrupts is called base cycle time of the system; its inverse is called base frequency. Alternating nodes allow for the implementation of multi-rate systems as long as the cycle times of all blocks are integer multiples of the base cycle time. In other words, the base cycle time should be chosen such that it represents the least common denominator of the individual cycle times of all applications in the system. As an example, if three applications shall be executed that need to be run at 100 Hz, 500 Hz, and 1000 Hz, the base cycle time should be set to 1 ms. The applications can then be executed in every \(10^3\) (100 Hz), in every \(5^3\) (500 Hz), and in every single cycle (1000 Hz).

2.5 Benefits

The concepts of the component framework presented in this section reduces the drawbacks listed in Section 1.2. The static scheduling approach of our framework improves the predictability of the real-time system compared to dynamic scheduling approaches with priority-based scheduling (cf. Liu, Section 4.4 [8]). Its simplified programming model lets developers focus on the algorithms in the blocks instead of priorities and synchronization mechanisms. It has been shown empirically that the implementation effort can be reduced significantly by providing an appropriate programming model [11].

Through the explicit representation of control flow and data flow, blocks become more reusable because they can be developed independently of the context in which they are used. Furthermore, programmers can rely on the robust communication protocol between blocks and do not have to implement a protocol themselves.

We have shown that blocks are executed sequentially. As we will show in Section 3, context switches can be avoided when sacrificing component separation. In this case, OS overhead will be reduced. See Section 5 for a discussion on how this overhead could be reduced while still maintaining component separation.

Furthermore, system engineers have the freedom to adjust the communication overhead depending on their respective priority for performance and safety. This will be discussed in the subsequent section.

3 Implementation

Besides the conceptual separation of concerns by using components on the code level, our framework allows for the physical enforcement of such separation at runtime. By assigning components to distinct address spaces, a defect in one component cannot affect arbitrary other components by corrupting their memory. As an example, on POSIX-based systems, components can run as individual processes and the channels between components can be implemented...
using message passing.

However, this increased level of safety comes at a price: while blocks in the same address space can be executed without overhead, blocks in different address spaces require additional context and address space switches. Whereas components in the same address space can communicate via light-weight mechanisms such as shared memory, components in different address spaces must employ indirect mechanisms such as message passing or sockets for communicating with each other.\(^3\)

To allow system engineers to make a trade-off between performance and safety, our framework can be configured at compile time to enable or disable physical separation while still having the same conceptual separation. Therefore, systems with a high level of safety can be built by implementing physical separation of all components. For such systems, more powerful and thus more expensive CPUs have to be used. If in contrast the price of a system should be kept as low as possible, engineers can waive increased safety and build a system using a low-cost CPU without physical separation of components.

Consequently, we implemented two versions of our framework based on POSIX. In the high-performance (or fast) implementation, the framework and all components are executed in the same address space and shared memory is used for inter-component communication. In the high-safety (or safe) implementation, the framework and each component run as processes in separate address spaces and message queues are used for communication. In Section 4, we compare the performance between these two implementations.

In order to foster reusability of the framework and the components, maximal platform independence has been a major design decision. In fact, both versions of our framework, high-performance and high-safety, share an extensive common code base except for a very thin platform layer. It requires C++ wrapper classes and class templates for only a few platform-specific mechanisms:

- **Timers** allow the framework to perform its execution cycles and watchdog control nodes to terminate blocks.
- **Data Transmission** (e.g., shared memory or message queue) is required for implementing channels.
- **Synchronization** (e.g., mutex, semaphore, or message queue) is used for initiating the execution of blocks and notify the framework of their completion.

### 4 Performance Measurement

In the following, we compare the performance of the high-safety implementation and the high-performance implementation.

The hardware used for the measurements is a PC with an Intel Core2 Duo CPU E6550 running at 2.33 GHz. The Linux kernel (2.6.31-9-rt) was instructed to only use one CPU core. The hardware used for the measurements is certainly more powerful than current embedded hardware. However, the measurements will provide valuable information on the factor by which performance decreases in the high-safety implementation.

Figure 6 shows the setup for measuring. In this setup, two blocks b1 and b2, which belong to different components, communicate with each other. In each cycle, b1 gets the current time from the system and sends the timestamp to b2. In Figure 6, two numbered dotted arrows indicate the following two measurements:

1. **Channel transmission time**: The time it takes for data to be sent across a channel. In the high-performance implementation, a channel is simply a location in memory. In the high-safety implementation, a channel is a message queue provided by the OS.

2. **Block control**: The time it takes for the framework to start the execution of a block and regain control after its execution. The high-performance implementation directly calls functions whereas the high-safety implementation relies on message queues.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{performance Measurement setup.png}
\caption{Performance measurement setup}
\end{figure}

Table 1 lists the results of our performance measurements. Each measurement is the average result

\(^3\)On the other hand, research indicates that message passing can be faster than shared memory on multi-core machines [1].
of hundreds of measurements. The tolerance of best case and worst case is about 10% to 15%.

<table>
<thead>
<tr>
<th></th>
<th>“fast”</th>
<th>“safe”</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Channel transmission</td>
<td>0.02μs</td>
<td>1.97μs</td>
</tr>
<tr>
<td>(2) Block control</td>
<td>0.46μs</td>
<td>6.79μs</td>
</tr>
<tr>
<td>Sum</td>
<td>0.48μs</td>
<td>8.76μs</td>
</tr>
</tbody>
</table>

**TABLE 1: Performance measurement results**

In total, it takes the framework about 0.48μs to schedule a block and a channel in the high-performance implementation and 8.76μs in the high-safety implementation. In measurements performed by Li et al. [7] on a comparable system, the average context switch was around 3.8μs. This indicates that most of the overhead of a safe implementation is caused by context switches and not by the feature that is actually required for safety, address space separation.

5 Conclusions

We have presented a component-based software framework that enforces structurization of cyclic real-time control software systems vertically and horizontally. By decoupling different aspects, such as application logic, control flow, or communication, our approach can be expected to simplify the construction of complex control systems and to reduce the implementation effort.

In addition, static scheduling and non-preemptible execution of function blocks increase the system’s determinism and thus its predictability compared to interleaved execution of threads and dynamic scheduling. Moreover, our approach allows system engineers to adjust the system to their needs if safety is of lesser concern than cost because the number of context switches and thus system overhead can be reduced.

We have shown how the abstractions offered by the framework can be implemented on RTLinux and provided performance measurements for two different implementations.

5.1 Discussion

In domains such as power and automation systems, components are required to be separated from each other at runtime to prevent the propagation of faults across component boundaries. We have shown that it is feasible to construct systems that are either safe or fast.

The prevailing concept of processes and threads, however, makes the construction of systems that are safe and fast difficult. We argue that a small and easy-to-implement modification of this concept will overcome this limitation: Instead of running a thread in the same process, i.e., address space, we propose to allow threads to change the address space during runtime.

With this modification, we could statically schedule the execution of blocks in one thread per CPU core. During execution of this schedule, the thread would enter and leave address spaces in correspondence to the blocks’ components. An obvious advantage is the lack of context switches because blocks always run to completion and because they leave the stack empty after execution.

Moreover, inter-process communication could be implemented efficiently by using shared memory without synchronization mechanisms. Imagine two blocks A and B in different components that are connected by a channel such that A sends data to B. Since B only gets started after A finished its execution, B will always read consistent data from the channel/shared memory. This is still true if the same thread is executing A and B, or if A and B are executed on different cores.

Note that it is also possible to implement an operating system based on our component/block paradigm instead of the process/thread paradigm. We currently consider this idea less feasible. The main reason is that in cyclic control systems there are also sporadic tasks without real-time requirements, e.g., an FTP server. Such low-priority tasks run in the slack time of the real-time cycle, and typically their execution requires the slack time of more than one cycle. They therefore need to be preemptible and the operating system would have to provide some preemption mechanism similar to the one in the process/thread paradigm.

5.2 Future Work

We see considerable potential for automatic tools that can assist in system design. For instance, component boundaries do not have to be drawn arbitrarily. Instead, an automatic tool can formally derive component boundaries according to logical constraints. In our example, the three components have to exist because the two protection functions have to be separated: If one of them fails the other one is not affected. In addition, the sample manager needs to
be separate because it must not be affected by faults from either protection function in order to still be able to serve the other one. However, if there is only one protection function it can be merged with the sample manager because a fault in any block renders the whole system dysfunctional.

In Section 2.4, we showed how the system schedule maps all blocks to be executed onto the same CPU core. This approach can be extended to multiple cores by providing one mapping for each core such that every block is statically assigned to one core. This allows for executing our framework in multi-core, multi-CPU, or even distributed scenarios. Synchronization between the different cores is simplified to well-defined synchronization points because the dependencies between the blocks are explicitly specified in the application schedules.

The dispatcher in our framework can be extended such that blocks are not only executed in sequence, but at precise points in time. This is an effective means for reducing the system’s jitter.

References


Real-Time Performance of L4Linux

Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
adam@os.inf.tu-dresden.de

Janis Danisevskis, Jan Nordholz, Michael Peter
Technische Universität Berlin
Deutsche Telekom Laboratories
Security in Telecommunications
{janis,jnordholz,peter}@sec.t-labs.tu-berlin.de

Abstract

Lately, the greatly improved real-time properties of Linux piqued the interest of the automation industry. Linux holds a lot of promise as its support by a large active developer community ensures an array of supported platforms. At the same time, automation vendors can tap into a huge reservoir of sophisticated software and skilled developers. The open nature of Linux, however, raises questions as to their resilience against attacks, particularly in installations that are connected to the internet.

While Linux has a good security record in general, security vulnerabilities have been recurring and will do so for the foreseeable future. As such, it is expedient to supplement Linux’ security mechanisms with the stronger isolation afforded by virtual machines. However, virtualization introduces an additional layer in the system software stack which may impair the responsiveness of its guest operating systems.

In this paper, we show that L4Linux — an encapsulated Linux running on a microkernel — can be improved on such that it exhibits a real-time behavior that falls very close to that of the corresponding mainline Linux version. In fact, we only measured a small constant increase of the response times, which should be small enough to be negligible for most applications. Our results show that it is practically possible to run security critical tasks and control applications in dedicated virtual machines, thus greatly improving the system’s resilience against attackers.

1 Introduction

Traditionally, embedded systems follow the trend set by their siblings in the desktop and server arena. Rapid technological advances allow for ever more functionality. At the downside, the growing complexity poses a risk as software defects may badly impair the expected machine behavior. This risk is particularly threatening for embedded systems as these systems often have to meet stringent safety requirements.

Another trend carrying over from desktop and servers is the use of open source software, particularly Linux. The adoption of Linux is remarkable insofar as it started out as a hobbyist’s system aimed at desktops. Although neither security nor scalability were considered in the beginning, Linux came to address either of them quite well, making it hugely popular as server operating system. Linux also got a significant foothold in the embedded market. Apart from the absence of licensing costs, its availability for a host of systems holds a lot of appeal for device manufacturers. Having ports of Linux for many platforms is due in no small part to an active community, which allows for improvements to circulate as open source. A third contributing factor for Linux’ popularity is its maturity. For example, as embedded systems embrace network connectivity, they can draw on a mature network stack and a variety of complementing user level software. Originally not being a real-time operating system, the tireless effort of the community managed to evolve Linux to the point where it can run applications with real-time requirements. With that ability, Linux is set to make inroads into markets that used to be the preserve of dedicated real-time operating systems, most of them proprietary.

With regard to security, the operating system is of particular importance. It falls to it to keep applications running on them in check. Yet, prevalent operating systems have a rather poor record when it comes to isolation. The first problem is that they do poorly when it comes to the principle of least
authority. Some applications need special privileges to fulfill their task. Unfortunately, the granularity with which these privileges can be granted is too coarse, leaving many applications running with excessive rights. If such an application succumbs to an attacker, then he can wield that right for its malicious purposes. The point is corroborated by the observation that exploiting applications running with the root identity are in high numbers. However, today’s operating systems cannot be easily phased out because porting their large number of applications and device drivers is a laborious endeavor with uncertain prospects of success.

Confronted with the wanting protection of operating systems, designers of security-critical systems have long turned to physical separation, that is, placing each application on its own machine. While ensuring excellent separation, this approach is not easily applicable to use cases where budget, space, and power limitations have to be met. That is where operating system encapsulation comes into play. While virtualization is currently the most prominent encapsulation technique, it is neither the only one nor the one best suited for all use cases. In the absence of hardware virtualization support, other approaches like OS rehosting might be the better choice.

While virtualization is a well-studied subject, we are not aware of research covering real-time operating systems in virtual machines. In this paper, we set out to investigate whether the real-time behavior of Linux can be preserved if it is executed in a container. Our results indicate that such a setup is feasible if moderately longer event response times can be tolerated.

We will proceed with a brief summary of OS encapsulation use cases followed by a description of L4Linux, the system we build on. The design chapter follows up with a discussion of the problems we faced when deriving an L4Linux version from a Linux with the PREEMPT_RT patch applied. To evaluate the real-time performance of L4Linux, we conducted a number of experiments. The results will be presented before we discuss related work and conclude.

2 Mitigating OS Shortcomings

In the past, monolithic operating system kernels have been perceived as an obstacle for the evolution of secure, robust, and flexible systems. The underlying concern was that their growing complexity would inevitably introduce software defects. If such a defect triggers a fault, the consequences are most likely fatal and will crash the system, which is unacceptable for safety-critical systems.

As an alternative solution, the research community put forth designs based on a minimalistic kernel which is small enough to be thoroughly inspected. Most of the functionality that used to be part of monolithic kernels (device drivers, protocol stacks) would be moved into user level tasks. The rationale was that such an architecture would allow for easy customization by replacing or duplicating the relevant user level components.

As device driver development had been known to be the weak spot for any new system, there was the need to reuse as many device drivers as possible. That raised the question of how to provide the execution environment they were written for. The original vision was that monolithic kernels would be decomposed into small components, each executing in its own protection domain. Since faults would be contained, many system architect nourished the hope that the resulting system would be superior in terms of flexibility, fault tolerance, and security. This endeavor, though, proved complicated. Plans to move to a system completely devoid of legacy operating system code have not materialized so far and it is dubious if they will in the foreseeable future.

As set out above, the insufficiencies of prevalent operating systems pose a risk that may be unacceptable for security- and safety-critical applications. At the same time, they cannot be completely retired as many applications and device drivers depend on them. One solution might be to deploy multiple of them each dedicated to a specific task and ensure that no unsanctioned interference between them occurs. The most frequently cited arguments are the following:

Security. Embedded systems are ever more often connected to the internet whereby the attack surface is drastically enlarged. Given the checkered security record of existing operating system, it seems prudent to place exposed functionality in dedicated compartments which provide an additional line of defense.

Consolidation. High end embedded systems often employ multiple electronic control units. Not only does such a setup incur high material cost, each unit has also to be wired and adds to the overall energy consumption. Consolidating multiple of them onto one unit can yield significant cost, weight, and energy savings.

Certification. So far, safety-certified components could not be deployed alongside noncertified
ones on the same controller necessitating physical separation which incurs costs. A certifiable kernel holds the promise of providing isolation strong enough so that a coexistence on one controller becomes viable.

Development. The diversity of current software may be difficult to leverage if the development is constrained to one operating system. It would be much better if the respective class-leading solutions could be used while preserving the investments in existing software assets.

3 Design

Our design is based on L4Linux[?], a port of the Linux kernel to the L4[5] microkernel. When it was initially developed it set the standard regarding performance for encapsulated operating systems. Ever since it has evolved alongside with Fiasco as the latter improved on its security features (capability-based access control) and became available for multiprocessors and multiple architectures (IA32, AMD64, ARM).

Unlike previous versions of L4Linux, which relied on L4 threads, the current vCPU-based execution model[1] is very similar to the environment found on physical machines. As such, any improvement of the mainline Linux kernel with respect to interrupt latencies should in principle also be available in the corresponding version of L4Linux. The overhead due to the additional context switches and privilege transitions will have an impact but should lengthen the critical paths only by a constant delay.

That said, the previous versions of L4Linux were designed with an emphasis on throughput performance. Design decisions that yield good performance results might however prove problematic for real-time performance. In fact, we came up against three problems of that kind:

Timer. The current version of Fiasco uses a 1 KHz clock to drive its scheduler. Accordingly, timeouts are limited to that granularity as well, which is insufficient for high-resolution timers.

Communication. L4Linux operates in an environment where some services are provided by other tasks. Requesting them involves the platform communication means, in case of Fiasco primarily synchronous IPC. Waiting for an outstanding reply may delay the dispatch of an incoming event.

Memory virtualization. For security reasons, L4Linux cannot manage its page tables directly but has to use L4 tasks instead. It turned out that some applied performance optimizations have an adverse effect on responsiveness.

We will touch on each of these points in the following sections.

3.1 Communication

In our architecture, L4Linux draws on services provided by other tasks. Depending on the use case, communication may either be synchronous or asynchronous. Synchronous IPC has performance advantages for short operations, while asynchronous communication can better hide latencies.

An example where synchronous IPC was chosen is the graphics server. L4Linux maintains a shadow framebuffer and notifies the graphics servers when changes have occurred. The graphics server, which has also access to the shadow framebuffer, then updates the device framebuffer taking into account the current visibility situation (as multiple L4Linux instances can have windows at various positions).

Considering performance, such an arrangement is reasonable. However, whenever a screen update is in progress, the L4Linux main thread cannot pick up on incoming events as it waits for the reply from the graphics server. While this situation is hardly noticeable in performance measurements, latency-sensitive applications are hurt badly. Our solution involves a second thread relieving the main thread from engaging directly in IPC. The two threads themselves interact through a ringbuffer in shared memory and use asynchronous notification. As a result, the main thread is never tied up and can always promptly respond to incoming events.

FIGURE 1: Left: Linux running directly on a machine. Right: Linux running on top of Fiasco, an L4 kernel.
3.2 Timer

Unlike other peripheral devices, timers are not directly accessible by user level components. Where timeouts or delays are required they are typically implemented using IPC timeouts. IPC timeouts as implemented by the Fiasco microkernel have timer tick granularity—typically with a one millisecond period. As this granularity is too coarse to use it as a timer source for a high-resolution timer, we decided to ditch the periodic IPC-timeout based timer. Instead, L4Linux was granted direct access to the HPET device, which can be used to trigger timer events with a high resolution.

3.3 Memory Virtualization

As a user-level task, L4Linux has no direct access to the page tables, which are under the exclusive control of the microkernel. To isolate its processes, L4Linux makes use of L4 tasks. L4 allows two tasks to share memory and provides the original owner with the means to later revoke that sharing. L4Linux uses that memory sharing facility to provision its processes with memory. Whenever Linux modifies the pagetable of one of its processes, this change is reflected in a corresponding change in the memory configuration in the process’ task.

While under normal operations page table updates are propagated individually, the destruction of a process is handled differently. To avoid the overhead of a microkernel syscall for each single page table invalidation, the destruction is performed by a single system call. The destruction of a task address space requires the microkernel to iterate the page directory and return page table to its internal memory pools. Although task destruction can be preempted, it will not be aborted once started. As such, it is fully added to worst-case times of timing critical paths in L4Linux.

We added a thread to L4Linux, which disposes of tasks of perished processes. Not longer executing long-latency syscalls, the main L4Linux thread remains responsive to incoming events.

4 Evaluation

To evaluate our design we conducted a number of experiments. Our test machine contained a 2.7 Ghz Athlon 64 X2 5200+ processor, an nVidia-MCP78-based motherboard and 4 GB RAM. In order to increase the magnitude of the differences between our test scenarios, we reduced the memory available to the OS to 128 MB and disabled the second core of the CPU.

As base OS version we chose the recent Linux version 3.0.3 and the corresponding release of L4Linux. We then took Thomas Gleixner’s Linux-RT patches and applied them to both code bases. L4Linux was supplemented with the current version of Fiasco and a minimal L4 runtime environment, which consisted of a framebuffer and console driver. Both resulting setups were finally booted directly from Grub.

We chose the following software components as our test suite:

- **cyclictest** is a simple benchmark to measure the accuracy of OS sleep primitives. The tool is part of the kernel.org "rt-tests" benchmark suite. To achieve accurate results, we executed it at the highest realtime priority and chose `clock_nanosleep()` as sleep function, as it allows an absolute specification of the wakeup time and thus ignores the time spent setting up the actual timer event.

- **hackbench** transmits small messages between a fixed set of processes and thus exerts stress on the OS scheduler. Its current development version is also hosted through git.

- Finally, the compilation of a Linux kernel both causes a considerable amount of harddisk interrupt activity and creates and destroys a large number of processes. We used a standard 2.6 series Linux source tree and the default i386 configuration as baseline.

4.1 Throughput

To get a feeling for the setup we were about to experiment with, we started out with some throughput measurements. While these are by no means representative due to the restrictions to very little memory and only one core, they serve as a good starting point and already hint at some of the expected results.

As every switch between a userspace task and the L4Linux server involves a round-trip into the microkernel and a switch to a different address space, L4Linux suffers from frequent TLB and cache misses.

---

1. RT-Tests Repository: git://git.kernel.org/pub/scm/linux/kernel/git/clrkwwmls/rt-tests.git
2. Hackbench Repository: https://github.com/kosaki/hackbench
To highlight the effect of this disadvantage, we created an "intermediate" version of native Linux without support for global pages, large pages and with explicit TLB flushes on every kernel entry and exit path.

The hackbench benchmark (cf. Fig. 2) shows the stripped-down version of Linux almost halfway between native Linux and L4Linux, which demonstrates that the impact of repeated cache misses is quite severe when tasks are rapidly rescheduled. The compilation of a Linux kernel (cf. Fig. 3) displays as expected only a mild slowdown for the non-caching Linux variant.

4.2 Memory Management

Another possible source of large latencies are long-running non-abortable system calls. One particularly long operation especially in the Fiasco-based setup is the destruction of an address space, as this requires assistance from the microkernel. To see this problem in effect, we created two latency histograms with cyclicstest under concurrent operation of the hackbench benchmark (cf. Fig. 4) and a kernel compilation (cf. Fig. 5), respectively. While the maxima of both histograms are in the expected order of magnitude, the former shows outliers up to 100µs and the latter (due to its constant destruction of processes) even an almost constant distribution of "long" latencies beyond 40µs.

![FIGURE 2: Runtime of hackbench (40 processes, 100’000 iterations).

![FIGURE 3: Duration of kernel compilation.

As outlined in section 3.3, we therefore externalized the destruction of address spaces to a separate L4 thread. While the execution context issuing the destruction request still waits for the destruction to complete, L4Linux as a whole is then interruptible during the wait and can react to external events.

4.3 Latency under Load

Taking all these findings into consideration, we finally compared native Linux and our improved L4Linux implementation directly using our established benchmark combinations
cyclic/ hackbench and cyclic/kernel-compile. The results are shown in Fig. 6 and Fig. 7.

The first obvious result is that the L4Linux distribution is offset from the native distribution by 5\(\mu\)s. The explanation for this is pretty simple. As L4Linux is not allowed to receive interrupts directly, the microkernel must first handle the interrupt, determine the recipient attached to it and pass the interrupt on. This operation induces a very constant amount of overhead – we measured a delay of 3.65\(\mu\)s in our setup for the delivery of an edge-triggered interrupt. Handling level-triggered interrupts requires even more time, as L4Linux has to issue an extra syscall to unmask the interrupt line once it is done with the device in these cases.

As our timer is edge-triggered, there remains a delay of about 1.3\(\mu\)s. We attribute this to the overhead induced by the microkernel syscalls involved in switching to the real-time task (restoring global descriptor table and execution state) as well as to the aforementioned additional address space switch.

Both distributions reach their maximum not immediately, which means that the ideal interrupt delivery path is not the most frequent. This effect is likely linked to execution paths during which Linux (just as L4Linux) have disabled interrupt reception. The delay incurred by this deferred delivery is more pronounced in the rehosted setup, because deferring the delivery causes additional round-trips to the microkernel once L4Linux has enabled interrupt reception again. Overall though, the difference is not exceptionally large.

Finally, both setups exhibit outliers well beyond the main part of their respective distribution. With hackbench as load-generating benchmark, these show the usual difference between native and rehosted and are therefore no specific effect of an implementation detail of Fiasco or L4Linux. The interrupt-heavy kernel compilation on the other hand demonstrates that additional interrupts (mostly generated by the hard drive controller) affect L4Linux much harder than native Linux due to the interrupt delivery overhead. This effect is even worsened by the fact that the cyclic benchmark has no way to atomically determine the clock overrun once it is awoken from its sleep syscall: interrupts hitting between the sleep and the time lookup are accounted for with their full duration.

L4Linux even has to deal with another interrupt source which is not present in the native scenario: Fiasco employs the on-board PIT for its own scheduling needs and configures it to generate interrupts in regular intervals.

5 Related Work

Virtualization shares many goals with OS encapsulation, it can be even viewed as an alternative implementation. For a long time, the adoption of virtual machines was hampered by the lack of hardware support in processors. The construction of VMs that are efficient, equivalent, and properly controlled requires the instruction set architecture to meet certain requirements[6]. Unfortunately, many widely used processors, in particular those of IA32 provenance[7], do not, and among those which do meet the basic requirements are again many which lack support for virtualization of memory management structures. While this deficiency does not bar virtualization on these processors, performance is indeed severely hampered, as operations on a guest’s page tables require extensive help by the hypervisor[2]. The performance degradation usually cancels the benefits provided by the comfortable hardware virtualization interface, so OS encapsulation remains a viable solution on these platforms.
Virtualization support has also been announced for ARM processors. It remains to be seen how long it takes until processors without it are fully supplanted.

Xen[3] provides for running multiple operating systems on one physical machine. As with L4Linux, guests have to be modified if the underlying machine does not provide virtualization support. Unlike Fiasco, Xen does not provide light-weight task and IPC. Although Xen was ported to the ARM architecture, this port has not found the echo of its siblings on the desktop or server.

Virtualization and microkernel architecture do not rule each other out, they are rather complementary[8, 9]. While thorough research of worst case execution times of microkernels themselves exists[4], so far, neither of the encapsulation approaches has been examined as to their applicability for real-time applications.

6 Conclusion

In this paper we investigated the real-time execution characteristics of an encapsulated operating system. Our results indicated that run-time overhead is incurred in timing-critical paths. However, these overhead is in the order of the latency experienced with native Linux. As such, L4Linux is suitable for use cases where the security of real-time applications shall be bolstered by deploying them in dedicated OS capsules.

Future work will concentrate on reducing the current overheads. We see good chances that the influence of large overhead contributors such as the early timer expiration can be mitigated, e.g. by delaying the timer and rearming it until after the real-time task has finished executing.

Our choice of using OS rehosting as encapsulation technology was mainly motivated by its applicability to a host of processors currently deployed in embedded systems. As hardware virtualization support finds its way into more and more processors, the question comes up whether the software stack underneath a virtual machine can be designed in such a way that it allows for real-time operations in the virtual machine.

References


Tiny Linux Kernel Project: Section Garbage Collection Patchset

Wu Zhangjin
Tiny Lab - Embedded Geeks
http://tinylab.org
wuzhangjin@gmail.com

Sheng Yong
Distributed & Embedded System Lab, SISE, Lanzhou University, China
Tianshui South Road 222, Lanzhou, P.R.China
shengy07@lzu.edu.cn

Abstract

Linux is widely used in embedded systems which always have storage limitation and hence requires size optimization. In order to reduce the kernel size, based on the previous work of the “Section Garbage Collection Patchset”, This paper focuses on details its principle, presents some new ideas, documents the porting steps, reports the testing results on the top 4 popular architectures: ARM, MIPS, PowerPC, X86 and at last proposes future works which may enhance or derive from this patchset.

1 Introduction

The size of Linux kernel source code increases rapidly, while the memory and storage are limited in embedded systems (e.g. in-vehicle driving safety systems, data acquisition equipments etc.). This requires small or even tiny kernel to lighten or even eliminate the limitation and eventually expand the potential applications.

Tiny Lab estimated the conventional tailoring methods and found that the old Tiny-linux project is far from being finished and requires more work and hence submitted a project proposal to CELF: “Work on Tiny Linux Kernel” to improve the previous work and explore more ideas.

“Section garbage collection patchset(gc-sections)” is a subproject of Tiny-linux, the initial work is from Denys Vlasenko[9]. The existing patchset did make the basic support of section garbage collection work on X86 platforms, but is still outside of the mainline for the limitation of the old GNU toolchains and for there are some works to be done(e.g. compatibility of the other kernel features).

Our gc-sections subproject focuses on analyzes its working mechanism, improves the patchset(e.g. unique user defined sections), applies the ideas for more architectures, tests them and explores potential enhancement and derivation. The following sections will present them respectively.

2 Link time dead code removing using section garbage collection

Compiler puts all executable code produced by compiling C codes into section called .text, r/o data into section called .rodata, r/w data into .data, and uninitialized data into .bss[2, 3]. Linker does not know which parts of sections are referenced and which ones are not referenced. As a result, unused(or ‘dead’) function or data cannot be removed. In order to solve this issue, each function or data should has its own section.

gcc provides –ffunction-sections or –fdata-sections option to put each function or data
to its own section, for instance, there is a function called `unused(func())`, it goes to `.text.unused func` section. Then, `ld` provides the `--gc-sections` option to check the references and determine which dead function or data should be removed, and the `--print-gc-sections` option of `ld` can print the the function or data being removed, which is helpful to debugging.

The following two figures demonstrates the differences between the typical object and the one with `-ffunction-sections`:

![Figure 1: Typical Object](#)

![Figure 2: Object with -ffunction-sections](#)

To learn more about the principle of section garbage collection, the basic compiling, assebling and linking procedure should be explained at first (Since the handling of data is similar to function, will mainly present function below).

### 2.1 Compile: Translate source code from C to assembly

If no `-ffunction-sections` for gcc, all functions are put into `.text` section (indicated by the `.text` instruction of assembly):

```
$ echo 'unused(){} main(){}' | gcc -S -x c -o - - \
  | grep .text

.main
  .text
    main: instructions
    func: instructions
```

Or else, each function has its own section (indicated by the `.section` instruction of assembly):

```
$ echo 'unused(){} main(){}' \$
  | gcc -ffunction-sections -S -x c -o - - | grep .text

.main
  .section .text.unused,"ax",@progbits
  .section .text.main,"ax",@progbits
```

As we can see, the prefix is the same `.text`, the suffix is function name, this is the default section naming rule of gcc.

Expect `-ffunction-sections`, the section attribute instruction of gcc can also indicate where should a section of the function or data be put in, and if it is used together with `-ffunction-sections`, it has higher priority, for example:

```
$ echo '__attribute__((__section__(".text.test"))) unused(){} \ 
  __attribute__((__section__(".text.test"))) main(){}' \ 
  | gcc -ffunction-sections -S -x c -o - - | grep .text

.main
  .section .text.test,"ax",@progbits
  .section .text.main,"ax",@progbits
```

`.text.test` is indicated instead of the default `.text.unused`. In order to avoid function redefinition, the function name in a source code file should be unique, and if only with `-ffunction-sections`, every function has its own unique section, but if the same section attribute applies on different functions, different functions may share the same section:

```
$ echo '__attribute__((__section__(".text.test"))) unused(){} \ 
  __attribute__((__section__(".text.test"))) main(){}' \ 
  | gcc -ffunction-sections -S -x c -o - - | grep .text

.main
  .section .text.test,"ax",@progbits
  .section .text.main,"ax",@progbits
```

Only one section is reserved, this breaks the core rule of section garbage collection: **before linking, each function or data should has its own section**, but sometimes, for example, if want to call some functions at the same time, section attribute instruction is required to put these functions to the same section and call them one by one, but how to meet these two requirements? use the section attribute instruction to put the functions to the section named with the same prefix but unique suffix, and at the linking stage, merge the section which has the same prefix to the same section, so, to linker, the sections are unique and hence better for dead code elimination, but still be able to link the functions to the same section. The implementation will be explained in the coming sections.

Based on the same rule, the usage of section attribute instruction should also follow the other two rules:

1. The section for function should named with `.text` prefix, then, the linker may be able to merge all of the `.text` sections. or else, will not
be able to or not conveniently merge the sections and at last instead may increase the size of executable for the accumulation of the section alignment.

2. The section name for function should be prefixed with .text. instead of the default .text prefix used gcc and break the core rule. And we must notice that: ‘You will not be able to use “gprof” on all systems if you specify this option and you may have problems with debugging if you specify both this option and -g.’ (gcc man page)

```sh
$ echo ' unused (){} main (){} ' | 
  gcc -ffunction-sections -p -x c -o test -
  <stdin>:1:0: warning : -ffunction-sections disabled ; 
  it makes profiling impossible
```

2.2 Assemble: Translate assembly files to binary objects

In assembly file, it is still be possible to put the function or data to an indicated section with the .section instruction (.text equals .section “.text”). Since -ffunction-sections and -fdata-sections doesn’t work for assembly files and they has no way to determine the function or data items, therefore, for the assembly files written from scratch (not translated from C language), .section instruction is required to added before the function or data item manually, or else the function or data will be put into the same .text or .data section and the section name indicated should also be unique to follow the core rule of section garbage collection.

The following commands change the section name of the ‘unused’ function in the assembly file and show that it does work.

```sh
$ echo 'unused(){} main(){}' | 
  gcc -ffunction-sections -S -x c -o test -
  <stdin>:1:0: warning : -ffunction-sections disabled ; 
  it makes profiling impossible
```

2.3 Link: Link binary objects to target executable

At the linking stage, based on a linker script, the linker should be able to determine which sections should be merged and included to the last executables. When linking, the -T option of ld can be used to indicate the path of the linker script, if no such option used, a default linker script is called and can be printed with ld --verbose.

Here is a basic linker script:

```sh
OUTPUT_FORMAT("elf32-1386", "elf32-1386")
OUTPUT_ARCH(1386)
ENTRY(_start)
SECTIONS{
  .text:
  { *(.text .stub .text.* .gnu.linkonce.t.*) } ...
}
.data:
{ *(.data .data.* .gnu.linkonce.d.*) } ...
/DISCARD/ : { *(.note.GNU-stack) *(.gnu.ito_*) }
```

The first two commands tell the target architecture and the ABI, the ENTRY command indicates the entry of the executable and the SECTIONS command deals with sections.

The entry (above is _start, the standard C entry, defined in crt1.o) is the root of the whole executable, all of the other symbols (function or data) referenced (directly or indirectly) by the the entry must be kept in the executable to make ensure the executable run without failure. Besides, the undefined symbols (defined in shared libraries) may also need to be kept with the EXTERN command. and note, the --entry and --undefined options of ld functions as the same to the ENTRY and EXTERN commands of linker script respectively.

--gc-sections will follow the above rule to determine which sections should be reserved and then pass them to the SECTIONS command to do left merging and including. The above linker script merges all section prefixed by .text, .stub and .gnu.linkonce.t to the last .text section, the .data section merging is similar. The left sections will not be merged and kept as their own sections, some of them can be removed by the /DISCARD/ instruction.

Let’s see how --gc-section work, firstly, without it:

```sh
$ echo 'unused(){} main(){}' | gcc -x c -o test -
```

```
$ objdump -d test | grep .section
Disassembly of section .text.text: Disassembly of section .text.main:
```

```
Second, With --gc-sections (passed to ld with -Wl option of gcc):
$ echo 'unused(){} main(){}' | gcc -ffunction-sections -Wl,-gc-sections -x c -o test -
```

```
$ size test text data bss dec hex filename 800 252 8 1060 424 test
```

It shows, the size of the .text section is reduced and --print-gc-sections proves the dead ‘unused’ function is really removed:
The above output also proves why the size of the .data section is also reduced.

But if a section is not referenced (directly or indirectly) by the entry, for instance, if want to put a file into the executable for late accessing, the file can be compressed and put into a .image section like this:

```c
SECTIONS {
  ... .data {
    __image_start = .;
    *(.image)
    __image_end = .;
  } ...
}
```

The file can be accessed through the pointers: __image_start and __image_end, but the .image section itself is not referenced by anybody, then, --gc-sections has no way to know the fact that .image section is used and hence removes .image and as a result, the executable runs without expected. In order to solve this issue, another KEEP instruction of the linker script can give a help.

```c
SECTIONS {
  ... .data {
    __image_start = .;
    KEEP(*(.image))
    __image_end = .;
  } ...
}
```

3 Section garbage collection patchset for Linux

The previous section garbage collection patchset is for the -rc version of 2.6.35, which did add the core support of section garbage collection for Linux but it still has some limitations.

Now, let’s analyze the basic support of section garbage collection patchset for Linux and then list the existing limitations.

3.1 Basic support of gc-sections patchset for Linux

The basic support of gc-sections patchset for Linux includes:

- Avoid naming duplication between the magic sections defined by section attribute instruction and -ffunction-sections or -fdata-sections

The kernel has already defined some sections with the section attribute instruction of gcc, the naming method is prefixing the sections with .text., as we have discussed in the above section, the name of the sections may be the same as the ones used by -ffunction-sections or -fdata-sections and hence break the core rule of section garbage collections.

Therefore, several patches has been upstreamed to rename the magic sections from {.text.X, .data.X, .bss.X, .rodata.X} to {.text..X, .data..X, .bss..X, .rodata..X} and from {.text.X.Y, .data.X.Y, .bss.X.Y, .rodata.X.Y} to {.text..X..Y, .data..X..Y, .bss..X..Y, .rodata..X..Y}, accordingly, the related headers files, c files, assembly files, linker scripts which reference the sections should be changed to use the new section names.

As a result, the duplication between the section attribute instruction and -ffunction-sections/-fdata-sections is eliminated.

- Allow linker scripts to merge the sections generated by -ffunction-sections or -fdata-sections and prevent them from merging the magic sections

In order to link the function or data sections generated by -ffunction-sections or -fdata-sections to the last {.text, .data, .bss, .rodata}, the related linker scripts should be changed to merge the corresponding {.text.*, .data.*, .bss.*, .rodata.*} and to prevent the linker from merging the magic sections(e.g. .data..page_aligned), more restrictive patterns like the following is preferred:

* {.text.[A-Za-z0-9_]$}*

A better pattern may be the following:

* {.text.[^-]*}*

Note, both of the above patterns are only supported by the latest ld, please use the versions newer than 2.21.0.20110327 or else, they don’t work and will on the contrary to generate bigger kernel image for ever such section will be
linked to its own section in the last executable and the size will be increased heavily for the required alignment of every section.

- Support objects with more than 64k sections

The variant type of section number (the `shnum` member of `elf{32,64}_hdr`) is `u16`, the max number is 65535, the old modpost tool (used to postprocess module symbol) can only handle an object which only has small than 64k sections and hence may fail to handle the kernel image compiled with huge kernel builds (allyesconfig, for example) with `-ffunction-sections`. Therefore, the modpost tool is fixed to support objects with more than 64k sections by the document “IA-64 gABI Proposal 74: Section Indexes”:


- Invocation of `-ffunction-sections/-fdata-sections` and `--gc-sections`

In order to have a working kernel with `-ffunction-sections` and `-fdata-sections`:

```sh
$ make KCFLAGS="-ffunction-sections -fdata-sections"
```

Then, in order to also garbage-collect the sections, added

```sh
LDFLAGS_vmlinux += --gc-sections
```

in the top-level Makefile.

The above support did make a working kernel with section garbage collection on X86 platforms, but still has the following limitations:

1. Lack of test, and is not fully compatible with some main kernel features, such as Ftrace, Kgcov
2. The current usage of section attribute instruction itself still breaks the core rule of section garbage collections for lots of functions or data may be put into the same sections (e.g. `__init`), which need to be fixed
3. Didn’t take care of assembly carefully and therefore, the dead sections in assembly may also be reserved in the last kernel image
4. Didn’t focus on the support of compressed kernel images, the dead sections in them may also be reserved in the last compressed kernel image
5. The invocation of the gc-sections requires to pass the gcc options to ‘make’ through the environment variables, which is not convenient
6. Didn’t pay enough attention to the the kernel modules, the kernel modules may also include dead symbols which should be removed
7. Only for X86 platform, not enough for the other popular embedded platforms, such as ARM, MIPS and PowerPC

In order to break through the above limitations, improvement has been added in our gc-sections project, see below section.

### 3.2 Improvement of the previous gc-sections patchset

Our gc-sections project is also based on mainline 2.6.35(exactly 2.6.35.13), it brings us with the following improvement:

1. Ensure the other kernel features work with gc-sections
   Ftrace requires the `__mcount_loc` section to store the mcount calling sites; Kgcov requires the `.ctors` section to do gcov initialization, these two sections are not referenced directly and will be removed by `--gc-sections` and hence should be kept by the `KEEP` instruction explicitly. Besides, more sections listed in `include/asm-generic/vmlinux.lds.h` or the other arch specific header files has the similar situation and should be kept explicitly too.

```sh
/* include/asm-generic/vmlinux.lds.h */
... *(__mcount_loc) \ + KEEP(*(__mcount_loc)) \ ...
... *(.ctors) \ + KEEP(*(.ctors)) \ ...
... 
```

2. The section name defined by section attribute instruction should be unique
   The symbol name should be globally unique (or else gcc will report symbol redefinition), in order to keep every section name unique, it is possible to code the section name with the symbol name. `FUNCTION` (or `func` in Linux) is available to get function name, but there is no way to get the variable name, which means there is no general method to get the symbol name so instead, another method should be used, that is coding the section name with line number and a file global counter. the combination of these two will minimize the duplication of the section name (but may also exist duplication) and also reduces total size cost of the section names.
gcc provides __LINE__ and __COUNTER__ to get the line number and counter respectively, so, the previous __section() macro can be changed from:

```
#define __section(S) \ 
   __attribute__ (( __section__(#S )))
```

...to the following one:

```
#define __concat (a, b) a##b
#define __unique_impl(a , b) __concat (a, b)
#define __ui(a, b) __unique_impl(a, b)
#define __unique_counter(a) \
   __ui (a, __COUNTER__)
#define __uc(a) __unique_counter(a)
#define __unique_line(a) \
   __ui(a, __LINE__)
#define __ul(a) __unique_line(a)
#define __unique_string(a) \ 
   __stringify(__unique(a))
#define __us(a) __unique_string(a)
#define __unique(__) __uc ( __ui( __ul(a), l_c))
#define __unique_string(__) \ 
   __stringify(__unique(__))
#define __section(S) \ 
   __attribute__ (( __section__( __us(S .))))
```

Let’s use the __init for an example to see the effect, before, the section name is .init.text, all of the functions marked with __init will be put into that section. With the above change, every function will be put into a unique section like .text.init.13l and make the linker be able to determine which one should be removed.

Similarly, the other macros used the section attribute instruction should be revisited, e.g. sched.

In order to make the linker link the functions marked with __init to the last .init.text section, the linker scripts must be changed to merge .init.text.* to .init.text. The same change need to be applied to the other sections.

3. Ensure every section name indicated in assembly is unique

-ffunction-sections and -fdata-sections only works for C files, for assembly files, the .section instruction is used explicitly, by default, the kernel uses the instruction like this: .section .text, which will break the core rule of section garbage collection tool, therefore, every assembly file should be revisited.

For the macros, like LEAF and NESTED used by MIPS, the section name can be uniqued with symbol name:

```
#define LEAF(symbol) \ 
   " . section . text; \ 
+ . section . text.asm.symbol;"
```

But the other directly used .section instructions require a better solution, fortunately, we can use the same method proposed above, that is:

```
#define __asm_section(S) \ 
   . section __us (S.)
```

Then, every .section instruction used in the assembly files should be changed as following:

```
/* include/linux/init.h */
#define __HEAD . section " .head.text","ax"
#define __HEAD __asm_section(__us(S .))
```

4. Simplify the invocation of the gc-sections

In order to avoid passing -ffunction-sections, -fdata-sections to ‘make’ in every compiling, both of these two options should be added to the top-level Makefile or the arch specific Makefile directly, and we also need to disable -ffunction-sections explicitly when Ftrace is enabled for Ftrace requires the -p option, which is not compatible with -ffunction-sections.

Adding them to the Makefile directly may also be better to fix the other potential compatibilities, for example, -fdata-sections doesn’t work on 32bit kernel, which can be fixed as following:

```
# arch/mips/Makefile
ifndef CONFIG_FUNCTION_TRACER
   cflags-y := -ffunction-sections
endif
# FIXME: 32 bit doesn’t work with -fdata-sections
ifdef CONFIG_64BIT
   cflags-y += -fdata-sections
endif
```

Note, some architectures may prefer KBUILD_CFLAGS than cflags-y, it depends.

Besides, the --print-gc-sections option should be added for debugging, which can help to show the effect of gc-sections or when the kernel doesn’t boot with gc-sections, it can help to find out which sections are wrongly removed and hence keep them explicitly.

```
# Makefile
ifeq ($( KBUILD_VERBOSE) ,1)
   LDFLAGS_vmlinux += --print-gc-sections
endif
```

Then, make V=1 can invocate the linker to print which symbols are removed from the last the executables.

In the future, in order to make the whole gc-sections configurable, 4 new kernel config options may be required to reflect the selection of -ffunction-sections, -fdata-sections, --gc-sections and --print-gc-sections.

5. Support compressed kernel image

The compressed kernel image often include a compressed vmlinux and an extra bootstraper, the bootstraper decompress the compressed...
kernel image and boot it, the bootstraper may also include dead code, but for its Makefile does not inherit the make rules from either the top level Makefile or the Makefile of a specific architecture, therefore, this should be taken care of independently.

Just like we mentioned in section 2.3, the section stored the kernel image must be kept with the KEEP instruction, and the -ffunction-sections, -fdata-sections, --gc-sections and --print-gc-sections options should also be added for the compiling and linking of the bootstraper.

6. Take care of the kernel modules

Currently, all of the kernel modules share a common linker script: scripts/module-common.lds, which is not friendly to --gc-sections for some architectures may requires to discard some specific sections. therefore, a arch specific module linker script should be added to arch/ARCH/ and the following lines should be added to the top-level Makefile:

```
# Makefile
+ LDS_MODULE = -T $(srctree)/arch/$ (SRCARCH)/module.lds
+ LDFLAGS_MODULE = $(if $(wildcard arch/$ (SRCARCH)/module.lds),
    $(LD_MODULE))
```

Then, every architecture can add the architecture specific parts to its own module linker script, for example:

```
# arch/mips/module.lds
SECTIONS {
  /DISCARD/ : {
    *(.MIPS.options)
  }...
}
```

In order to remove the dead code in the kernel modules, it may require to enhance the common module linker script to keep the functions called by module_init() and module_exit(), for these two are the init and exit entries of the modules. Besides, the other specific sections (e.g. .modinfo, _version) may need to be kept explicitly. This idea is not implemented in our gc-sections project yet.

7. Port to the other architectures based platforms

Our gc-sections have added the gc-sections support for the top 4 architectures (ARM, MIPS, PowerPC and X86) based platforms and all of them have been tested.

The architecture and platform specific parts are small but need to follow some basic steps to minimize the time cost, the porting steps to a new platform will be covered in the next section.

3.3 The steps of porting gc-sections patchset to a new platform

In order to make gc-sections work on a new platform, the following steps should be followed (use ARM as an example).

1. Prepare the development and testing environment, including real machine(e.g. dev board) or emulator(e.g. qemu), cross-compiler, file system etc.

For ARM, we choose qemu 0.14.50 as the emulator and versatilepb as the test platform, the corss compiler (gcc 4.5.2, ld 2.21.0.20110327) is provided by ubuntu 11.04 and the file system is installed by debootstrap, the ramfs is available from http://d-i.debian.org/daily-images/armel/.

2. Check whether the GNU toolchains support -ffunction-sections, -fdata-sections and --gc-sections, if no support, add the toolchains support at first.

The following command shows the GNU toolchains of ARM does support gc-sections, or else, there will be failure.

```
$ echo ‘unused (){} main (){} ’ | arm-linux-gnueabi-gcc -ffunction-sections-Wl,--gc-sections -S -x c -o - - |
```

3. Add -ffunction-sections, -fdata-sections, at proper place in arch or platform specific Makefile.

```
# arch/arm/Makefile
ifndef CONFIG_FUNCTION_TRACER
 KBUILD_CFLAGS += -ffunction-sections
 endif
 KBUILD_CFLAGS += -fdata-sections
```

4. Fix the potential compatibility problem (e.g. disable -ffunction-sections while requires Ftrace)

The Ftrace compatibility problem is fixed above, no other compatibility has been found up to now.

5. Check if there are sections which are unreferenced but used, keep them

The following three sections are kept for ARM:
# arch/arm/kernel/vmlinux.lds.S
...
KEEP (*(. proc.info.init*))
...
KEEP (*(. arch.info.init*))
...
KEEP (*(. taglist.init*))

6. Do basic build and boot test, if boot failure happens, use make V=1 to find out the wrongly removed sections and keep them explicitly with the KEEP instruction

```
$ qemu-system-arm -M versatilepb -m 128M \ -kernel vmlinux -initrd initrd.gz \ -append "root=/dev/ram init=/bin/sh"
```

7. Add support for assembly files with the _asm_section() macro
   Using grep command to find out every .section place and replace it with the _asm_section() macro, for example:

```
# arch/arm/am/proc-am926.S
- .section "...rodata" + _asm_section(.rodata)
```

8. Follow the above steps to add support for compressed kernel
   Enable gc-sections in the Makefile of compressed kernel:

```
# arch/arm/boot/compressed/Makefile
EXTRA_CFLAGS+=-ffunction-sections -fdata-sections ...
LDFLAGS_vmlinux -= --gc-sections ifeq ($KBUILD_VERSIBLE),1
LDFLAGS_vmlinux += --print-gc-sections endif ...
```

   And then, keep the required sections in the linker script of the compressed kernel:

```
# arch/arm/boot/compressed/vmlinux.lds.in
KEEP(* (.start))
KEEP(* (.text))
KEEP(* (.text.call_kernel))
```

9. Make sure the main kernel features (e.g. Ftrace, Kgcov, Perf and Oprofile) work normally with gc-sections
   Validated Ftrace, Kgcov, Perf and Oprofile on ARM platform and found they worked well.

10. Add architecture or platform specific module.lds to remove unwanted sections for the kernel modules
   In order to eliminate the unneeded sections(e.g. .fixup, .ex_table) for modules while no CONFIG_MMU, a new module.lds.S is added for ARM:

```
# arch/arm/module.lds.S

SECTIONS {
/ DISCARD / : {
 ifndef CONFIG_MMU
 *(. fixup)
 *(.ex_table)
 #endif
}
# arch/arm/Makefile
extra-y := module.lds
```

11. Do full test: test build, boot with NFS root filesystem, the modules and so forth.
   Enable the network bridge support between qemu and your host machine, open the NFS server on your host, config smc91c111 kernel driver, dhcp and NFS root client, then, boot your kernel with NFS root filesystem to do a full test.

```
$ qemu-system-arm -kernel path/to/zImage \ -append "init=/bin/bash init=/dev/nfs \ ip=dhcp"
```

4 Testing results

Test has been run on all of the top 4 architectures, including basic boot with ramfs, full boot with NFS root filesystem and the main kernel features (e.g. Ftrace, Kgcov, Perf and Oprofile).

The host machine is thinkpad SL400 with Intel(R) Core(TM)2 Duo CPU T5670, the host system is ubuntu 11.04.

The qemu version and the cross compiler for ARM is the same as above section, the cross compiler for MIPS is compiled from buildroot, the cross compiler for PowerPC is downloaded from emdebian.org/debian/, the details are below:

<table>
<thead>
<tr>
<th>arch</th>
<th>board</th>
<th>net</th>
<th>gcc</th>
<th>ld</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>versatilepb</td>
<td>emo91c111</td>
<td>4.5.2</td>
<td>2.21 0.20110327</td>
</tr>
<tr>
<td>MIPS</td>
<td>malta</td>
<td>pcnet</td>
<td>4.5.2</td>
<td>2.21</td>
</tr>
<tr>
<td>PPC</td>
<td>g3beige</td>
<td>pcnet</td>
<td>4.4.5</td>
<td>2.20.1 20100303</td>
</tr>
<tr>
<td>X86</td>
<td>pc-0.14</td>
<td>net2k</td>
<td>4.5.2</td>
<td>2.21 0.20110327</td>
</tr>
</tbody>
</table>

**TABLE 1:** Testing environment

Note:

- In order to boot qemu-system-ppc on ubuntu 11.04, the openbios-ppc must be downloaded from debian repository and installed, then use the -bios option of qemu-system-ppc to indicate the path of the openbios.
Due to the regular expression pattern bug of ld <2.20 described in section 3, in order to make the gc-sections features work with 2.20.1.20100303, the linker script of powerpc is changed through using the pattern .text.*, .data.*, .bss.*, .sbss.* but to avoid wrongly merging the kernel magic sections (e.g. .data..page_aligned) to the .data section, the magic sections are moved before the merging of .data, then it works well because of the the .data..page_aligned will be linked at first, then, it will not match .data.* and then will not go to the .data section. Due to the inconvenience of this method, the real solution will be forcing the users to use ld >= 2.21, or else, will disable this gc-sections feature to avoid generating bigger kernel image.

```plaintext
SECTIONS {
  .data..page_aligned : ... {
    PAGEAligned_DATA(PAGE_SIZE)
  }
  .data : AT(ADDR(.data) - LOAD_OFFSET) {
    DATA_DATA
    *(.data.*)
    ...
    *(.sdata.*)
    ...
  }
  ...
}
```

The following table shows the size information of the vanilla kernel image (vmlinux, .orig) and the kernel with gc-sections(.gc), both of them are stripped through `strip -x` (or even `strip s`) because of gc-sections may introduce more symbols (especially, non-global symbols) which are not required for running on the embedded platforms.

The kernel config is gc_sections_defconfig placed under arch/ARCH/configs/, it is based on the versatile_defconfig, malta_defconfig, pmac32_defconfig and i386_defconfig respectively, extra config options only include the DHCP, net driver, NFS client and NFS root file system.

<table>
<thead>
<tr>
<th>arch</th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
<th>save</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>3062975</td>
<td>137504</td>
<td>198940</td>
<td>3650762</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3034990</td>
<td>137120</td>
<td>198688</td>
<td>3608866</td>
<td>-1.14%</td>
</tr>
<tr>
<td>MIPS</td>
<td>3952132</td>
<td>220664</td>
<td>134400</td>
<td>4610028</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3899224</td>
<td>217560</td>
<td>123224</td>
<td>4545436</td>
<td>-1.40%</td>
</tr>
<tr>
<td>PPC</td>
<td>5945289</td>
<td>310362</td>
<td>153188</td>
<td>6671729</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5849879</td>
<td>309326</td>
<td>152920</td>
<td>6560912</td>
<td>-1.66%</td>
</tr>
<tr>
<td>X86</td>
<td>2302973</td>
<td>317220</td>
<td>1086632</td>
<td>3572700</td>
<td>-2.61%</td>
</tr>
<tr>
<td></td>
<td>2106804</td>
<td>311292</td>
<td>498916</td>
<td>3657200</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 2: Testing results**

5 Conclusions

The testing result shows that gc-sections does eliminate some dead code and reduces the size of the kernel image by about 1~3%, which is useful to some size critical embedded applications.

Besides, this brings Linux kernel with link time dead code elimination, more dead code can be further eliminated in some specific applications (e.g. only parts of the kernel system calls are required by the target system, finding out the system calls really not used may guide the kernel to eliminate those system calls and their callees), and for safety critical systems, dead code elimination may help to reduce the code validations and reduce the possibility of execution on unexpected code. And also, it may be possible to scan the kernel modules ('make export-report' does help this) and determine which exported kernel symbols are really required, keep them and recompile the kernel may help to only export the required symbols.

Next step is working on the above ideas and firstly will work on application guide system call optimization, which is based on this project and maybe eliminate more dead code.

And at the same time, do more test, clean up the existing patchset, rebase it to the latest stable kernel, then, upstream them.

Everything in this work is open and free, the homepage is tinylab.org/index.php/projects/tinylinux, the project repository is gitorious.org/tinylab/tinylinux.

References


Performance Evaluation of openPOWERLINK

Yang Minqiang, Li Xuyuan, Nicholas Mc Guire, Zhou Qingguo

*Distributed & Embedded System Lab, SISE, Lanzhou University, China
Tianshui South Road 222, Lanzhou, P.R.China
yangmq09@lzu.edu.cn

Abstract

For the low cost and high bandwidth of Ethernet, there have been a lot of investigations to implement real-time communication over Ethernet. To overcome the intrinsic non-determinism of Ethernet, a few of real-time Ethernet variants turn up but industrial users are still hesitating about the choice among those variant. openPOWERLINK is a open source industrial Ethernet solution which follows POWERLINK - a real-time Ethernet protocol. For the good performance and wide platform support of RT-PREEMPT, a soft PLC which consists of openPOWERLINK and RT-PREEMPT might be a cheap and easy solution for many deployment cases. In this paper, we would evaluate synchronicity, jitter, cycle time and other most relevant indicators of quality of openPOWERLINK on RT-PREEMPT in distributed systems which are used commonly to implement tightly coordinated controllers, data acquisition or synchronization systems. To allow evaluating such typical scenarios with inexpensive hardware we used the parallel-port to simulate an input/output unit of a PLC. We designed some benchmark cases to evaluate the related indicators and represented the result as reference data. Further with the increased use of COTS and FLOSS components for safety related systems, which are often distributed/replicated systems, the results of this evaluation may also be of interest to designers of safety related systems. OpenPOWERLINK provides basic capabilities suitable to build replicated or redundant systems (i.e. TMRs).

Keywords: openPOWERLINK, EPL, Real Time Ethernet, VIAC3

1 Introduction

It is generally known that Ethernet has low-price devices, while providing higher and higher speeds, but also its intrinsic non-deterministic. A further noteworthy advantage clearly is the availability of well trained engineers as Ethernet is a widely in-use standard technology. In order to utilize these good features, a lot of researches and attempts to use Ethernet in the industrial control context are underway. Real-time Ethernet[1] is a communication architecture using standard Ethernet hardware with various modifications which introduce real-time and deterministic property to meet the requirements of industrial control system. These modifications lie on different layers in TCP/IP reference model[2].

Ethernet variants like Modbus/TCP use the TCP/IP protocol stack without any modifications, but it only fulfills the requirement of the low level real-time demand applications. Another kind of Ethernet field-buses are realized with modified Ethernet hardware, e.g. EtherCAT. It generally means that you could get better performance (cycle time and throughput) and you probably cannot use the normal Ethernet device directly. Solutions such as EtherCAT require special Ethernet controller which have two Ethernet ports and the capability of processing packets "on the fly"[4].

Ethernet POWERLINK (EPL) is another variant of Ethernet; it introduces Slot Communication Network Management (SCNM) to provide deterministic communication and leaves the hardware without any modification, but uses its own software protocol stack (note that the EPL NIC utilizes the CANopen application layer interface). This compromise on the modification to the Ethernet brings a balance be-
tween cost and performance. Moreover, for its good synchronization, EPL attracted much attention of industrial users.

To provide reference to industrial users and related developers, we built a simple distributed system with openPOWERLINK[9] which is the open-source implementation of EPL. Some demo and benchmark applications were implemented for benchmark. In the following sections, we will introduce our work and represent the evaluation result.

2 Background: EPL and openPOWERLINK

Bringing together Ethernet, CANopen, and a newly developed stack for real-time data communication, POWERLINK integrates features and abilities from three different worlds. In contrast to a number of competing products, POWERLINK keeps very close to the Ethernet standard, retaining original Ethernet features, and thus reducing the cost of industrial deployment. It expands Ethernet with a mixed Polling and Time slicing mechanism named SCNM (refer to figure 1).

There are two kinds of nodes in EPL, namely managing node (MN) and controlled node (CN). A MN, which acts as the master in the EPL network, polls the CN cyclically. This process takes place in the isochronous phase of the EPL cycle. Immediately after the isochronous phase follows an asynchronous phase for communication which is not time-critical, e.g. TCP/IP communication. The isochronous phase starts with a Start of Cyclic frame on which all nodes are synchronized. This schedule design avoids collisions, which are usually present on Standard Ethernet, and ensures the determinism of the hard real-time communication. It is implemented in the EPL data link layer. The SoC packet is sent for synchronizing and indicating that the start of the isochronous phase of a new cycle. SoA starts the asynchronous phase[16].

EPL integrates the CANopen, a robust and proven protocol widely used throughout the automation world, which greatly simplifies setting up networks because of its extensive standardization.

CANopen is one of the most popular higher layer protocols for CAN-based networks. Therefore there are a number of device and application profiles under development or already available which are used in example in building related applications like door control or elevators, for ships, trains, municipal vehicles or railway as well as for medical applications. Besides these standardized profiles, another big advantage of CANopen is that it’s used in a wide range of proprietary systems and applications which eases integration of these and the transformation to open-source.

openPOWERLINK is an open source industrial Ethernet solution provided by SYSTEC electronic [10]. It contains the Ethernet POWERLINK protocol stack for the Managing Node (master) and for the Controlled Nodes (slaves). It is released under the BSD License.

3 Performance Evaluation

3.1 Performance Indicators

There are a lot of indicators that should be considered when evaluating real-time Ethernet. We paid more attention on the most important performance indicators (PIs) in this paper, namely synchronicity, minimum cycle time, latency and jitter, and the scalability over the number of end nodes. Some other PIs...
were also considered, such as topology, throughput etc[3]. To get the worst case values, all the tests in the paper have been done with heavy system load.

3.2 Setup

We built a three nodes distributed system using VIA boards with little modification on the default configuration of openPOWERLINK. The motivation for the three node setup being that a common setup safety related systems is as a triple modular redundancy (TMR) - thus this is one of the target profiles we are interested in for a real-time Ethernet solution. There was one MN and two CNs. The three nodes were connected via 10Mbp HUB or 100Mbp Switch (the ideal solution a 100Mbp hub was not available). A parallel-port cable was attached to each CNs. Suitable pins of the other end of the cable were connected to an oscilloscope. We used two channels of the oscilloscope to display the signals from the two CNs - one channel for each CN. The structure of the setup was figured as follows.

![System Setup Diagram](image)

**Fig. 3: System Setup**

The following are the detail of the system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Model/Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>VIA Nehemiah (C3), 1GHz</td>
</tr>
<tr>
<td>NIC</td>
<td>RTL8139 10M/100Mbps</td>
</tr>
<tr>
<td>RAM</td>
<td>256M</td>
</tr>
<tr>
<td>Hub</td>
<td>Ovislink 10Mbps</td>
</tr>
<tr>
<td>Switch</td>
<td>TP-Link TL-SF1016 10/100Mbps</td>
</tr>
<tr>
<td>EPL</td>
<td>openPOWERLINK 1.6</td>
</tr>
<tr>
<td>RRTOS</td>
<td>RT-PREEMPT linux-2.6.33.9-rt31</td>
</tr>
<tr>
<td>Distribution</td>
<td>Debian GNU/Linux 5 and 6</td>
</tr>
</tbody>
</table>

Before we run openPOWERLINK, we have to make sure our RTOS is correctly configured and running properly. To ensure this the de-facto standard cyclictest[11] benchmark was used - the overall result for RT-PREEMPT on VIA C3 is quite good - results are plotted in the following figures.

![System Latency of RT-PREEMPT](image)

**Fig. 4: System Latency of RT-PREEMPT**

We run cyclictest with different interval (500, 1000, 2000 in macrosecond) and the worst case latency were less than 80 microsecond. The distributions of the latency over frequency are quit stable.

3.3 Synchronicity

As we described above, there was one MN and two CNs in our setup. To meet our requirement, we need to configure the related entries of object dictionary and let the MN send a 8-bit real-time data to each of the two CNs periodically. The two CNs output the data to their parallel-port when they get the next SoC packet. To facilitate our programming, the MN sent 0x00H and 0xFFH to the CNs separately and reversed them in the next cycle. The snapshot of the oscilloscope below shows the output of the CNs.

Implementation of MN includes the following main parts.

- Define object dictionary
- Link the variable to the corresponding object
- Event callback function
  TPDO should be initialized when the reset-communication event occurs.
- Synchronization callback function
  The synchronization function invoked while transmitting SoC packet will reverse the output value to generate the PWM signal.
- Clean up
  Close NMT and EPL stack

The CN also need the object dictionary to be defined as well as link the variable with the entry of the object dictionary. The related callback functions should be properly setup and configured following the specification[12][16]. Make sure that no
other machine is connected to the hub the EPL environment is using, if there are, make sure the machine does not send any packet to the network. Raise the priority of the interrupt handler thread of openPOWERLINK to a value above the other peripherals and system management kernel threads (we used a priority above 50 - 50 being the default priority of IRQ-threads in RT-Preempt prior to Linux 3.X).

EPL protocol provides good synchronization capabilities, but the latency of software stack of openPOWERLINK on the CNs and RTOS mainly contribute the synchronization deviation, we need to know how good the synchronicity we could get by running openPOWERLINK as CNs the latency of MN does not affect the synchronization and the jitter of NIC and Ethernet HUB is negligible). As we described above, the parallel-port of the two CNs were connected to the two channels of the oscilloscope. The oscilloscope used the signal of one channel (A) as the reference (trigger), while the signal of the other channel (B) would fluctuate relatively and is a direct measure for the synchronicity of the two CNs. By accumulating the wave shape, we could get the relative synchronization performance of openPOWERLINK. We run the openPOWERLINK for 30 minutes with heavy load and recorded the persistent mode snapshot in which it indicated about 100 microsecond’s synchronization deviation. Note that the deviation is not the worst case synchronization time, because the worst cases of the two nodes probably did not occur at the same time. The worst case synchronization time should be less than the deviation - to estimate this the distribution of jitter would need further study.

3.4 Cycle time and number of nodes

The cycle time is the key issue of real-time communication for many automation and control applications. The homepage of openPOWERLINK announced that “POWERLINK, which operates with standard, on-board Ethernet controllers, reaches cycle times down to 0.5 milliseconds in this Open Source implementation while ensuring high synchronicity. Supported by co-processors, POWERLINK even ensures cycle times down to 0.1 ms.”[9] Some benchmark results on openPOWERLINK by B & R also presented results indicating that the minimal cycle time of 250 microseconds was achieved[7] though in the presented setup such cycle times are out of reach.

The jitter of the SoC can directly reflect the overall real-time performance of openPOWERLINK on RT-PREEMPT, which can be seen as reference for the system (including application, RTOS and hardware) jitter. To measure the jitter, a rising edge or a falling edge was generated alternately on the parallel-port while sending SoC packet. We used the persist mode of oscilloscope to capture the PWM signal.
The cycle time which is commonly one of the most critical indicators depends on the system jitter, number of nodes and transmission hardware (NIC and Ethernet). Ethernet packet capture tools like Wireshark cannot meet the high precise timer requirement for a system analysis. Thus we need to record the time stamps via high resolution timer. The MN and CNs used independent clocks, so we cannot compare the time stamp from different nodes without time synchronization. Our solution is to record the related time stamps on the same machine so that we could get the duration for different phases. The following figure showed a simplified communication period.

Fig. 8: Cycle Timing

Refer to figure 8 for the latency benchmark of some steps which mainly contribute to the overall openPOWERLINK latency. To analyze the overall real-time performance and cycle time, namely $T_{sr}$ (duration from sending SoC to getting PRes), $T_{rr}$ (time from sending PReq to getting PRes).

The whole cycle of EPL can be simply formulated below,

$$T_C = T_{soc} + T_{rr} + n + T_{ai}$$ (1)

in which, $T_C$ is the cycle time. $T_{soc}$ is the time for delivering SoC and the safety margin after it. $T_{rr}$ indicates the time from sending PReq to getting PRes. $n$ is the number of nodes. $T_{ai}$ is the time for asynchronous period and the idle phase. Obviously, we have

$$T_{soc} = T_{sr} - T_{rr}$$ (2)

The histogram of $T_{sr}$ and $T_{rr}$ is showed in Figure 9 and Figure 10.

Fig. 9: Timing over 10Mbps HUB

<table>
<thead>
<tr>
<th>Setup</th>
<th>$T_{avg}$ sr</th>
<th>$T_{avg}$ rr</th>
<th>$T_{avg}$ soc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10M Hub</td>
<td>437</td>
<td>176</td>
<td>261</td>
</tr>
<tr>
<td>100M Switch</td>
<td>279</td>
<td>75</td>
<td>204</td>
</tr>
</tbody>
</table>

Because the system (protocol stack, RTOS, NIC and switch etc.) have jitters and the data might be lost for various factor. The polling mechanism of EPL protocol requires a timeout $T_o$ within which the MN should get the PRes packet after it sent PReq packet to the CN, otherwise MN stop waiting the CN and turn to poll next CN. If the timeout value is too big, the cycle time will be unnecessary long. On the other side that, if the timeout value is too small, the late PRes from the previous CN will collide with the PReq to the next CN. The EPL standard does not define the timeout value which is closely dependent on the EPL implementation, system, hardware and the transport media, so that we have to define...
the timeout value which fits to our specific environment and setup. A new equation can be derived from Equation 1,
\[ T_C = T_{soc} + T_{rr} \times n + T_a + T_o \times n \] (3)
in which \( T_a \) is the time for asynchronous phase (that we can refer to the \( T_{rr} \) to calculate); \( T_o \times n \) is the idle time to ensure the jitter of the system will not affect the regular cyclic communication. Note that the possibility of the worst case latency occur at the same time is very low. We do not need a safety margin as long as \( T_o \times n \). In other words, you might need to find out a balance between cycle time and failure rate (here means the tolerable rate of the actual cycle time exceeds the configured value).

As soon as the timeout, specification of communication (RT and non-RT data amount) and the number of nodes are specified, referring to the data and equation above we could get the time for delivering SoC (including the gap) and the round time for PReq and Pres and the time for the asynchronous and idle phase. And go a step further, we could roughly estimate the possible cycle length of the EPL network with 10Mbps HUB or 100Mbps Switch.

Comparing to the system jitter, the jitter of switch is negligible in our case. We could take the average value of the parameters as the reference to estimate the possible cycle time we can get (note we got data in overload system condition). Normally the latency of a 100Mbps switch is about 10 microseconds (the precise value depends on a specific hardware), so you may use a 100Mbps Hub to avoid the latency generated by switch and get better performance.

### 3.5 Other indicators

Up to 240 CNs can be employed connected in various configurations. You may use both hubs (hub is recommended for its small latency) or switches (you need to consider the latency) in more than one level with different topology. Rather than fixed topology of EtherCAT, EPL has flexible topology. A mixed tree and line structure is available when a large number of nodes are being used. As the intrinsic Ethernet property, the EPL network can be easily connected via gateways to non-real-time networks.

The throughput is closely related to the amount of data to send during one cycle and minimal cycle time. After getting those data, we could easily estimate the relationship between throughput, payload and cycle time. As the Ethernet has enough bandwidth to fulfill the RTE and non-RTE throughput, we would not extend the issue in this paper.

### 4 Conclusion

As the strong potential of introducing Ethernet into distributed real-time control system, we took several important performance indicators of real-time Ethernet into consideration when evaluating openPOWERLINK. User could easily setup the EPL on a Linux machine without any special hardware or particular topology and get relatively good hard real-time communication. This feature facilitates the user and saves much cost.

In this paper, we studied the openPOWERLINK on a few significant indicators, benchmarked the critical phases in the EPL communicate cycle and gave a reference model for industrial user or related researchers and developers. The data we got indicates that the cycle time and synchronization performance of openPOWERLINK-on-RT-PREEMPT solution meet the requirement of process control systems and most motion control systems[2]. The data shows that the system jitter is quite big which makes a great impact on the EPL cycle time and synchronicity. To use it in high precision motion control system, there are some points need to be optimized, such as the software architecture, code and RT-PREEMPT etc. Besides optimization of those points and implementation of a real distributed control which has mechanical equipment integrated, to seek for a solution to effectively reuse the Ethernet card driver in Linux kernel is also our future work.

### 5 Reference

#### References


[12] EPSG, *Ethernet POWERLINK V2.0 Communication Profile Specification, Version 1.0.0*


Improving Responsiveness for Virtualized Networking Under Intensive Computing Workloads

Tommaso Cucinotta
Scuola Superiore Sant’Anna
Pisa, Italy
cucinotta@sssup.it

Fabio Checconi
IBM Research, T.J. Watson
Yorktown Heights, NY, USA
fchecon@gmail.com

Dhaval Giani
Scuola Superiore Sant’Anna
Pisa, Italy
dhaval.giani@gmail.com

September 27, 2011

Abstract

In this paper the problem of providing network response guarantees to multiple Virtual Machines (VMs) co-scheduled on the same set of CPUs is tackled, where the VMs may have to host both responsive real-time applications and batch compute-intensive workloads. When trying to use a real-time reservation-based CPU scheduler for providing stable performance guarantees to such a VM, the compute-intensive workload would be scheduled better with high time granularities, to increase performance and reduce system overheads, whilst the real-time workload would need lower time granularities in order to keep the response-time under acceptable levels. The mechanism that is proposed in this paper mixes both concepts, allowing the scheduler to dynamically switch between fine-grain and coarse-grain scheduling intervals depending on whether the VM is performing network operations or not. A prototype implementation of the proposed mechanism has been realized for the KVM hypervisor when running on Linux, modifying a deadline-based real-time scheduling strategy for the Linux kernel developed previously. The gathered experimental results show that the proposed technique is effective in controlling the response-times of the real-time workload inside a VM while at the same time it allows for an efficient execution of the batch compute-intensive workload.

Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7 under grant agreements n. 214777 “IRMOS—Interactive Realtime Multimedia Applications on Service Oriented Infrastructures” and n. 248465 “SoOS – Service-oriented Operating Systems.”
1 Introduction and Related Work

Virtualization is increasingly gaining momentum as the enabling technology for the management of physical resources in data centers and Infrastructure-as-a-Service (IaaS) providers in the domain of Cloud Computing. Indeed, virtualization enhances the flexibility in managing physical resources, thanks to its capability to virtualize the hardware so as to host multiple Virtual Machines (VMs) executing potentially different Operating Systems, and the capability to live-migrate them as needed without interrupting the provided service, except for a very low downtime. Virtualized systems are also capable of exhibiting a performance nearly equal to the one experienced on the bare metal, due to the hardware virtualization extensions provided by modern processors.

As a consequence of virtualization, multiple under-utilized servers can easily be consolidated onto the same physical host. This allows a reduction in the number of required physical hosts to support a number of virtualized OSes, leading to advantages in terms of costs for running the infrastructure and of energy impact.

However, once multiple VMs are deployed on the same physical resources, their individual performance is at risk of becoming greatly unstable, unless proper mechanisms are utilized. A VM which temporarily saturates either the processing, networking, or storage access capacity of the underlying physical resources immediately impacts the performance of the other VMs which share the same resources. This is a potentially critical issue for IaaS providers where proper QoS specifications are included in the Service-Level Agreements (SLAs) with the customers.

The problem of providing a stable performance to individual VMs has been studied in the past. For example, Gupta et al. [8] introduce in the Xen hypervisor a proper CPU scheduling strategy accounting for the consumption of device driver domain(s) as due to the individual VMs operations. In [11], an extension to the Xen credit-based scheduler is proposed, to improve its behavior in presence of multiple different applications with I/O bound workloads. Also, Liao et al. [9] propose to modify the Xen CPU scheduler, by making it cache aware, and the networking infrastructure to improve the performance of virtualized I/O on 10Gbps Ethernet.

For the KVM hypervisor, Cucinotta et al. [3, 4, 5] investigated on the use of hierarchical deadline-based real-time CPU scheduling [2] for the Linux kernel in order to stabilize the performance of individual compute-intensive VMs, tackling the problem of network-intensive VMs later [6].

The latter works rely on the use of a reservation-based scheduler [2] for the CPU (a hard-reservation variant of the Constant Bandwidth Server [1]) that allows for configuring the scheduling guarantees for a given VM in terms of a budget ($Q$) and a period ($P$). The scheduler will guarantee that each VM will be scheduled for $Q$ time units every period of $P$ time units, under the usual assumption of non-saturation for EDF ($\sum_i \frac{Q_i}{P_i} \leq 1$, see [10] for details). The reservation period can be specified independently for each VM, and it constitutes the time granularity over which the CPU allocation is granted to the VM.

A shorter period improves the responsiveness of the VMs at the cost of higher scheduling overheads, thus being beneficial for time-sensitive workloads. On the other hand, a longer period leads to lower scheduling overheads, thus it is beneficial for batch and high-performance workloads, at the cost of potentially longer time intervals during which the VM is unresponsive (in the worst-case, a VM might have to wait as much as $2(P - Q)$ before being scheduled again). However, for VMs embedding both batch computing activities (including both main VM functionality or typical bookkeeping OS activities, such as updating indexes) and time-sensitive tasks (e.g., reporting on the progress of batch tasks, or realizing independent features), both configurations do not fit very well, as highlighted by Dunlap in the discussion about future work on the new upcoming Xen Credit Scheduler [7].

In this paper we propose a novel mechanism for scheduling VMs with both compute-intensive and network-responsive workloads. In absence of external requests the VM progresses with its (long) period configuration (e.g., hundreds of ms) and can perform batch computing activities reducing scheduling overheads to the minimum. However the occurrence of external requests allows the VM to be woken up by the scheduler within a much shorter interval (e.g., ms or tens of ms), to perform relatively short activities configured at a higher priority inside the VM, so as to respond very quickly to external events.

---

2 Approach

The mechanism proposed in this paper applies to virtual machines scheduled under a reservation-based real-time scheduler like the one presented in [2]. For the sake of simplicity the focus is on single-core VMs scheduled according to a partitioned EDF policy (so one or more VMs are pinned on each physical core and scheduled on it).

Each VM can be configured with a set of scheduling parameters denoted by \((Q, P)\), with the meaning that \(Q\) time units are granted to the VM for each \(P\) time units. The interest in having \(Q/P < 1\), thus the possibility to have multiple VMs co-scheduled on the same processor and core, comes from the fact that the infrastructure provider may have an interest in “partitioning” the big computing power available on a single powerful core into multiple VMs with lower computing capabilities and rent them separately, or merely from the fact that the hosted VMs have an expected workload (e.g., as due to requests coming from the network) that cannot saturate the computing power on the underlying physical core, thus enabling the provider to perform server consolidation. The \(Q\) value constitutes both a guarantee and a limitation (i.e., we are using hard reservations). This ensures that the performance of each VM is not affected (too much) from how much intensively other VMs are computing [5, 4].

Roughly speaking, at equal \(Q\) over \(P\) ratios, the chosen value for \(P\) regulates the responsiveness of the associated VM. It is easy to see that, if the VM is running alone, then its schedule comes out as shown in Figure 1, and the non-responsiveness time interval for the VM may be as long as \(P \sim Q\). However, the worst-case condition when the VM is co-scheduled with other VMs is the one shown in Figure 2, with the budget granted to the VM at the beginning of a \(P\) time window (for example, because at that time all other VMs were idle), and at the end of the time window immediately following (for example, as due to the wake-up of a VM at the beginning of this second time window, with a deadline slightly shorter than the first VM, under theoretical saturation for the EDF scheduler).

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure1.png}
\caption{Example schedule of a VM with generic scheduling parameters of \((Q, P)\), when running alone, exhibiting a nonresponsiveness time interval of \(P - Q\).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure2.png}
\caption{Example schedule of a VM with generic scheduling parameters of \((Q, P)\), when co-scheduled, exhibiting a nonresponsiveness time interval of \(2(P - Q)\).}
\end{figure}

Also, \(P\) controls the scheduling overheads imposed on the system. In fact, the scheduler forces a context switch at least every interval as long as the minimum \(P\) value across all the reservations configured on the core.

This kind of scheduler allows heterogeneous virtualized workloads to safely coexist as far as they belong to different VMs. One can easily configure a short \(P\) value for a VM with a real-time workload that needs to be responsive, and a long \(P\) for a VM that performs mainly batch computations. However, mixing such types of workloads in the same VM may lead to problems. One can configure the responsive activities in the VM to run at a higher priority as compared to the batch computing ones (i.e., by exploiting priority-based scheduling as available on every OS). However, still the non-responsive periods of the VM will largely dominate the response time of the real-time task(s). So, in order to keep such response times low, the normal option would be the one to use small \(P\) values, obtaining high scheduling overheads also while the VM is doing its batch computing activities without any request from the outside triggering the real-time functionality.

In order to resolve this problem, in this paper we propose the following mechanism (see Figure 3). The VM is normally attached to a reservation configured with scheduling parameters \((Q, P)\), with a period \(P\) tuned for the batch computing case, i.e., it is relatively large, for example in the range of hundreds of milliseconds. In addition, a second “spare” reservation is configured in the system with parameters \((Q_s, P_s)\) tuned for the operation of the real-time activity, i.e., \(P_s\) is relatively small, for example in the range of tens of milliseconds or shorter, and \(Q_s\) sufficient to complete an activation of the real-time activity. Now, whenever the VM receives a network packet and its current budget is exhausted (i.e., it is in the non-responsive time frame), the VM
is temporarily attached to the “spare” reservation. Having a much shorter deadline, the spare reservation forces the VM to be scheduled and receive \( Q_s \) execution time units on the processor within the \( P_s \) deadline from the packet receive time; this will cause the VM to run, receive the packet and possibly activate the real-time activity that will perform some fast computation (and possibly provide a response packet). If the real-time activity cannot complete within the first activation of the spare reservation, it will be resumed during the subsequent activations, so it will receive additional \( Q_s \) time units during the following \( P_s \) time window, and so on, till the time of replenishment of the original reservation budget, at which time the VM relinquishes the spare reservation. With a proper tuning of the \( Q_s \) and \( P_s \) parameters a VM configured for batch computing activities should exhibit a tremendously improved response-time to sporadic requests coming from the network, at the cost of keeping some extra-capacity unused in the system.

**FIGURE 3:** Example schedule of a VM with generic scheduling parameters of \((Q, P)\), and a spare reservation of \((Q_s, P_s)\) which is dynamically activated and attached to the same VM on a new packet arrival. Despite the budget for the VM at packet arrival time was exhausted, the VM can complete a short real-time activity of duration \( Q_s \) within the spare reservation period \( P_s \).

The requirements of the real-time workload are assumed to be relatively small, and in any case the additional reservation to be attached dynamically to a VM cannot be too large in terms of utilization (budget over period), because it needs to remain unused for all the time in which the VM does not access the network. For example, it might require a 10% or a lower CPU utilization to complete. This should allow the real-time activity triggered by the received network packet to complete, assuming it is configured in the VM for running at higher priority than other activities. For example, the VM may perform kernel-level activities inside the networking driver and stack, and relatively short userspace activities, which may be running in a task that was waiting for the packet arrival.

Finally, in order to avoid keeping a spare reservation for each and every VM hosted onto the same physical host, we propose to use a pool of spare reservations which can be used for the purpose illustrated above. The idea is that, exploiting statistical multiplexing of the networking traffic patterns among independent VMs, one can assume that the probability of having all the VMs requiring a spare reservation attached dynamically at the same time be very small. This way, the additional utilization to keep for spare reservations may be kept limited.

Therefore, a pool of a few reservations with short periods will be ready to be used for boosting reservations (with longer periods) of VMs when they receive packets from the external world but their normal budget is exhausted due to compute-intensive activities. This allows for a very quick reaction-time of the VMs.

### 3 Implementation Details

In order to validate the proposed approach we implemented a proof of concept in the Linux kernel, using the KVM hypervisor to execute the VMs. We started from the IRMOS scheduler [2], modifying it to include support for reservations providing “spare” bandwidth, and introducing the glue code needed to use this new feature.

From the interface point of view, each reservation may have the property of *providing* spare bandwidth to the reservations needing it, and/or the property of *using* spare bandwidth from reservations providing it. The system administrator controls the parameters of the reservations and the dependencies between users and providers of spare bandwidth using the CGROUP filesystem interface.

To recognize the events that are related to VM I/O, and consequently activate the spare bandwidth mechanism we modified the networking code. In our modified kernel, when a packet arrives we check its destination and if is headed towards a Virtual Machine we retrieve its server using a simplified hash table. If the server has run out of bandwidth we set a flag to mark that it needs to access its spare reservation. Setting the flag may also imply requeuing the running tasks belonging to the same VM, as they may need to access the spare bandwidth too.

When a task is activated, along as performing a regular activation, the scheduler checks if the task belongs to a virtual machine, and if the VM’s server needs spare bandwidth; if this is the case, the task is not only enqueued in its own server, as would be
done anyway, but it is also enqueued in the server providing the spare reservation.

The flag set on the VM’s server needs to be reset, and this may happen on two conditions. The first possibility is when the emergency bandwidth has been set for a certain duration, empirically determined not as a function of time, but rather of the chances the server has had to execute its tasks. The other possibility is when the original server has its bandwidth restored.

4 Experimental Results

The approach presented in the previous section was validated through an experiment conducted on a prototype implementation of the mechanism, evaluated on a Linux 2.6.35 kernel patched with the IRMOS real-time scheduler [2], running on an Intel Core 2 Duo P9600 CPU configured for running at a fixed 2.66 GHz frequency. The VM was configured with the CPU thread running at real-time priority lower than the one used for all its other threads. We were unable to use the full implementation described in Section 3, and we used only a subset of it, handling part of the transition to the spare reservation from userspace; however in the experiments we made sure that the mapping of the VMs to the reservation was compatible with the described approach.

In order to show the advantages of the technique, the ping times for reaching the VM have been measured under various conditions (so, the ping time is representative of the responsiveness of the VM), while a fake compute-intensive workload was used inside the VM, using a throughput utility that has the capability to measure how many repetitions of a basic for loop with a few arithmetic operations have been realized over a time horizon. Note that a ping packet only reaches the kernel-level network driver of the target VM (which runs at higher priority as compared to user-space computing applications). The evaluation of the technique with real user-space applications (e.g., a webserver that needs to remain responsive) is deferred as future work on the topic.

In the experiment, the potential of the mechanism is highlighted by measuring the worst-case responsiveness of the system, under the assumption of sporadically interspaced, non-enqueuing ping requests, while the VM is under heavy compute-intensive workload. This has been achieved running the throughput utility inside the VM, attaching it to a reservation with scheduling parameters \((Q, P) = (40ms, 100ms)\), and by using a spare reservation configuration of \((Q_s, P_s) = (4ms, 10ms)\). Also, in order to evaluate the worst-case latency experienced by ping, the VM was pinned on the first physical core of the host, while a user-space tool, pinned on the other core, was used to spin-wait for budget exhaustion of the associated reservation, and issue a ping request at that time. As highlighted in Section 2, the minimum observed ping time is theoretically \(P - Q = 60ms\) in this case (but far higher values were observed, actually). However, the mechanism introduced in this paper foresees the attachment of the spare reservation to the VM at the ping packet receive time, thus the VM has a chance to run for \(Q_s = 1ms\) within the deadline of \(P_s = 10ms\) (and for an additional 1ms for each subsequent 10ms time window, till the replenishment of the original reservation budget), thus responding to the request much more quickly.

The obtained ping times with the VM running under the real-time scheduler are shown in Figure 4. As it can be seen when using the spare reservation (bottom curve) mechanism, the experienced ping times are highly reduced as compared to when not using it (top curve).

![FIGURE 4: Obtained ping times without the additional spare reservation (top curve) and with the spare reservation (bottom curve).](image)

Also, looking at the throughput that can be achieved by the batch computing activities inside the VM with various equivalent reservation configurations (in terms of occupied CPU share), we can observe that with a reservation of \((40ms, 100ms)\) our program was reporting 1.11 cycles per microsecond, while with a reservation of \((4ms, 10ms)\) it was reporting 0.56 cycles. The big difference is due to the additional scheduling overheads due to the ten times more context switches. Therefore, it is highly beneficial to keep the VM configured with the longer period, in this case, while our mechanism allows to greatly improve its responsiveness.
5 Conclusions and Future Work

In this paper we present a novel scheduling mechanism to provide efficiently a tight responsiveness to virtual machines hosting mixed compute-intensive and real-time workloads. It is possible to schedule such VMs with a reservation-based scheduler by using a large period for minimum overheads during compute-intensive periods, but at the same time ensure that the VM responds within a much shorter deadline when receiving input from the outside, as in the case of receiving a network packet.

The presented mechanism can still be improved, and various directions for future extension are possible. Firstly, the mechanism may be improved to shorten the response-time of the VM also during the periods in which its own reservation has still budget, but the deadline is still quite far away. This may be seen in workloads where the other VMs in the system have deadlines shorter than the one of the VM receiving the packet, but still quite far away as compared to the desired tightness of the real-time activity response. Second, the current implementation is only a proof-of-concept and needs to be better engineered to reach production quality levels. Third, the idea of the pool of spare reservations has only been sketched out, but it needs to be refined, implemented and experimented on a real system. Finally, the presented mechanism needs to be applied to some real-life workload in order to highlight its full potential for real application scenarios.

References


Evaluation of RT-Linux on different hardware platforms for the use in industrial machinery control

Thomas Gusenleitner, DI(FH)
ENGEL Austria GmbH
Ludwig-Engel-Strasse 1, A-4311 Schwertberg
thomas.gusenleitner@engel.at

Gerhard Lettner, BSc
ENGEL Austria GmbH
Ludwig-Engel-Strasse 1, A-4311 Schwertberg
gerhard.lettner@engel.at

October 3, 2011

Abstract

Using Linux and Open Source in an industrial environment is becoming more and more common, in part to ensure participation with daily improvements and compatibility with future development. One of the most important requirements in the environment of industrial machinery control is realtime, so we decided to evaluate RT-Linux on different hardware platforms. To generate a realistic load which is comparable to the real machinery control a simplified version of machinery control, called testplc, was developed and used in the hardware assessments conducted. The results of this evaluation should give a clear statement about the applicability of each hardware platform for the machinery control area.

1 Introduction

Repeatable cycles is the main goal of our machinery control, so one of the preconditions is an exact and reproducible timing of the scheduler. The machinery control in real is a bit too complex for this evaluation so we decided to implement a little test control application with a comparable core design.

To get a concrete appreciation of the interaction between RT-Linux, CPU, chipset and board design we evaluate the same application with the same RT-Linux kernel on different hardware platforms.

2 Method

To get a wide range of results we take some really different hardware platforms from old industrial over modern energy saving up to high performance and consumer electronics.

In this environment its impossible to take the real control application for evaluation on these different hardware platforms, the used test control application is explained in the next section.

The used operation system is based on debian lenny, the kernel version is 2.6.33.9 which is patched with the realtime patch rt312. More details see chapter 2.2.

The aim of this evaluation work is to get more into details using RT-Linux for industrial machin-
ery control. It’s very interesting to implement a small test control application designed towards the real control application. The concrete results give us a better understanding which influences lead to a scheduling malfunction and the properties of multi core cpus are displayed in the latency histograms.

Indeed a machinery control application in real life has to deal with locking, interrupts, priority inheritance and other complicated stuff, but if this simple test control application shows problems in scheduling and runtime a real machinery control application will never work.

2.1 Test Control Application

The test control application is implemented in pure C with posix threads. Five of these threads build the core of this control application with different cycle times and priorities to simulate the real machinery control. The timing is controlled with absolute timestamps starting with a global start time. In every cycle the period time is added to the absolute time. Each thread greps the actual system time after returning from `clock_nanosleep` before it calculates the difference between scheduled time and the actual time. We call this difference latency and inserted the value in a histogram. The parameters for `clock_nanosleep` are `clockId=CLOCK_MONOTONIC` and `flags=TIMER_ABSTIME`. For security reasons we introduce a latency threshold to detect a scheduling fault.

Every thread has a number of functions to execute, for example calculate 100 double multiplications, 500 double multiplications, sorting lists, calculating pi and communicate 1024 bytes over udp to a server. The udp communication is only done in the 10000 micro second cycle task to avoid network problems. This communication generates a huge amount of interrupts to stress the scheduler.

For measurement we take the time of 500 double multiplications in nano seconds which are also stored in a histogram. The timing and priority configurations of these threads are

1. 100 us cycle time with priority 80
2. 500 us cycle time with priority 75
3. 1000 us cycle time with priority 70
4. 2000 us cycle time with priority 65
5. 10000 us cycle time with priority 60

This test control application gives us two histograms as result. The first one is a latency histogram of all five threads within a range of 0 to 300 micro seconds. The second one is a timing histogram of the 500 double multiplications within a range of 0 to 20000 nano seconds.

2.2 RT-Linux

To get comparable results its evident to use equal Kernel configurations for the evaluation on each hardware platform. As already mentioned in chapter 2 the used Kernel was patched using the required RT-Patch which is available at www.kernel.org. To ensure the real time behaviour the kernel configuration (see section 2.2.1) as well as the runtime configuration (see section 2.2.2) is important and was adopted.

2.2.1 Kernel configuration

The following items in the Kernel configuration were considered:

**Processor type and features**
- [ * ] High Resolution Timer Support
- [ * ] Symmetric multi-processing support
- Preemption Mode
  - (*) Compl.Preemption (Real-Time)
- [ * ] Generix x86 support
- Timer frequency
  - (x) 1000 HZ

**Power management and ACPI options**
- [ * ] Power Management Support
- [ ] CPU Frequency Scaling

**Kernel Hacking**
- [ * ] Tracers
  - [ * ] Scheduling Latency Tracer
  - [ * ] Scheduling Latency Histogram
  - [ * ] Missed Timer Offset Histogram

2.2.2 Runtime configuration

To ensure real time behavior during runtime, the Real-Time group scheduling must be modified. Therefore the content of the files `/proc/sys/kernel/sched_rt_period_us` and `/proc/sys/kernel/sched_rt_runtime_us` has to be set equal. The standard content for `sched_rt_period_us` is 1000000 (1s) and for
sched_rt_runtime_us 950000 (0.95s). Using the standard settings causes to give the non real time tasks a chance in 5% of the CPU time, if a real time task would lock the whole CPU. The described modification has the effect that the real time tasks get the whole CPU time and so can eventually lock the system so that no more non real time task can run. Further details can be found in the Kernel documentation [3].

3 Results

In the following sections we list the results and descriptions of the used test systems. The following histograms show the latency and multiplication times in micro seconds on the X axis and the number of occurrences on the Y axis in a logarithmic scale.

The following figures (Figure 2, 4, 6, 8) with the multiplication times show the variation of calculation time in the different real time tasks and illustrate the influence of the thread priority.

3.1 Testsystem Z530

3.1.1 Description

The test system Z530 is based on a Kontron mainboard with an Intel(R) Atom(TM) CPU Z530 clocked with 1.60GHz and two giga bytes of memory.

3.1.2 Result

The test plc takes about 80 % of the whole CPU performance. In this case the latency times vary over more than 100 micro seconds, the peaks look like serialized tasks in order to the configured priority. Maybe one of the main reasons for this picture of latency times in Figure 1 is the poor performance and the missing ability for parallel computing. The first three, highest priority tasks do not exceed the maximum worst case latency of 300 micro seconds, the 2000 and 100000 micro second task exceed the worst case latency of 300 micro seconds very often, as shown in the histogram on the right peaks where the overruns are cumulated.

3.2 Testsystem T7500

3.2.1 Description

The test system T7500 is based on a Kontron mainboard with an Intel(R) Core(TM)2 Duo CPU T7500 clocked with 2.20GHz and three giga bytes of memory.

3.2.2 Result

The test plc takes about 15 % of the whole CPU performance. In this case the latency times as shown in Figure 3 do not vary much and concentrate within 10 micro seconds. In this plot we see the ability of parallel computing, the two fastest tasks set their cpu affinity so that each of them use one core.
3.3 Testsystem D525

3.3.1 Description

The testsystem D525 is based on a Gigabyte GA-D525TUD mainboard with an Intel(R) Atom(TM) CPU D525 clocked with 1.80GHz and two giga bytes of memory.

3.3.2 Result

The testplc takes about 32 % of the whole CPU performance. In this case of Figure 5 the latency times cover a range up to 200 micro seconds. Only a view peaks are detected which exceed the 300 micro seconds.

3.4 Testsystem CP255

3.4.1 Description

The testsystem CP255 is based on a KEBA mainboard with an Intel(R) Pentium(R) M processor clocked with 1.40GHz and one giga bytes of memory.

3.4.2 Result

The testplc takes about 25 % of the whole CPU performance. In this case the latency times in Figure 7 of the three highest priority tasks do not vary much and concentrate within 30 micro seconds. The lower the priority the higher the variation in the latency time, but even the lowest priority tasks latency so far stays below 150 micro seconds.
The results in Figure 7 were produced using a vendor specific rt-kernel based on version 2.6.33.9.

3.5 Testsystem OMAP4

3.5.1 Description

The testsystem OMAP4 is based on an OMAP4 Panda board with an ARMv7 Processor rev 2 (v7l) processor clocked with 1.00GHz and one giga bytes of memory.

3.5.2 Result

The testplc takes about 75% of the whole CPU performance. In Figure 9 the latency times of the three highest priority tasks vary much and reach 300 microseconds as well as the lower priority tasks. This behaviour shows that the realtime capability is not given for this testsystem.

The result in Figure 9 must be accepted with reservation, as the version of the realtime patch and kernel configuration for the used operation system was unknown.

4 Conclusion

The results of the recorded histograms show the different real time behaviours of the evaluated hardware platforms. For the Z530 and OMAP testsystem the real time capability can’t be confirmed as the latency time shows a variation more than 300 micro seconds. In contrast to the testsystem Z530 the histograms from the testsystem T7500 show a very small range of latency variation of only 40 micro seconds for the tasks with the lowest priority. Although the testsystem CP255 is optimized for realtime, the results are not as good as the values from testsystem T7500. We assume the cpu performance is the reason for this behaviour. The four higher priority tasks show only a variation of 15 micro seconds. As the implementation considers the requirements of a real machine control and the evaluation gives a good overview about the realtime capability, we consider the acquired evaluation method as an extension to the OSADL QA farm [2] evaluation.
Evaluation of RT-Linux on different hardware platforms for the use in industrial machinery control

References


openPOWERLINK in Linux Userspace: Implementation and Performance Evaluation of the Real-Time Ethernet Protocol Stack in Linux Userspace

Wolfgang Wallner
wolfgang.wallner@br-automation.com

Josef Baumgartner
josef.baumgartner@br-automation.com

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H
B & R Strasse 1, 5142 Eggelsberg, Austria

Abstract

The RT-Preempt patch for Linux turns the Linux operating system into a real-time operating system and therefore is an ideal platform to implement a real-time Ethernet protocol stack like openPOWERLINK. The initial implementation of the openPOWERLINK stack on X86 Linux was developed as a kernel module. The solution completely bypasses the Linux network stack and achieves maximum performance through the usage of its own network interface drivers. However, this limits the protocol stack to the few openPOWERLINK network interface drivers currently available and also makes the protocol very dependent on the used kernel version. To circumvent these drawbacks, the whole protocol stack was implemented as Linux user space library. As most of the necessary real-time features are also available in user space and many applications do not need the performance level of the kernel space implementation, this solution is adequate for a lot of applications.

This paper describes the porting of the openPOWERLINK stack to user space and examines the performance of the user space implementation. Therefore, the influence of the user space implementation on the network jitter and on the generated system load is analyzed and compared with the kernel space implementation. Due to the long term goal to integrate the lower level layers of the openPOWERLINK stack into the mainline Linux kernel, in this paper it is furthermore discussed how the protocol stack could be segmented into a kernel part that would be integrated into the Linux kernel and a user part that is provided as a user space library.

1 Ethernet POWERLINK

1.1 Overview

POWERLINK is a strict, deterministic hard real-time protocol based on Fast Ethernet (100 MBit). It supports time-isochronous data transfer along with asynchronous communication between network nodes. POWERLINK was originally developed by B&R in 2001. In 2002 the protocol was opened to the public and the Ethernet POWERLINK Standardization Group (EPSG) was founded. The EPSG is an organization which drives the further development of the protocol. Ethernet POWERLINK is a patent-free and open standard, and the specification[1] is freely available on the EPSG website. As Fast Ethernet is used unmodified as the physical layer, no proprietary hardware (e.g. ASICs) is needed for an implementation. The POWERLINK network can be built with standard Ethernet hardware. Network switches could be used, but should be avoided, as they have no upper bound for the forwarding delay of frames. Instead of switches, hubs are preferred, because they provide lower latencies and more deterministic behaviour. As all nodes in a POWERLINK
network have to support the timing rules, standard Ethernet devices may not be connected directly to a POWERLINK domain.

1.2 POWERLINK Layer Model

Figure 1 shows the generic Ethernet POWERLINK layer model. The POWERLINK protocol is located at OSI layer 2 and 7 (Data Link Layer (DLL) and application layer). The characteristic timing that is used to circumvent non-real-time attributes of standard Ethernet (mainly CSMA/CD) belongs to the DLL. The POWERLINK specification defines that the CANopen interface is used as application layer. The usage of CANopen as application layer makes it easy to integrate classical CANopen applications to POWERLINK. The CANopen concepts of Device Profiles, the Object Dictionary, Service Data Objects (SDOs), Process Data Objects (PDOs) and Network Management (NMT) are all reused in POWERLINK. This is the reason why POWERLINK is often referred to as "CANopen over Ethernet".

![Diagram](image)

**FIGURE 1:** Overview of the POWERLINK Protocol Layers.

1.3 Communication Principle (Data Link Layer)

A POWERLINK device can be a Managing Node (MN) or a Controlled Node (CN). A POWERLINK network has exactly one active MN (active redundant MNs are possible to increase fault tolerance). The MN regulates the activity on the network. All other active devices in the network are CNs.

Communication in POWERLINK networks happens in cycles. Each cycle starts with the transmission of the Start of Cyclic (SoC) frame by the MN. The SoC frame is sent as a multicast and can be received and processed by all other POWERLINK stations in the network. No application data is transported in the SoC; it is only used for synchronization.

Immediately after transmitting the SoC, the MN addresses each CN in the network with a Poll Request frame (PReq). Each CN responds with a Poll Response (PRes). This frame is sent as multicast and can therefore be received by the MN as well as by all other CNs in the network. Therefore, the PRes can not only send input data from the CN to the MN, but also allows cross-communication among the CNs. Direct cross-communication allows the times for data exchange between stations to be reduced considerably, since the data need not be copied in the MN.

A CN only transmits when it receives a directly addressed request (PReq) from the MN. The MN waits for the response from the CN. This prevents collisions on the network and enables deterministic timing.

A fixed time is reserved in the network cycle for asynchronous data. Asynchronous data differs from cyclic data in that it need not be configured in advance. Asynchronous data is generated on-demand by a POWERLINK station. Examples are visualization data, diagnostic data, etc. One asynchronous frame can be sent per POWERLINK cycle. The CNs can signal the MN in the poll response frame that they would like to send asynchronous data. The MN determines which station is allowed to send, and shares this information in the Start of Asynchronous (SoA) frame. Any Ethernet frame can be sent as an asynchronous frame (ARP, IP, etc.). However, a maximum length (MTU = Maximum Transfer Unit) must not be exceeded.

The most important timing characteristic in an Ethernet POWERLINK network is the cycle time, which is measured between the start of two consecutive SoC frames. The worst case jitter of the cycle time is a quality attribute of the MN. A typical POWERLINK communication cycle is shown in Figure 2.
FIGURE 2: Schematic showing a typical POWERLINK cycle. Notice how the cycle time is measured from one SoC frame to the next.

2 openPOWERLINK

2.1 Overview

openPOWERLINK is an open source implementation of the POWERLINK technology. It was originally developed by SYS TEC electronic GmbH [9] and later released under the BSD license in 2006. The openPOWERLINK project is hosted on the SourceForge website [10].

A main design goal of openPOWERLINK was portability. Current implementations include Linux, Windows, VxWorks, bare-metal devices and more.

2.2 Software Architecture

The software architecture of openPOWERLINK is very similar to the generic POWERLINK architecture as previously shown in Figure 1. A remarkable exception is the strict partitioning in two parts:

- **Kernel part:** The DLL and all layers below, like Ethernet driver or High Resolution Timers (HRTs), are contained in what is called the kernel part. This part contains the time critical modules of POWERLINK.
- **User part:** The CANopen specific modules (Object Dictionary, PDO, SDO, ...) are grouped in this part.

These two parts exchange information through the Communication Abstraction Layer (CAL). The notations kernel part and user part are currently only naming conventions. In the current implementations these two parts are always located in the same memory space. However, this is one of the preparations for future implementations where this two parts are actually split apart.

2.3 Porting openPOWERLINK

To increase portability, platform dependent code is concentrated in a few isolated places. The porting process of the openPOWERLINK stack to a new platform typically consists of the adaption or reimplementation of the following modules:

- **Multi-Tasking:** The openPOWERLINK stack requires some kind of concurrent execution for its modules. On bare-metal devices, this can be done using IRQs, on hosted platforms this is usually implemented using the platform specific thread API.
- **Shared Buffer:** These are message queues, which are internally used by the CAL to connect the kernel and user part of the stack.
- **Ethernet Driver:** In order for the DLL to be platform independent, an interface has defined to access the network. Each platform needs an implementation of an Ethernet Driver (Edrv) module to access the platform specific network interface that uses this interface.
- **Low Resolution Timers (LRTs):** Some parts of the stack need to watch timeouts in the range of milliseconds (i.e. SDO transfer timeout). These timeouts are not critical for real time, and the timers used for these purpose are referred to as Low Resolution Timers.
- **High Resolution Timers (HRTs):** The cyclic transmission of frames is controlled by the HRTs. These timers need to handle time-outs in the micro second range with a desired precision of a few nano seconds. To generate precise isochronous SoC frames, a POWERLINK MN implementation needs a very accurate system timer and low interrupt latencies.

While the first three points are usually straightforward, the last point poses a challenge on many platforms.

3 Userspace Implementation

3.1 Motivation

On the Linux platform, previously the only implementation of the openPOWERLINK stack was completely in kernel space, having only the application code in user space. This implementation is characterized by the following properties:

- Provides high performance and precision
- Requires special Ethernet drivers
- Maintenance burden (not mainline)
- Hard to debug
The performance and precision reached by this implementation are satisfying (cycle times down to 250µs, jitter in the two-digit microsecond range).

However, there are disadvantages: As it needs special device drivers, a new device driver has to be written for every additionally supported Ethernet chip. As these drivers are not part of the mainline Linux kernel, this will increase the amount of maintenance needed to keep them functional. Additionally, this implementation is not suitable for general purpose debugging of the openPOWERLINK stack (kernel space debugging is more difficult). This lead to the idea of porting the stack completely to user space. The result of the porting efforts should have the following advantages:

+ Support for all Ethernet chips by using some kind of standard network interface
+ Less maintenance effort (stable interfaces in user space)
+ Easier to debug than kernel space implementation
+ Still enough performance for many production applications
+ Possible first step to a later kernel space/user space hybrid solution (outlined in Section 5)

3.2 Linux platform overview

Figure 3 shows the general architecture of different openPOWERLINK stack implementations on the Linux platform. The first architecture shows the complete openPOWERLINK stack implemented in Linux kernel space. This implementation is described in detail in [2]. A long term performance and stability test of this implementation is run in the OSADL Realtime QA Farm[7]. The second architecture shows the current port to user space that is based on the pcap library, which is in the focus of this paper. The architecture on the right shows an implementation of the user space stack which uses the openPOWERLINK kernel space drivers. This implementation was developed to examine the influence of the libPCAP interface on the performance and determinism of the system. It is shown for comparison, but will not be further described in the rest of this paper.

FIGURE 3: Overview of different openPOWERLINK stack implementations: (a) Stack in kernel space (b) Stack in user space using pcap (c) Stack in user space using kernel driver (experimental).

3.3 Porting to user space

Section 2.3 sketched to general porting procedure, this section will describe the design decisions that were made for the Linux user space implementation.

3.3.1 Multi-Tasking

The user space implementation is based on the pthread library, which is used to provide concurrent execution of different openPOWERLINK modules.

3.3.2 Shared Buffer

The shared buffers in user space were implemented using plain malloc. This is possible because all parts of the stack not only reside in the same memory space, but also as threads inside the same process. POSIX semaphores and mutexes were used for synchronization between the different threads.

3.3.3 Ethernet Interface

To provide access to the Ethernet interface from user space two possible implementations could be used, either libpcap or raw sockets. Because PCAP based openPOWERLINK Edrv modules are already available for Windows XP and Windows CE, it was also
chosen as the basis for the Linux platform. As libPCAP uses RAW sockets on Linux there is nearly no performance difference.

3.3.4 Timers

Implementations for both LRTs as well as HRTs use the POSIX timer API. Using POSIX timers for the needed HRTs is possible because of the high-resolution timers that were introduced by Thomas Gleixner and Ingo Molnar as part of the Linux kernel since 2.6.16. The new timer system does no longer depend on the periodic tick of the operating system and allows nanoseconds resolution. However, the resolution depends on the available timer hardware of the system. On an Intel X86 architecture there are different clock sources available (hpet, tsc, acpi_pm) which provide a usable timer resolution in the microsecond range. These high-resolution timers can be used to increase the precision of POSIX user space timers, which is exactly what we needed.

A detailed overview of the new architecture is given in the paper Hrtimers and Beyond: Transforming the Linux Time Subsystems[3].

4 Performance Evaluation

4.1 Test description

The different implementations of the openPOWERLINK stack on Linux were configured as MN and used to control a network of up to 30 CNs. While the timing of frames that were sent by the MN were monitored by an external PC, different load scenarios were run on the MN to analyze their influence. To simulate a real world application, each CN was equipped with digital I/O modules. The control application on the MN modified the outputs based on the input values in every cycle.

4.1.1 Hardware wiring

Figure 4 shows the general hardware setup that was used for the performance and precision tests. The most interesting node in the test setup is the MN, which is shown in blue in the drawing. The MN is the Linux PC that was used to compare the different openPOWERLINK implementations. A variable amount of Ethernet POWERLINK CNs was connected to the MN using a standard Ethernet hub. Up to 30 CNs were used, partitioned as 3 daisy chains, each consisting of up to 10 nodes.

The Ethernet POWERLINK network in the figure is highlighted in orange, other connections that are shown in white indicate standard non-real-time Ethernet. As measuring network times using tools like Wireshark on a standard desktop PC suffers from larger jitter in the timestamps of individual frames, a B&R Network Analyzer X20ET8819 was used. The B&R Network Analyzer is equipped with two network ports, one for POWERLINK and one for standard Ethernet. It is able to capture frames on the POWERLINK network and timestamp them with a 20ns resolution. It packs the timestamped POWERLINK frames into UDP packets and sends them onto the Ethernet interface for further analysis. The PC that was used to collect the captured POWERLINK frames and later run statistical analyzes and create test protocol is shown in the upper right corner. To generate network stress, another external device was needed. For this purpose, another Linux PC was used, which is shown in the upper left corner.

4.1.2 Node configuration

Managing Node (MN)

The hardware platform used for the MN was a B&R AutomationPC810[5] (APC810). Besides being designed as mechanically robust for harsh environments, it is not different from a standard X86 desktop platform.

The APC810 used in our tests was equipped with a Intel Core2Duo U7500 dual core processor running
at 1.06 GHz, 1 GByte DDR2 PC2-5300 DRAM and a 40GB hard disk drive. The Intel 945GME chipset contains the Graphics Media Accelerator GMA 950. The APC is equipped with two on-board network interfaces, which use different Ethernet chips. One of these interfaces is based on the Intel 82573L, while the other uses a Realtek 8111B. A third interface was added as a PCI card, based on the Realtek 8139 chip.

These connections were used as follows:

- **Intel 82573L**: Used as POWERLINK interface. This interface was configured to have no IP address, to avoid interference between the POWERLINK and the Linux network stack.
- **Realtek 8111B**: Connected to the corporate network, used for TCP/IP communication.
- **Realtek 8139**: Directly connected to the PC that serves as flood ping generator (used for network stress test).

The operating system used on the MN was an Ubuntu 10.04 LTS (Lucid Lynx). We used the latest stable real time kernel 2.6.33.7-rt30 as listed on the OSADL webpage[8]. The thread priorities were adjusted to the following values (*demo*-*pi*-*console* is the name of the used demo application):

<table>
<thead>
<tr>
<th>Thread</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sirq-hrtimer/0</td>
<td>-81</td>
<td>High-res timer</td>
</tr>
<tr>
<td>sirq-hrtimer/1</td>
<td>-81</td>
<td>High-res timer</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-76</td>
<td>High-res timer</td>
</tr>
<tr>
<td>irq/29-eth1</td>
<td>-71</td>
<td>Network interface</td>
</tr>
<tr>
<td>sirq-net-rx/0</td>
<td>-61</td>
<td>Network handling</td>
</tr>
<tr>
<td>sirq-net-rx/1</td>
<td>-61</td>
<td>Network handling</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-56</td>
<td>Shared Buffer K→U</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-51</td>
<td>Shared Buffer U→K</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-51</td>
<td>Edrv (PCAP)</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-50</td>
<td>Low-res timer</td>
</tr>
<tr>
<td>demo-<em>pi</em>-console</td>
<td>-21</td>
<td>Startup thread</td>
</tr>
</tbody>
</table>

**TABLE 1:** Thread priorities used by the user space implementation.

The priorities of the system threads were increased using the tool *chrt* before the POWERLINK application was started. To adjust the priorities of the stack internal threads, the the API call *sched_priority* was used during run time.

As stated earlier in section 2.3, the interrupt latencies for timerIRQs need to be as low as possible to increase precision. This is the reason why the timer related threads are set to the highest priorities. For the same reason, the priorities of network related threads has been increased. It is important that the real-time related openPOWERLINK threads all have a higher priority than the other system threads. The internal priority relation between the different openPOWERLINK threads is based on the stack architecture (timer threads higher than network threads, thread of kernel-to-user shared buffer higher than user-to-kernel shared buffer, . . .). The Low-res timer threads have the same priority as the system SIRQs, as they are not critical to the real-time behaviour. The startup thread has a very low priority, because it is mainly used for initialization, but has nothing to do during cyclic operation.

**Controlled Nodes (CNs)**

A network of standard B&R POWERLINK bus couplers (X20BC0083[6]) was used as CNs. As the application on the MN should simulate a real world implementation, these CNs were equipped with input and output modules and exchanged new data in every cycle. These CNs were addressed as standard CANopen DS401: Generic I/O modules.

**Network Analyzer**

To generate high precision time stamps for the observed POWERLINK frames, a B&R network analyzer (X20ET8819) was used. This device is equipped with two Ethernet ports. One of these interfaces is used as a pure POWERLINK input port to analyze the received frames. It latches the time of reception with a precision of 20 ns. This information is packed in UDP packets and sent out on the second Ethernet port. On the PC this information can be received and further processed, i.e. to create high precision Wireshark traces. In our case these measurements were evaluated in our test program to measure the SoC jitter.

**Flood ping generator**

A standard desktop PC running Linux was used to create high amounts of network IRQs on the MN by sending flood pings.

**Measurement PC**

Another standard desktop PC running Linux that was used to dump the timing measurements sent by the network analyzer, do statistical calculations on them using GNU R, and create the test reports with *LATEX*.
4.1.3 Load scenario

- **Idle**: The first measurement was done on an idle system as a reference for the different stress tests.

- **CPU load**: For the CPU stress test, the tool cpuburn was used [11]. It is designed to load X86 CPUs as heavily as possible for the purposes of system testing.

- **Hard Disk I/O load**: The tool dd was used to read and write large amounts of data from and to the hard disk drive.

- **USB I/O load**: As for the hard disk, dd was used on an USB drive to produce USB I/O load.

- **Network load**: Heavy network stress was caused by an external flood ping on the first Ethernet interface.

- **Scheduling load**: Heavy process scheduling load was caused by hackbench [12]. It spawns over a hundred processes which are communicating by sending signals to each other.

- **Miscellaneous load**: To cause miscellaneous system load a Linux kernel compilation was started.

4.2 Results

**Precision**

For a comparison between the measured jitter values of the kernel space and the user space implementation, see Figure 5. The influence of the different load scenarios is very similar for both the user space and kernel space implementation. Notice however the different scale: in the range of 100 µs for user space and in the range of 40 µs for kernel space. High scheduling load has the greatest impact on the network latencies on both implementations.

**Performance**

The measured CPU load of the user space implementation on different configurations is visualized in Figure 6. The kernel space and user space implementation are compared in the following table (CPU load is given in percent of a single CPU core):

<table>
<thead>
<tr>
<th>Cycle time</th>
<th>CNs</th>
<th>User [%]</th>
<th>Kernel [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ms</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10 ms</td>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>10 ms</td>
<td>20</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>10 ms</td>
<td>30</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>5 ms</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>5 ms</td>
<td>10</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>5 ms</td>
<td>20</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>5 ms</td>
<td>30</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>2 ms</td>
<td>3</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>2 ms</td>
<td>10</td>
<td>36</td>
<td>10</td>
</tr>
<tr>
<td>2 ms</td>
<td>20</td>
<td>N/A</td>
<td>16</td>
</tr>
<tr>
<td>2 ms</td>
<td>30</td>
<td>N/A</td>
<td>20</td>
</tr>
</tbody>
</table>

**TABLE 2**: Comparison of the CPU load on different network configurations.

As can be seen from these numbers, the CPU load to drive the same network configuration may increase by a factor of 4-5. With the current user space implementation, it was not possible to handle more than 15 CNs with a cycle time of 2 ms. The reason for this limitation is currently unknown and must be further examined.

**FIGURE 6**: CPU load of the pcap based POWERLINK stack in different configurations.

5 Conclusion and Future Work

The measured values of performance and precision of the user space implementation are inferior to the kernel space variant, which was expected. While high performance application still need to be served by the kernel space implementation, the experiments have shown that the user space variant can be used for many applications with lower requirements. A noticeable benefit of the user space implementation is the portability. Through the use of the pcap library it can be used on any Ethernet chip that is supported by the mainline Linux kernel. In combination with
the RT-Preempt patch this implementation can be used to turn any standard X86 Linux box into a master for real time industrial networking based on Ethernet POWERLINK.

A possible and preferred future stack architecture is shown in figure 7. We would like to submit the time critical parts (mostly the Data Link Layer (DLL)) directly into the mainline kernel. The non-time critical parts (mainly CANopen) could be implemented and distributed as a user space library. This setup would greatly reduce the amount of time and effort needed to turn a standard Linux installation into a hard real-time network master while still providing high performance.

The openPOWERLINK stack has already been prepared for the use as a kernel space/user space hybrid solution and many parts of the needed infrastructure are already in place. However, before we can finally split the two parts, more work needs to be done. There is ongoing effort in the openPOWERLINK community to realize the described architecture.

Additionally some enhancements in the Linux network stack architecture are needed for a high-performance POWERLINK stack. It is necessary to be able to capture and insert Ethernet packets with low latencies. These topics were already outlined by Thomas Gleixner in the document *Powerlink - Linux kernel support*[4]. As these functions are of general interest there is a good chance that they will be implemented in the mainline kernel.

![Figure 5: Boxplots showing the measured jitters values during different load scenarios. (a) shows the jitter values of the kernel based implementation, while (b) shows the results of the user space version using pcap.](image)

![Figure 7: Architecture of the preferred future implementation: Time critical parts included in the mainline Linux kernel (using standard network drivers), while higher layers are implemented and distributed as a user space library.](image)
References


Timing Analysis of a Linux-Based CAN-to-CAN Gateway

Michal Sojka¹, Pavel Píša¹, Ondřej Špinka¹, Oliver Hartkopp², Zdeněk Hanzálek¹
¹Czech Technical University in Prague
Technická 2, 121 35 Praha 6, Czech Republic
{sojkam1, pisa, spinka, hanzalek}@fel.cvut.cz
²Volkswagen Group Research
Brieffach 1777, 38436 Wolfsburg, Germany
oliver.hartkopp@volkswagen.de

Abstract

In this paper, we thoroughly analyze timing properties of CAN-to-CAN gateway built with Linux kernel CAN subsystem. The latencies induced by this gateway are evaluated under many combinations of conditions, such as when traffic filtering is used, when the gateway is configured to modify the routed frames, when various types of load are imposed on the gateway or when the gateway is run on different kernels (both rt-preempt and vanilla are included). From the detailed results, we derive the general characteristics of the gateway. Some of the results apply not only for the special case of CAN-to-CAN routing, but also for the whole Linux networking subsystem because many mechanisms in the Linux networking stack are shared by all protocols.

The overall conclusion of our analysis is that the gateway is in pretty good shape and our results were used to support merging the gateway into Linux mainline.

1 Introduction

Controller Area Network (CAN) is still by far the most widespread networking standard used in the automotive industry today, even in the most recent vehicle designs. Although there are more modern solutions available on the market [1, 2] (such as FlexRay or various industrial Ethernet standards), CAN represents a reliable, cheap, proven and well-known network. Thanks to its non-destructive and strictly deterministic medium arbitration, CAN also exhibits very predictable behavior, making it ideally suited for real-time distributed systems. Because of these indisputable qualities, it is unlikely that the CAN is going to be phased out in foreseeable future.

The maturity of CAN technology means that there exist a huge number of CAN compatible devices on the market. It is therefore quite easy to build prototypes of complex systems by just connecting off-the-shelf devices and configuring them to do what it desired. Sometimes, however, there are devices (or complete subsystems) that are not compatible with each other. They may use different protocols or simply use fixed CAN IDs, that collide with other devices. Then, it is necessary to separate those devices using a gateway that ensures that only the traffic needed for communication with the rest of the system passes the gateway, perhaps after being modified to not collide with the rest of the system.

For this reason a widely configurable CAN gateway has been implemented inside the Linux kernel, which is based on the existing Linux CAN subsystem (PF_CAN/SocketCAN [3]) and can be configured via the commonly used netlink configuration interface. This gateway is designed for CAN-to-CAN routing and allows frame filtering and manipulation of the routed frame content. Obviously, such a gateway must satisfy very strict real-time requirements, especially if it connects critical control systems. Therefore, the gateway had to undergo a set of comprehensive tests, focused on measuring latencies intruded by the gateway under various conditions.

¹CAN uses the term “frame” for what other networks call packet or message.
The results of this testing are reported in this paper. The complete data set, consisting of gigabytes of data and more than one thousand graphs, as well as the source codes of our testing tools, are available for download in our public repositories [4, 5]. This allows other people interested in this topic to independently review our results and methods, as well as to use them as a base for their own experiments. Our methods and results are relevant not only for the special case of CAN-to-CAN routing but, since Linux networking subsystem forms the core of many other protocols, also for other networks including Ethernet, Bluetooth, Zigbee etc.

The paper is organized as follows: the next section describes the setup of our testbed and how we measured the gateway latencies. Section 3 summarizes the main results found during our testing. We give our conclusion in Section 4.

2 Testbed Setup

The testbed, used for gateway latency measurements, is depicted in Figure 1 and consists of a standard PC and the gateway. The PC is Pentium 4 running at 2.4 GHz with 2 GB RAM, equipped with Kvaser PCI quad-CAN SJA1000-based adapter. The gateway is an embedded board based on MPC5200B (PowerPC) microcontroller running at 400 MHz. There are two CAN buses that connects the PC with the gateway. The PC generates the CAN traffic on one bus and looks at the traffic routed via the gateway on the other bus. The gateway is also connected to the PC via Ethernet (using a dedicated adapter in the PC). This connection serves for booting the gateway via TFTP and NFS protocols, for configuring it via SSH, and also to generate Ethernet load to see how it influences the gateway latencies.

The software configuration is kept as simple as possible in order to make the results not disturbed by unrelated activities. The gateway runs only a Linux kernel, a Dropbear SSH server and, obviously, the gateway itself. On the PC, a stripped-down Debian distribution is used. The tasks that generate the test traffic and measure the gateway latency are assigned the highest real-time priority and their memory is locked in order to prevent page-faults. SocketCAN was used on both the gateway and the PC as the CAN driver.

2.1 Measurement Methodology

To measure the gateway latency, we generate CAN traffic in the PC and send it out from can0 interface. As can be seen in Figure 1, this interface is directly wired to the can1 interface of the PC as well as to one interface of the gateway. The can1 interface is used to receive the frames to determine the exact time when each frame actually appears on the bus. This is necessary in order to exclude various delays such as queuing time in the can0 transmit queue. When a frame is received on can1 interface, it is timestamped by the driver in its interrupt handler. These timestamps are sufficiently precise for our measurements.

The frames routed through the gateway are received on can2 interface of the PC. Again, these frames are timestamped the same way as was described in the previous paragraph. The total latency is then calculated by simply subtracting the timestamps measured on the can2 and can1 interfaces (see Figure 2). It is worth noting that both timestamps are obtained using the same clock (in our case timestamp counter register of the PC’s CPU), which ensures that the results are not influenced by the offset of non-synchronized clocks.

To calculate the latency, we need to determine which received frame corresponds to which transmitted one, and this mechanism must be able to cope with possible frame losses or frame modifications in the gateway. For this purpose, the first two bytes of the data payload are used to store a unique number that is never modified by the gateway. This number serves as an index to a lookup table, which stores the timestamps relevant to the particular frame. This allows for easily detection of frame losses. When the corresponding entry in the lookup table contains just one timestamp after a certain timeout, which is set to 1 s by default, the frame is considered lost.

We could also use kernel provided TX timestamps for this, but it somehow didn’t work in our setup.

FIGURE 1: Testbed configuration.
2.2 What Was Tested?

Our goal was to measure the properties of the gateway under a wide range of conditions. These included:

1. The **gateway configuration** such as frame filters, frame modifications, etc.

2. **Additional load** imposed on the gateway system. The following types of load were considered: no load; CPU load i.e. running **hackbench**\(^3\) on the gateway; Ethernet load i.e. running **ping** -f -s 60000 -q **gw** on the PC with **gw** being the IP address of the gateway.

3. **Type of CAN traffic.** We tested the gateway with three kinds of traffic: **One frame at a time**, where the next frame was sent only after receiving of the previously sent frame from the gateway; **50% bus load**, where frames were sent with a fixed period which was equal to two times the transmission duration and finally, **100% bus load** (flood), where frames were sent as fast as possible.

4. **Linux kernel version** used on the gateway. The following versions were tested: 2.6.33.7, 2.6.33.7-rt29, 2.6.36.2, 3.0.4 and 3.0.4-rt14.

We run all experiments for all possible combinations of the above conditions which resulted in 653 experiments. The interested reader can find the full set of the results at [5]. The most important findings are discussed later in this paper.

2.3 Presentation of Results

In every experiment we measured the latency of multiple (in most cases 10000) frames. The results are presented below in a sort of histogram called “latency profile”. Figure 3 shows how the latency profile (at the bottom) relates to the classical histogram (at top). In essence, latency profile is a backward-cumulative histogram with logarithmic vertical axis.

![Diagram](image.png)

**FIGURE 2:** *Calculation of gateway latency.*

Since we are interested only in the latency introduced by the gateway (see GW latency in Figure 2), we subtract from the total latency the duration of the frame transmission, where we take into account the stuff bits inserted by CAN link layer.

The sources of our testing tools and the individual test cases can be found in our git repository [4] under **gu-tests** directory.

4. We used analyzer called CANalyzer (http://www.vector.com/vi_canalyzer_en.html)
method 2 (CANalyzer) requires a lot of manual work to save and process the measured data. The results of the comparison can be seen in Figure 4. It can be observed that the time resolution of CAN analyzer is only 10 µs while the PC was able to measure data with far better resolution, thanks to the support of high resolution timers in Linux. Our histogram uses bins of 1 µs. The difference between the two methods is most of the time below 10 µs. Occasionally (for less then 0.01% of frames), we got a bigger difference. Such precision is sufficient for our experiments.

FIGURE 4: Comparison of total latency measurement by PC and by CANalyzer, payload: 4 bytes.

Since the PC not only timestamps the received frames but also generates the Ethernet load to artificially load the gateway, this additional activity influences the precision of the measurements. To see how big this influence is, we ignored the frames received on can2 interface and calculated the latency \( l \) from RX timestamp on can1, the time before the frame was sent (in user space) to can0 and TX duration (i.e. \( l = t_{RX} - t_{send} - t_{duration} \)). The ideal result would be a vertical line at time 0, i.e. all frames were received immediately after being transmitted, but in reality (see Figure 5) we get a sloped line around 31 µs, because the measurement includes the overhead of sending and receiving operation. The second line in the graph shows, the generating the Ethernet traffic decreases the measurement precision by approximately 30 µs, if we ignore a few out-layers, or by about 200 µs if we do not ignore them. Both number are far below the increase of gateway induced latency (which is in order of milliseconds – see Section 3.3) and therefore, the precision is sufficient even when the PC generates the Ethernet load.

FIGURE 5: Influence of Ethernet load generator on measurement precision (no GW involved).

3 The Results

In this section we present the main results of the analysis. Some of the results gives a nice insight in how networking in Linux works and how Linux schedules various activities.

3.1 Simple gateway

In the first experiment we measured the behavior of the gateway which simply routes all frames from one bus to another without any modifications.


The latency profile and the corresponding time chart is shown in Figure 6. It can be seen that the best-case gateway latency is about 35 µs and the worst-case is about 140 µs. Most of the observed latencies fall into two groups, which are split by a gap around 50 µs. We attribute this gap to timer interrupts which were triggered during processing the frame in the gateway. It can be seen that the cost of the timer interrupt is about 20 µs.

3.2 Batched Processing of Frames

Linux kernel processes the incoming CAN frames (or packets in Ethernet networks) in batches. Basically, when RX soft-irq is scheduled, it runs in a loop and tries to process all frames sitting in receive buffers (either in hardware or in software). The graph in Figure 7 shows nicely the effect of this. If we compare the latencies when the CAN traffic was generated with one frame at a time and flood methods, it can be seen that in the former case, the overhead of scheduling the RX soft-irq is always included (the latency profile starts at 35 µs), whereas in the latter
case, the overhead is reduced. Whenever the gateway receives a frame just when it finishes processing of the previous frame, it does not exit the soft-irq and continues processing the new frame. Therefore, the best-case latencies are much lower in that case (the latency of 0 is of course caused by measurement inaccuracies). The worst-case is about the same in both cases.

3.3 Effect of Loading the Gateway

Figure 8 shows the effect of loading the gateway. The No load line is the same as in the graphs before. The CPU load line represents the case when the CPU of the gateway was heavily loaded by doing many inter-process communications in parallel (hackbench). This approximately doubles the worst-case latency from 140 µs to 250 µs. The Ethernet load (flood ping), however, influences the gateway much more significantly. As it was shown in [6], this is due to the shared RX queue for both CAN and Ethernet traffic. Therefore, processing of CAN frames has to wait after the Ethernet packets (in our case big ICMP echo requests) are processed.

3.4 Frame Filtering

The SocketCAN gateway allows for filtering the frames based on their IDs. There are two kinds of filter implementations. First implementation (used for all EFF\(^5\) frames) puts the filtering rules into a linked list. Whenever a frame is received, this list is traversed and when a match is found, the frame is routed to the requested interface. The second implementation is optimized for matching single SFF\(^6\) IDs. Since there is only 2048 distinct SFF IDs, the filter uses the frame ID as an index to the table and the destination interface if found without traversing a potentially long list.

The top graph in Figure 9 show the cost of having a different number of EFF filters in the list and only the last one matches the sent frames. The gateway latency obviously increases with the number of filters. From a more detailed graph [5] it was nicely visible when the list started to be bigger than CPU cache and the latency started to increase quicker. In our case this boundary was hit for about 80 filters.

Additionally, when the filter list is too long and CAN frames arrives faster, the gateway is no longer able to handle all of them and starts dropping them. This is visible in the appropriate graph at [5].

The bottom graph in Figure 9 shows that the single ID SFF filters perform the same for all frame IDs even when there is 2048 distinct filtering rules.

\(^5\)Extended Frame Format
\(^6\)Standard Frame Format
3.5 Frame Modifications

Besides routing the frames, the gateway is also able to modify them. There are different operations which can be applied to the frames: AND, OR, XOR, SET and two checksum generators. The graph in Figure 10 shows the cost of modifying the frames. In essence, most of the cost comes from copying the socket buffer before modifying it. The difference between different modifications is negligible.

![Figure 10: The Cost of Frame Modification. Conditions: GW kernel 2.6.33.7, traffic: one frame at a time, load: none, payload: 8 bytes.](image)

3.6 Differences between Kernels

Figure 11 shows the differences between different kernel versions. We tried to keep the configs of our gateway kernels as similar as possible by using `make oldconfig`. From the graph, we can observe two things. First, with increasing non-rt versions, the latency increases as well. The difference between 2.6.33 and 2.6.36 is about 10µs, the difference between 2.6.36 and 3.0 is smaller – about 2µs.

![Figure 11: Different kernels. Conditions: traffic: one frame at a time, load: none, payload: 4 bytes.](image)

Interestingly, 3.0-rt is much better in this regard than 2.6.33-rt. Compared to non-rt kernels, 3.0-rt increases latencies by about 20µs, whereas 2.6.33-rt increases them by almost 100µs.

![Figure 12: Huge latencies in rt-preempt kernel. Conditions: 2048 EFF filters, GW kernel: 3.0.4-rt14, traffic: one frame at a time, load: none, payload: 2 bytes.](image)

There seems to be a bug in rt-preempt kernels. For certain workloads, we get the latencies as big as 50ms. This can be seen in Figure 12. Interestingly, we get such a high latency regularly, exactly every one second. This behavior appears in both -rt kernels tested and we will try to find the source of the latency later. Additionally, kernel 3.0.4-rt14 hangs with heavy Ethernet load, which is also something we want to look at in the future.

3.7 User-Space Gateway

The SocketCAN gateway is implemented in kernel space. It is interesting to see the differences when the gateway is implemented as a user space program. The user space gateway was executed with real-time scheduling policy (SCHED_FIFO) with priority 90.

![Figure 13: Kernel-space vs. User-space gateway. Conditions: kernel: 2.6.33.7, load: none, traffic: one frame at a time, payload: 2 bytes.](image)

In Figure 13 can be seen that the time needed to route the frame in user-space is about three times bigger than with the kernel gateway.

The configs of our gateway kernels can be found at https://rttime.felk.cvut.cz/gitweb/can-benchmark.git/tree/HEAD:/kernel/build/shark
It is interesting to compare these graphs for different kernel versions. From that, one can see where the increase of overhead comes from. The interested reader is referred to our web site [5].

4 Conclusion

This paper presented the timing analysis of a Linux-based CAN-to-CAN gateway and studied influence of various factors (like CPU and bus load, kernel versions etc.) on frame latencies. The results indicate that the gateway itself introduces no significant overhead under real-life bus loads and working conditions and can reliably work as a part of a distributed embedded system. Our results were used to support merging the gateway into Linux mainline. The gateway should appear in Linux 3.2 release.

On the other hand, it must be noted that especially excessive Ethernet traffic or improperly constructed frame filters can lead to significant performance penalties and possible frame losses. The CAN subsystem, which forms the core of the examined CAN gateway, is inherently prone to problems under heavy bus loads, not only on CAN bus, but also on other networking devices, as was already demonstrated in our previous work [6]. Nevertheless, the described gateway is a standard and easy-to-use solution, integrated in Linux kernel mainline, and therefore represents the framework of choice for most developers.

It was also clearly demonstrated that the kernel-space solution works much better than the user-space solution, and that it can be beneficial to use standard non-rt kernels (providing that the gateway runs in kernel-space). This allows to avoid greater overhead and resulting performance penalty of rt kernels, providing that the standard kernel is properly configured.

Finally, our benchmarks revealed a few problems in -rt kernels. We will investigate these problems as our future work.

References


Evaluation of embedded virtualization on real-time Linux for industrial control system

Sanjay Ghosh
Industrial Software Systems, ABB Corporate Research
Bhoruka Tech Park, Whitefield Road, Bangalore, India
sanjay.ghosh@in.abb.com

Pradyumna Sampath
Industrial Communications, ABB Corporate Research
Bhoruka Tech Park, Whitefield Road, Bangalore, India
pradyumna.sampath@in.abb.com

Abstract
Real-time control applications in industrial control systems have long been trusted to run on specially designed dedicated embedded hardware like PLCs, controllers etc. The essential non real-time functionalities of the automation systems like engineering and HMI are separately executed on independent hardware. Over the last decade, advancements in RTOS, technologies such as virtualization and availability of more powerful COTS hardware have enabled the recent trends, where industrial PCs are being proposed to replace the hardware controller units. This paper describes the evaluation of our prototype control system on a general purpose hardware based on Linux rt-preempt. In our setup, the control logic is executed by a runtime control engine alongside the standard engineering framework, standard HMI, data acquisition, EtherCAT based IO and also general PC utility applications on a uni-processor system without impacting the deterministic performance. The paper discusses in detail, our performance evaluation results and the methodologies used in terms of, the test setup, boundary conditions, and the parameters measured under typical load conditions as in real industrial applications.

1 Introduction and Background

The automation industry has seen a trend in the usage of control systems based on the commodity hardware instead of being based on the standard controller hardware. The key advantages this solution provides are in terms of higher flexibility of configuration and ease of customization. In fact, with the availability of high processing and memory capabilities, PCs can significantly out-perform most of the commercial controllers that are being currently used in the industry, in terms of the available hardware resources to perform tasks. Therefore, control applications running on a resource rich PC-based control may consume a comparatively lesser of the total available CPU than in the case of a traditional controller, leaving a significant resource left over to run other essential non real time applications. Additionally, PCs are available in several configurations and form factors to meet the diverse user needs compared to the controller hardware. The use of commercial PC along with desktop operating systems does not guarantee the deterministic hard real-time requirements for discrete manufacturing. Hard real-time controllers, by their nature are meant to provide fast, deterministic and repeatable scan times without being affected by the other background activities undertaken by the operating system. Typical discrete manufacturing applications require deterministic, repeatable scan times to be as fast as one millisecond. Therefore the control engineering community, as they move to the PC-based control, expects it to deliver similar level of performance and reliability. The challenge yet remains for the PC based control
on one hand to enable all the benefits discussed above and on the other hand to achieve the reliability required of a controller.

1.1 Related Work

Some previous works have been reported in the literature which explores the concepts of PC based control. Work by Magro et al. [1] include evaluation of time performances of a soft-PLC, OpenPCS by Info Team running on a native operating system Windows2000 with different configurations and loading. In one of the work [2], it was analyzed that, by carefully tuning the Linux rt-preempt based host hypervisor, and using hardware assisted virtualization along with a device emulation application, sub-millisecond scheduling latencies inside guests can be achieved. In fact, Wurmsdobler [3] in one of his article concludes based on their extensive testing that, for slow processes, Linux supports hard real time without any changes. There have been many real-time benchmarking studies on native operating systems [4], [5], [6] to observe the performance and capabilities of the real time patches. Most of these works conclude that; with appropriate tuning of real time priorities and also employing appropriate kernel configurations, appreciable real time behavior could be achieved, to support industry grade real time application demands. Schild et al. [7] studied the interrupt-response times of a real-time operating system, Linux rt-preempt, hosting virtual machines using hardware assisted virtualization technology. A mechanism called .coscheduling., [8] i.e. dynamically boosting the guest VM.s priority levels, is proposed in order to improve the CPU throughput of a general purpose OS VM on a RTOS host.

In this paper we present the evaluation of our prototype PC based control based on Linux rt-preempt to run real-time control. This additionally allows Windows environment to run on the same hardware thus providing the best of both worlds. Section 2 discusses the requirements for the PC based controller from the perspective of the factory automation domain and describes our prototype in detail. Section 3 describes the evaluation setup based on our prototype PC based controller. Section 4 presents the evaluation result and our observations. Section 5 describes the related work. Section 6 concludes the paper by summarizing our findings and also a note on our vision for future work.

2 Industrial Control on commodity hardware

2.1 Requirements for PC based control

The key high level requirement for a PC based control is to run the control engine and the IO communication as user space tasks in a real time operating environment with a real time execution guarantee. In addition to this, the non real time applications like the application engineering tool, HMI, monitoring program and PC utility applications should be able to run on the same hardware without jeopardizing the real time response. Generally, all the application engineering tools as well as HMI interfaces are packaged as Windows based application. The need for Windows based engineering stations stems from existing knowledge base of using windows for such applications. Furthermore, the user should be able to perform shift reporting, batch reporting etc using the general PC utility applications (mostly Windows based) like Microsoft Office. In such context, we try to define the requirements of a system that is capable of providing an environment to achieve the above functionalities. The control engine [9] executes inside an independent execution environment as a user space application on the operating system and fully relies on the underlying OS to provide infrastructure to ensure determinism.

2.1.1 The requirements on control tasks

There is a clear difference in the resource requirements of the different real time and non real time tasks that are required to be present on the PC based hardware. The execution behavior of the control tasks is usually short or periodic [11]. The control engine executed in a ‘scan cycle’, wherein, it performs input, executes the control logic, and ultimately produces output. One full machine cycle of the control is divided into the execution cycle times and the slack times both adding up to make the fixed interval cycle time. The control engine executes the cycle as fast as possible, and the worst-case loop time determines the response time of the system [3].

2.1.2 The requirements on non-realtime tasks

For factory automation processes, HMI especially is required to be responsive to ensure an expected level of user experience. However, these applications are
scarcely needed during operation of a factory automation system. The resource demands of the control engine execution and should not starve the other non real time tasks, especially the engineering and the HMI.

2.2 Technology alternatives for PC based control

In order to host the control engine we identified and selected Linux rt-preempt [12] over other available real time extensions to the Linux kernel such as RTAI and RT-Linux, mainly based on the requirements for our domain [13]. Other reasons for choosing rt-preempt is that a lot of the features in this patch is has been already included into the mainline kernel in parts. It is definitely important to ensure the availability of the community support that is sustainable over long product life cycles. rt-preempt patch implements real time behavior by allowing nearly the entire kernel to be preempted, with the exception of a few very small regions of code. Further by inclusion of the high resolution timers (hrtimers) hard real-time behavior can be achieved in rt-preempt [14]. Thus the control engine in the PC based control is to be run as user space task in Linux preempt RT with appropriate real-time priorities. In order to achieve co-existence of both the real time and the non real time tasks, a significant degree of isolation in terms of memory space is required. The Linux rt-preempt kernel with its implementation of a virtual memory model provides for this feature with real-time responses. Combining virtualization and real-time gives several use cases for embedded systems. Full virtualization options (such as Intel VT-x) are now commonly available in commodity hardware having x86 [10] architecture.

We also selected Kernel Based Virtual Machine (KVM) [15], [16] which utilizes the hardware virtualization extension to enable the virtualization in our PC based control prototype. Since 2.6.20, KVM as an active open source project has been a part of the mainline kernel as a Linux kernel module with a strong developer community. KVM runs unmodified guest operating systems on the host OS providing each virtual machine to own private virtualized hardware: a network card, disk, graphics adapter, etc. It relies on the host OS for tasks like scheduling, native interrupt, memory management, hardware management, etc. using a kernel-space device driver (/dev/kvm) and hence categorized as type-2 hypervisor. This uses a user-space component QEMU [17] for all device emulation. KVM adds an operating mode in addition to the default, the kernel and the user modes, in Linux called the guest mode, which in turn has its own kernel and user modes [17]. This resulting guest VM's physical memory can be mapped to the virtual memory of the host hypervisor. Using the corresponding host process for the VM guest process, standard configurations in terms of priority, affinity etc can be configured in order to flexibly influence the scheduling of virtual machines during runtime [2]. This is another advantage which a PC based control using a hardware assisted virtualization over a native operating system scores over the traditional control.

2.3 Prototype PC based control

Based on the requirements for the PC based Control mentioned in section 2.1 and considering the technology solution mentioned in section 2.2, we came out with a prototype PC based Control. For the control engine we selected one of the control engine that provides support for Linux operating system on x86 architecture hardware. Programming IDE runs on Windows guest OS and is based on the open international standard IEC 61131 and is also supplied by the same vendor. The control engine also supports visualization on the device where the HMI is executed as a user space process on the host operating system. The control engine executes as a high real time priority on the Linux rt-preempt based host while the engineering and other PC utility applications are executed inside one or many Windows based guests with comparatively lower process priorities. The execution model of the prototype PC based control is shown below in figure-1.

![FIGURE 1: Execution Model](image)

Figure 2 below shows the communication model of the prototype PC based control. The network configuration between the host operating system (Linux rt-preempt) and the guest operating system (Windows) is using the local Ethernet based soft bridge.
(bridge configuration using /etc/network/interfaces
  . auto br0). Communication between the control
engine (also known as the Target Device) executing
on the host and the programming IDE (also known
as the Engineering) executing on the guest is based
on the industry standard OPC Server. We used the
industry standard real time communication protocol,
EtherCAT for IO communication in our prototype.
The EtherCAT master stack executes as a high real
time priority user space process in the host Linux rt-
preempt operating system. This enables the Ether-
cat network interface card of the PC to communicate
to the EtherCAT slave module connected to it using
the EtherCAT protocol. An oscilloscope is used to
display and verify the real time output from the PC
based control.

3 Evaluation Setup

The earlier section described our PC based control
prototype in general. Going further, this section de-
cribes the specific evaluation setup and configura-
tions of the prototype we used for our experiments.

3.1 System and Runtime Environment

The host system in our evaluation setup consisted
of an Intel core two duo processor (Core2 G6950
at 2.8GHz) with 2 GB of RAM memory. The
CPU is x86 architecture based and has support for
hardware assisted full virtualization with the In-
tel VT-x technology. The system had a 1Gbps
Ethernet network interface card. Host was run-
ing the Linux distribution Ubuntu10.04. The host
kernel was built out of 2.6.33.7.2-rt30 kernel with
the rt-preempt patch. CONFIG_PREEMPT, CON-
FIG_PREEMPT_RT and the HRTTIMER were en-
abled and CONFIG ACPI_PROCESSOR was dis-
abled to achieve the best possible latencies. Both
the host as well as the guests used the native TSC
(constant tsc flag in /proc/cpuinfo) as clock source
so that the time measurements obtained within
the guest are reliable. In-kernel tracing functions
i.e. CONFIG_LATENCY_TRACE were built in, but
were kept disabled during the experiments[2]

```bash
echo 0 > /proc/sys/kernel/ftrace_enabled
echo 0 > /sys/kernel/debug/tracing/tracing_
   enabled
```

One of the CPU out of the two in the core two
duo system was kept disabled during the experiments

```bash
echo 0 > /sys/devices/system/cpu/cpu1/online
```

The experimental setup consisted of two non
real time custom created guests with 512MB virtual
RAM and a single VCPU created using Virtual Ma-
chine Manager application. Both of these virtual
machines were based on Windows XP SP3. One
of the guests VM runs the application engineering
while the other was meant to running PC utility ap-
lications. We also disabled the USB legacy option
in system BIOS, as it was reported in some earlier
work[8] that the USB legacy device is one key factor
that causes latencies arising due to SMIs. Instead we
used PS/2 keyboard and mouse. One of the exper-
iments requires running the application engineering
on a separate networked system. The other PC con-
sists of an x86 architecture based Intel Core i5 CPU,
M520, 2.4GHz, 2 GB RAM and running Windows
XP SP3.

3.2 Test Applications

The benchmarking control programs referred by the
PLCopen [18] have been used for performance eval-
uation of the PC based control in adherence to
the standard guidelines from the PLCopen. These scripts are written in IEC 61131 language structured text format [19] and are used as standard programs to benchmark the performance of PLCs. The application engineering interface used in our prototype allows application programming using the standard IEC 61131 language. There are two types of benchmarking scripts, for application oriented tests and for language oriented tests. Application oriented benchmarks are used to measure the whole cycle of the control i.e. from receiving an input signal, the internal processing, till writing an output signal. These are a set of different types of applications and their mixtures, which are typically used in the factory automation. Language oriented benchmarks evaluates the computational performance of a controller while performing all the available language constructs in 61131-3 language. We have evaluated our prototype PC based control using seven of the benchmarking programs including both the application oriented tests and the language oriented tests. Almost similar results were observed for all of these tests. For the sake of conciseness, in this paper, we present only the results of one of the experiments i.e. the language oriented test for the control statements. This test evaluates the performance of the PC based control in operation of one thousand instances of different control statements e.g. IF, CASE, FOR, WHILE etc. Repetition time is coded inside the test project scripts using looping. All the experiments were performed for more than two hours with the control engine continuously executing the test program during this duration with different test conditions. The application engineering tool used in the prototype also allows developing visualization applications too. The visualization program executes on the control engine along with the application test program, however with an execution cycle time, typically, two orders larger than that of the control application. We have created a simple visualization program which shows six visualization objects on the screen and linked to these to monitor and update the status of six different variables (i.e. six OPC tags) in the application test program.

While performing the performance evaluation of any real time systems, it is a usual practice to load or stress the system under test in order to observe the performance in such conditions. In order to stress the Linux rt-preempt host, we used the open source tool ‘Stress’ 1.0.4. [20] to run as a background process in order to imposes a configurable amount of CPU, memory, I/O, and disk stress on the system. An example of the command line used for executing the stress program is as follows

stress -cpu <c> -io <i> -vm <m> -timeout <t>&

Similarly, in order to load or stress the Windows based guest VMs, Windows based freeware HeavyLoad 3.0.0.159 [21] was used. In order to stress the system resources, HeavyLoad writes a large test file to the temp folder, allocates physical and virtual memory and draws patterns in its window.

3.3 Evaluation Parameters

For the performance evaluation of the PC based control, we mainly focus on measuring the upper bound or the worst case latencies of the cyclic execution of the control application. The key to appropriate performance evaluation is accurately measuring the cycle time. The actual cycle time is the time span between start and end of a test cycle excluding any possible overheads such as task startup process, IO access time etc. However, system specific overhead (like timer tick and task scheduler) gets included by default to the cycle time measurement. Therefore the time measurements are instrumented within the application test program. At every iteration cycle, at the start of the operations, the current system timestamp is stored in a variable. Then the operations are being performed according as per the application logic. Right at the end of all the operations, the system time is stored again and the time elapsed while executing the operations is being calculated. This is termed as the Execution Cycle time or more commonly Cycle Time. Based on these measurements per cycle, average, minimum and maximum execution cycle times is calculated. Another measurement parameter which is of interest is the Jitter in the execution of the control logic. This is the measure of how early or how late the execution cycle starts with reference to the desired time of start. Prior to performing the actual performance evaluation experiments, as the first step, it is also important measure the processing capabilities of a control system. In order to estimate the processing capability of the control system for the particular control logic, we execute the test control program with different interval cycle times and the watchdog enabled. Watchdog in this context is defined as a monitor inbuilt within the runtime engine which indicates an exception when the actual execution time of the IEC61131 application exceeds the designated interval time. That is estimated as the minimum possible interval cycle time of the system under test for that particular control application. Further, for all the performance evaluation experiments on the system using that test control application, the interval cycle time is configured to be the minimum
As mentioned in the description of our evaluation prototype, an oscilloscope is connected to the IO module in order to display and verify the real time output from the PC based control. The IO cycle time (also known as the scan time in the control engineering), for the purpose of all the experiments is kept constant and it is equal to the interval cycle time. Our experimental prototype consists of real time processes such as the control engine and EtherCAT IO master stack; and non real time processes such as a guest VM for running the application engineering tool and another guest VM for running the PC utility applications. Among all the user space processes running on the host system, the control engine process is assigned the highest real time priority followed by the EtherCAT IO master stack process.

In order to limit the size of the test matrix, we decided to focus only upon few specific aspects of the performance evaluation and hence we have fixed few of the variable parameters for all the experiments. For all the experiments we have fixed the real time priorities of the control engine process and the EtherCAT IO master stack process as 68 and 50 respectively. As mentioned in the earlier section, the stress program is used to produce controlled load on the host Linux system. The stress program continuously runs for the whole duration of the experiment with a constant value of the load. Value for the last 15 minutes average load from /proc/loadavg is considered to be the measure of the constant system load. Based on this measure, for the purpose of our experiments, we have defined the categories of the system load level as Light < 2, Moderate 5, Heavy 10 and Very Heavy <= 20. The reference numbers mentioned in the parenthesis approximately represents the /proc/loadavg values for each of the categories. According to the standard guidelines from the PLCopen each of the standard benchmark control programs needs to be repeated at least 10,000 times to get accurate results. We ran all our tests for a longer duration of at least two hrs each. If the interval cycle time be 1 millisecond, then during the duration of a test, the control application would run > 7x10^6 execution cycles, which is typically large enough to observe the maximum latencies.

4 Evaluation Results and Discussions

The objective of the performance evaluation is to measure the worst case latencies and jitters observed while executing the control applications on the PC based control. Also this gives the performance of the control applications and real time IO communication, when the engineering application interface, HMI and PC utility applications all run on the same hardware.

4.1 Test 1: Interval time evaluation

This experiment was performed in order to approximately measure the performance of the PC based control in terms of the minimum possible interval cycle times it can support for the execution of a control logic. For the PC based control, a comparative measurement was performed among the two possible scenarios. Scenario-1, when the application engineering tool, visualization as well as PC utility applications run on the same hardware with the control engine; scenario-2, when all these applications run on a different system. In this experiment, we tested the execution of the system by step wise reducing the interval cycle time of the control logic till the system throws watchdog exceptions. Following were our observations while performing the experiment for both of the above mentioned scenarios.

**Table 1: Execution time evaluation for Scenario-1**

<table>
<thead>
<tr>
<th>Interval(µs)</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Watchdog exception at start</td>
</tr>
<tr>
<td>350</td>
<td>Watchdog exception at start</td>
</tr>
<tr>
<td>400</td>
<td>Watchdog exception at start</td>
</tr>
<tr>
<td>450</td>
<td>Watchdog exception after few minutes</td>
</tr>
<tr>
<td>500</td>
<td>Smooth Execution</td>
</tr>
</tbody>
</table>

**Table 2: Execution time evaluation for Scenario-2**

<table>
<thead>
<tr>
<th>Interval(µs)</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Watchdog exception at start</td>
</tr>
<tr>
<td>350</td>
<td>Watchdog exception at start</td>
</tr>
<tr>
<td>400</td>
<td>Watchdog exception after few minutes</td>
</tr>
<tr>
<td>450</td>
<td>Smooth Execution</td>
</tr>
<tr>
<td>500</td>
<td>Smooth Execution</td>
</tr>
</tbody>
</table>

The measurements presented in the table 2 shows that, the PC based control was able to smoothly execute a particular test control application, when configured to run with interval cycle time of 450µs. This is only when the system runs just the control engine and the IO communication run as real time processes in the Linux rt-preempt host. In another scenario,
where the necessary non real time tasks are also run alongside these real time processes, the smooth execution of the control application is found possible only beyond the interval cycle time of 500µs. However, typical factory automation applications require scan times of as small as one millisecond. In both of the above mentioned scenarios, it was observed that the average execution time the system requires for executing one cycle of the test control logic was less than 225ms, i.e. well below even the 50% of the interval cycle time. As the control engine and the IO communication runs as high real time priority processes, these tasks are never preempted from the scheduler by other non real time tasks, during their execution. However, it is during these available slack times the non real time processes are scheduled if required.

4.2 Test 2: Latency and jitter evaluation

This experiment is meant to evaluate the performance of PC based control prototype for reliably running the control applications and real time IO communication, even in the presence of the necessary non real time tasks, running on the same hardware. For the purpose of this experiment we executed the test control application on the PC based prototype under seven different test conditions identified as the seven setup configurations, C1,C2,.., C7. These configurations are defined based on what all tasks are being run on the system

- Base Configuration (Base): Control engine + IO communication + Visualization on host + Windows VM1 running application engineering tool
- C1: Base
- C2: Base + Low stress on host
- C3: Base + Moderate stress on host
- C4: Base + Heavy stress on host
- C5: Base + Very Heavy stress on host
- C6: Base + Very Heavy stress on host + Max possible load on Windows VM1
- C7: Base + Very Heavy stress on host + Max possible load on Windows VM1 + Windows VM2 running text processing application

Table 3 below presents the measurements of the execution cycle times and jitter for the test control application and scan times for the EtherCAT based output observed using an oscilloscope.

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Control Application</th>
<th>External IO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycle Time (ms)</td>
<td>Interval time Jitter (ms)</td>
</tr>
<tr>
<td></td>
<td>Ave</td>
<td>Min</td>
</tr>
<tr>
<td>C1</td>
<td>219</td>
<td>206</td>
</tr>
<tr>
<td>C2</td>
<td>219</td>
<td>206</td>
</tr>
<tr>
<td>C3</td>
<td>218</td>
<td>210</td>
</tr>
<tr>
<td>C4</td>
<td>220</td>
<td>209</td>
</tr>
<tr>
<td>C5</td>
<td>224</td>
<td>208</td>
</tr>
<tr>
<td>C6</td>
<td>224</td>
<td>209</td>
</tr>
<tr>
<td>C7</td>
<td>229</td>
<td>209</td>
</tr>
</tbody>
</table>

FIGURE 3: Execution Model

The measurements presented in the table 3 shows that, the PC based control was able to accommodate the execution of typical non real time tasks required in a control system, on the same hardware, without compromising the real time guarantee of the control applications. Further, it was also observed that, even by deliberately applying heavy loads on the host system as well as the guest systems, there is only a negligible change on the real time execution behavior of the control application.

In one of the scenario, the reliability of the isolation between the Linux rt-preempt host and the Windows based guests partitions was also evaluated. Even during the deliberate crashing or during re-booting of the Windows guests, the control engine and the real time IO continues to perform unaffected in terms of cycle times and jitter.

5 Conclusion and Future Work

Our experiments with the prototype PC based control device has demonstrated that; it is possible to be able to achieve deterministic responses by using Linux rt-preempt along with KVM as the host RTOS. Irrespective of the concurrently running application load on the windows guests, the deterministic behavior of the user space applications running on the rt-preempt kernel is not affected. The choice of real-time tasks and their priorities must still be carefully managed and the host and guest must still follow the traditional separation of concerns i.e. real-time and non-real time respectively. The results of the tests performed also indicate that, such a system may be conceivable for certain industrial application domains, but maybe inappropriate for applications...
which demand more stringent real-time constraints (such as closed loop motion control).

Going forward we believe that there is potential for further work in the area. One such activity might involve the confluence of multi-core, virtualization and real-time. Comparative studies between SMP Virtualization and AMP virtualization for real-time systems is an area where the industry and academia might see benefit.

References


Turning Krieger’s MCS Lock into a Send Queue
or, a Case for Reusing Clever, Mostly Lock-Free Code in a Different Area

Benjamin Engel and Marcus Völp
Technische Universität Dresden
Operating-Systems Group
Nöthnitzer Strasse 46, Dresden, Germany
{voelp, engel}@os.inf.tu-dresden.de

Abstract
Lock- and wait-free data structures can be constructed in a generic way. However, when complex operations are involved, their practical use is rather limited due to high performance overheads and, in some settings, difficult to fulfil object lifecycles.

While working on a synchronous inter-processor communication (IPC) path for multicore systems, we stumbled over a clever piece of code that did fulfil most of the properties that this path requires for its send queue. Unfortunately, this piece of code was by no means a data-structure publication or somehow related to send queues. Reporting on our experience in translating Krieger’s MCS-style reader-writer lock into a send queue for cross-processor IPC, we would like to make the point that sometimes, searching for code could end up in a valuable treasure chest even for largely different areas.

1 Introduction
Predictability, security and the ease to support application-tailored OS functionalities all speak for microkernels and microhypervisors as host operating systems for todays and future manycore systems. In particular, augmented virtualization environments as we find them in desktop, server and embedded systems benefit from the ability to co-host large legacy software stacks next to more sensitive code such as real-time subsystems [2, 3] and secure applications [4].

Synchronous inter-process communication (IPC) is one of the central mechanisms many of these kernels implement. Reasons favoring synchronous (i.e., unbuffered and blocking) IPC are:

1. asynchronous communication and coordination primitives can easily be built on top of synchronous IPC, however the reverse is not possible in this generality;

2. synchronous IPC is able to exploit preallocated memory buffers, thereby eliminating the need for memory allocation during the message transfer; and,

3. processor local synchronous IPC implementations can be as fast as a few hundred cycles 1, which makes them suitable for higher-level synchronization primitives.

While processor-local IPC not necessarily requires a send queue, it turns out that in cross-processor IPC paths senders have to be blocked at the receiver and processed in some arrival-dependent order. In other words, multiprocessor synchronous IPC needs a send queue to avoid unbounded starvation of senders waiting to rendezvous with their receiver.

This paper reports on our experience implementing such a send queue and the surprising result we found: basically a wait-free MCS-style reader-writer lock by Krieger et al. [7] provides all the essential

---

1288 cycles on an Intel Core i7 920 2.67 GHz [5].
functionality that we required. In the following, we summarize the key requirements of send queues, motivate our choice of a non-blocking implementation and highlight the difficulties of using non-blocking code in the kernel. Then we briefly outline the original reader-writer lock by Krieger and present our modifications to turn it into a send queue. We evaluate our results and relate them to others, before introducing our idea on a treasure chest of non-blocking building blocks.

2 Send Queue Requirements

The primary purpose of a send queue is to order incoming requests to prevent sender starvation. In processor-local IPC paths, the order of send operations is governed by the sequence threads are picked by the scheduler. This completely eliminating the need of a send queue. For example, facilitating time and priority inheritance, servers may receive time from the currently highest prioritized thread blocked on them. Using this time, they can therefore commit potentially pending requests to process the call. Depending on whether these operations are used to block threads only in the case of a contended server or in every IPC, enqueuing into an empty list and dequeuing the head must be fast. In addition, send queues should also support a dequeue operation from the middle of the list, allowing threads to cancel their not-yet-started IPC. Situations where callers have to abort an IPC include the deletion of a thread and timeouts (e.g., set to react on certain error situations).

One of the key benefits of a synchronous IPC path is the possibility to preallocate all message buffers to avoid memory allocation during the IPC. The immediate consequence for the send queue is that all operations have to operate on preallocated memory as well. In particular, if a call is finished, the caller’s message buffer and all meta data used during the IPC must be ready for use in subsequent IPC calls. In particular for generic constructions of lock-free data structures, these properties are difficult to fulfill. Still non-blocking implementations have their benefits: they typically scale to higher CPU numbers than lock-based variants and, in our case, the effort and overhead required for a fair lock protecting the send queue is in the same order as the effort and overhead of a mostly lock-free send queue.

3 Krieger’s MCS-style Reader Writer Lock

By swinging a single pointer to the tail of a list, MCS locks implicitly arrange lock acquiring threads in FIFO order. Either if a thread enqueues into an empty queue (i.e., tail = 0 prior to the enqueue operation) or if the previous lock holder releases the lock by clearing a field the next thread in the list spins on, the thread at the head of the list becomes lock holder.

Eliminating contention on the active reader counter of Mellor-Crummey and Scott's original reader-writer lock [8], Krieger et al. [7] introduce a second pointer to each queue element to enqueue readers into a lazily updated double-linked list. Finished readers dequeue from the middle of the list using spin locks that are only taken for the purpose of

FIGURE 1: Data structure and dequeue operation from Krieger’s MCS lock

The operations required of a send queue are *enqueueing* to the tail of the list and *dequeueing* the head when replying. Depending on whether these operations are used to block threads only in the case of a contended server or in every IPC, enqueuing into an empty list and dequeuing the head must be fast.
protecting neighboring list nodes during a dequeue. The last active reader releases a subsequent writer if present. Figure 1 illustrates this algorithm and the used data structures.

The key features of Krieger’s MCS lock, making it a perfect starting point for send queues, are:

1. an implicit FIFO ordering by threads atomically swinging the tail pointer to their list element and lazily enqueuing afterwards;
2. an extremely fast enqueue operation for the uncontended case (essentially just an atomic swap of the tail pointer);
3. in most situations, an extremely fast dequeue operation of the head element (essentially only the release of the next thread and an atomic compare-and-swap if the queue is empty afterwards); and,
4. the possibility to dequeue from the middle of the queue.

4 Turning Krieger’s Lock into a Send Queue

Our goal is to translate Krieger’s MCS lock into a mostly lock-free send queue for a call-reply style IPC path. In general, there are two principle ways to use such a send queue: enqueue callers only if the callee is contended, or, enqueue all callers that are sending or waiting to send to the callee. In the first case, exclusive control over the callee must be taken to determine whether it is ready to receive new requests. If not, the caller would block in the send queue while waiting for the callee to be ready to receive its request. Otherwise, the caller, in case of a caller-driven implementation, or the callee, in case of a callee-driven implementation, starts transferring the message. However, because locking the callee to obtain exclusive control ideally involves a fair lock that facilitates local spinning, we expect comparable costs for locking the callee and for enqueuing into an MCS-style send queue.

To avoid the additional overhead of locking the callee in the uncontended case (i.e., when the callee is already receiving), we maintain the following invariant:

Invariant 1: A callee with an empty send queue is always receiving and implicitly locked by the first thread entering the send queue.

The first part of Invariant I is ensured automatically by the call-reply style IPC path as long as ongoing requests are not aborted (Section 4.1 below discusses how the spirit of this invariant can be maintained in case of aborts). The second part is ensured by leaving the callee locked in situations where the last thread dequeues itself from the send queue.

\[
\text{type } \text{SQ_Item} = \text{record} \\
\quad \text{next} : ^{\text{sq_item}} \\
\quad \text{pred} : ^{\text{sq_item}} \\
\quad \text{lock} : \text{precedence_spinlock} \\
\text{type } \text{Status} = \text{enum} \{\text{EMPTY}, \text{NOT_EMPTY}, \text{OTHER}\} \\
\text{type } \text{send_queue} = \text{class} \\
\quad \text{head} : ^{\text{sq_item}} \\
\quad \text{tail} : ^{\text{sq_item}} \\
\text{method } \text{enqueue}(I : ^{\text{SQ_Item}}) : \text{Status} \\
\quad \text{pred} = I \\
\quad \text{head} := I \\
\quad \text{release_precedence}(I->\text{lock}) \\
\quad \text{return } \text{EMPTY} \\
\quad I->\text{pred} := \text{pred} \\
\quad // \text{store fence} \\
\quad \text{pred}->\text{next} := I \\
\quad \text{release_clear_precedence}(I->\text{lock}) \\
\quad \text{return } \text{NOT_EMPTY}
\]

**FIGURE 2: Types and enqueue operation**

Figure 2 shows the pseudocode for enqueuing into the send queue\(^2\). Like in MCS locks, the function enqueue starts by atomically swinging the tail pointer of the send queue to the list element of the invoking caller and remembering the old value of tail. If this old value is nil, the queue was empty and the function may return after updating the head pointer. Otherwise, enqueue first sets the predecessor pointer of its element before completing the enqueue operation by updating the predecessor’s next pointer. As pointed out by Krieger, this write sequence prevents a race with a concurrent dequeue operation from the middle of the list and may require an additional fence. The role of the precedence spinlock and the validity of head will be discussed after we have introduced the two dequeue functions.

Krieger’s MCS lock does not distinguish between dequeuing from the middle of the list and dequeuing the head element, since finished readers have to retract from the queue lock no matter where they are. For a send queue however, the former operation is only invoked when a thread cancels its IPC, whereas dequeuing the head is used in every reply to a caller, thereby completing the IPC.

\(^2\)For a better comparison, we adopt the pseudocode introduced by Mellor-Crummey and Scott [8], which is also used in Krieger et al. [7].
method dequeue_head(I : ^SQ_Item) : Status
if acquire_precedence(I->lock) = false
    return OTHER
if I->next = nil
    if !compare_swap(tail, I, nil)
        repeat while I->next = nil
            next : ^SQ_Item := I->next
            if next != nil
                next->pred := nil
                head := next
            I->next := nil
            return NOT_EMPTY
        I->next := nil
        return EMPTY

FIGURE 3: Dequeue head operation

Figure 3 shows the pseudocode for dequeueing the head element. Again we defer the discussion of the precedence spin lock to Section 4.1. Like in all MCS-style locks, a nil next pointer can indicate one of two situations: either the dequeue operation is about to remove the last thread from the list (in this case, tail = I holds) or, a thread is about to enqueue into the list but did not yet manage to update the predecessor’s next pointer. The atomic compare-and-swap checks in which of the two states the list is in and clears tail in the first case. Otherwise, the dequeueing thread spins until the next pointer is set. This spinning is bounded because enqueue executes in the kernel with interrupts disabled. After returning from this loop, either the list is empty or the next pointer is set and the dequeueing thread can update the head pointer to the corresponding thread, now being the new head of the list.

Notice that the head pointer is not always current. In particular, dequeue_head does not update head in case the list gets empty. The invariant that maintains correctness of this implementation is the following:

Invariant II: Head is valid only for the thread that is under exclusive control of the callee.

An immediate consequence of this invariant is, that threads may not poll head until they reach the front of the send queue. The following sequence illustrates this race:

1. Thread A enqueues and dequeues from the list.
   Assuming the list was empty, head now refers to A because dequeue_head will not clear the head pointer, as doing so would race with with B’s concurrent enqueue operation.
2. Thread B starts enqueuing itself but is delayed after updating the tail pointer.
3. Thread A returns, enqueues itself to the list and finds itself to be the head because B did not yet update this pointer after Step 1. If now a thread C enqueues itself and both A and B dequeue themselves using dequeue_head, A will set C’s predecessor to nil but B’s later dequeue will set the already left A as a new head.

Although the list structure itself is maintained, any head dependent action by C will now block forever or work on the wrong head A. For call-reply style IPC paths, the restriction Invariant II imposed on the use of the head pointer is no problem, because head is used only by the thread having exclusive control of the callee and in one of the following two situations: (1) to pull in the next message after a reply; and (2) to identify the thread to reply to. In a callee-driven implementation, the callee is active anyway when these situations occur. In a caller-driven implementation, the exclusive owner of the callee takes over control of the thread at the head of the send queue to push its message to the receiver.

4.1 Cancel

method dequeue_middle(I : ^SQ_Item) : Status
    pred : ^SQ_Item = I->pred
    // chase and lock pred
    repeat while pred != nil
        if try_acquire(pred->lock)
            if pred = I->pred
                break
        release(pred->lock)
        pred := I->pred
        if pred
            acquire_precedence(I->lock)
            prev->next := nil
            if I->next = nil
                if !compare_swap(tail, I, prev)
                    repeat while I->next = nil
                        next : ^SQ_Item := I->next
                        next->pred := pred
                        if next != nil
                            pred->next := next
                        release(prev->lock)
                        I->pred := nil
                        I->next := nil
                return dequeue_head(I)

FIGURE 4: Dequeue middle operation

Figure 4 shows the pseudocode for dequeuing threads from the middle of the send queue, which is required for canceling waiting threads. Except for the precedence spinlocks, the above code directly resembles Krieger’s reader unlock operation. The first while loop chases the predecessor pointer of the dequeuing caller’s list element to lock it for the subsequent dequeue operation. In our code base, this loop is preemption reactive in the sense that it will abort
the dequeue operation if a preemption is pending. We have omitted this test for reasons of simplicity.

Having locked the predecessor, the own lock of the caller’s list element is acquired to prevent concurrent dequeues from modifying the prev and next pointers. Like in all MCS-style locks, compare-and-swap is used to update the tail pointer in situations where the tail element of the list is dequeued. Otherwise, the dequeue operation waits for a pending enqueue operation to update the next pointer of the to-be-dequeued element. Together with enqueue and dequeue_head, dequeue_middle maintains the invariant that dequeued list elements are always precedence locked. In the following we describe the role of these precedence locks in greater detail.

The primary purpose of the spin lock is to protect the prev and next pointers from concurrent dequeues. In our version, we grant dequeue_head precedence over concurrent dequeue operations from the middle of the list, which are invoked as part of an IPC cancel operation and, unlike dequeue_head, are not performance critical. To grant precedence, threads dequeuing from the middle can obtain the lock only if it is free and the precedence bit is clear. Therefore, by setting the precedence bit in case the lock could not be acquired immediately, the callee replying to the caller will obtain the lock after at most the duration of one dequeue operation.

In a callee-driven implementation of synchronous IPC, there is one situation where two threads concurrently attempt to acquire the lock with precedence: when the reply of a callee to its caller collides with an IPC cancel operation to the head of the send queue. In this situation and because dequeuing from the send queue is the last operation of the IPC, it does not matter which one of the threads completes its dequeue operation. Therefore, a thread will bail out from the dequeue_head operation with the status code Other if it finds the lock with the precedence bit set. If these two operations happen in any sequence one after the other, it is important to abort the second operation because otherwise, after the compare-and-swap operation in dequeue_head, the second dequeue_head would wait for a thread to update its next pointer, which will never happen. In Krieger’s MCS lock and likewise in our send queue, there is no means to detect just from the link information whether or not an element is enqueued. The invariant that dequeued elements are precedence locked introduces precisely this information to avoid the above lifelock in subsequent dequeue_head operations.

Because threads dequeuing from the middle may refer to elements that are no longer enqueued, threads and their corresponding send queue items may not be deallocated immediately upon their destruction. Instead, we reuse a read-copy update (RCU) like deferred destruction scheme [9], that was already available in Nova [5].

Although a callee is not necessarily receiving, the spirit of Invariant I holds trivially in a callee-driven implementation because a caller enqueuing into an empty list will simply activate the callee no matter in what state it is. After a cancel, this may result in the callee completing its prior operation or performing some cleanup before entering the receive state during which it will pull the caller’s message.

4.2 Evaluation

To evaluate the performance overhead of our send queue we have measured roundtrip times of the operations involved in a call-reply pair — enqueue followed by dequeue_head if the caller is at the head of the send queue. Measurements were done on an Intel Xeon X5650 @ 2.67 GHz by increasing the number of cores participating in the roundtrip. As seen in Figure 5, the call/reply to an uncontended server on a single core is quite fast, having multiple cores on the same socket competing on a queue adds a overhead of up to 1200 cycles, but having to access the cross-socket interconnect is really painful.

5 Related Work and Treasure Chest

In the literature, one finds several lock-free implementations of common data structures such as lists, stacks and trees [11, 12]. However, typically these
data-structures where designed as a reference implementation to illustrate some high-level concepts such as linearizability [1], wait-freeness [14] or obstruction freeness [13]. Others demonstrate the use of memory management schemes such as hazard pointers [10] to establish the type safe memory these data structures require. However, little work is published on the building blocks leading to lock-free data structures (Valois’ work [11] forming an exception) and on non-pure algorithms that, like Krieger’s MCS lock, are lock-free in the important operations but use potentially unfair locks when this unfairness does no harm. The consequences are that it is difficult to find an implementation that perfectly suits a given problem, it is hard to identify the building blocks used in such an implementation and even harder to perform the necessary adjustments.

More raising the problem than providing a definite solution, we propose to collect searchable building blocks for lock-free data structures. By building blocks we mean essential ways to introduce a certain functionality and the prerequisites and properties they imply. For example, dequeuing from the middle of a double-linked list may be implemented using Valois’ helper nodes [11], but with the limitation of not being able to reuse these nodes immediately or, in a mostly lock-free fashion, with Krieger’s spin locks that are just used for the dequeue operation. RCU [9] is an excellent building block for read-most data structures and deferred object destruction, however send queues are write dominated. In our case, relaxing the validity of the head pointer allowed for a very simple implementation of the queue and the queue-state dependent implicit locking of the callee improves performance for the uncontented case.

6 Conclusions

This paper describes the modifications necessary to turn Krieger’s MCS-style queue lock into a send queue for call-reply style synchronous IPC paths. To our surprise, Krieger’s lock already provides all the essential functionality. Our send queue extends Kriegers lock in two invariant driven ways: by introducing a head pointer and by introducing an implicit locking scheme where threads enqueuing into an empty queue immediately get hold of the callee lock. This resulted in a fast send queue usable in call-reply style synchronous cross-processor IPC paths.

References

[12] K. Fraser “Practical lock-freedom” PhD THESIS, University of Cambridge, Feb 2004
DYNAMIC MEMORY ALLOCATION ON REAL-TIME LINUX

Jianping Shen
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
shen.jianping@foerstergroup.de

Michael Hamal
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
hamal.michael@foerstergroup.de

Sven Ganzenmüller
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
ganzenmueller.sven@foerstergroup.de

Abstract
Dynamic memory allocation is used in many real-time systems. In such systems there are a lot of objects, which are referenced by different threads. Their number and lifetime is unpredictable, therefore they should be allocated and deallocated dynamically. Heap operations are in conflict with the main demand of real-time systems, that all operations in high priority threads must be deterministic. In this paper we provide a generic solution, a combination of the memory pool pattern with a shared pointer, which meets both: high system reliability by automatic memory deallocation and deterministic execution time by avoiding heap operations.

1 Introduction

Dynamic memory management on real-time multi-threaded systems has two handicaps.

1. Execution time for memory allocation and deallocation should be fast and predictable in time. Heap allocations in general (with new() or malloc()) are not deterministic, because of memory fragmentation and non-deterministic behaviour of system calls (brk(), sbrk(), mmap()) [1].

2. For objects which are referenced by more than one thread, it is difficult or even impossible to predict which thread will be the last user of the object and has the duty to deallocate the object’s memory at the right time.

For such systems we need a deterministic, automatic and multi-threading capable dynamic memory management solution for C and C++ real-time development.

2 Detailed Problem Description

In this section we will explain the two mentioned handicaps in detail and try to figure out the solutions.
2.1 Execution Time of Memory Allocation

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for `malloc()`, while others supply functions to control certain regions of data. The same dynamic memory allocator is often used to implement both `malloc()` and `operator new()` in C++ [2].

Thus, we firstly limit our discussion to the following preconditions:

- Operating system: Linux
- C Runtime Library: glibc
- Architecture: X86-32

Linux uses virtual memory, each process runs in its own virtual address space. Linux dynamically maps the virtual memory to physical memory during runtime. The virtual memory layout for a running process is shown in figure 1.

![FIGURE 1: Process memory layout on Linux.](image)

The allocator implementation in glibc is `ptmalloc2()`. A memory block managed by `ptmalloc2()` is called chunk. The heap organises all available chunks in two containers called Bins and Fastbins. Fastbins contains small sized\(^1\) chunks for fast allocation, Bins contains normal sized\(^2\) chunks. Available chunks in Bins are organized in 128 size-ordered double linked lists (shown in figure 2).

<table>
<thead>
<tr>
<th>index</th>
<th>2</th>
<th>avactbins</th>
<th>4</th>
<th>...</th>
<th>64</th>
<th>65</th>
<th>sorted bins</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>10</td>
<td>24</td>
<td>32</td>
<td>...</td>
<td>512</td>
<td>576</td>
<td>640</td>
</tr>
</tbody>
</table>

![FIGURE 2: Available chunks in Bin.](image)

If the deallocator function `void free(void* ptr)` is called, a chunk will be released. Which means the chunk is marked as available and dependent on its size, goes to Bins or Fastbins for further allocations. Obviously the available chunks in Bins and Fastbins are dynamic and dependent on runtime conditions.

If the allocator function `void* malloc(size_t size)` is called, the following steps will be processed:

1. If memory size ≤ `max_fast`, `ptmalloc2()` tries to find a chunk in Fastbins.
2. If step 1 failed or size > `max_fast` and size ≤ `DEFAULT_MMAP_THRESHOLD`, `ptmalloc2()` tries to find a chunk in Bins.
3. If step 2 failed, `ptmalloc2()` tries to increase the heap size by calling `sbrk()`.
4. If step 3 failed or size > `DEFAULT_MMAP_THRESHOLD`, `ptmalloc2()` calls `mmap()` to map a physical memory in the process virtual address space.
5. Finally return the allocated chunk, or `NULL` if steps 1 – 4 all failed.

This is a very simplified description. What really happens is much more complicated, but we don’t want to inspect the details. It is now important to know, that execution time of memory allocation is not predictable.

For a memory allocation of small or normal size (steps 1 – 3), `ptmalloc2()` tries to find an appropriate chunk in Bins or Fastbins. The number of available chunks in both containers is dynamic and dependent on runtime conditions. The allocation time is therefore not predictable.

For large size memory allocations (step 4), `ptmalloc2()` uses the system call `mmap()` to map

\(^1\)size ≤ `max_fast` (default 72 Bytes).

\(^2\)size > `max_fast` (default 72 Bytes) and size ≤ `DEFAULT_MMAP_THRESHOLD` (default 128 kB).
physical memory in the process' virtual address space. With virtual memory management, the physical memory may be swapped on hard disk. It involves disk IO, thus, its execution time is also unpredictable.

On other operating systems and architectures, the most allocator implementations involve system calls. Hence the memory allocation on such systems may be expensive and slow. The allocation time may or may not be predictable, depending on the concrete implementation.

A common solution for this problem is the memory pool approach. A memory pool preallocates a number of memory blocks during startup. While the system is running, threads request objects from the pool and return it back to the pool after usage. In a single threaded environment, the memory pool pattern allows allocations with constant execution time [3]. In a multi-threading environment, the execution time is predictable, but not constant.

2.2 Memory Deallocation at Right Time

Consider memory allocations in a multi-threading environment. ThreadA allocates a memory block, and passes it as pointer to ThreadB and ThreadC. Now ThreadA, ThreadB, and ThreadC are all users of this memory block. The last user must deallocate the memory block to prevent a memory leak. Which of them becomes the last user is dynamic and depends on runtime conditions. In such a situation, it is impossible to safely deallocate the memory by just calling the deallocator in one of the three threads.

A solution to this problem is the shared pointer pattern. The idea behind shared pointers (and other resource management objects) is called Resource Acquisition Is Initialization (RAII). Once a memory block is allocated, it will be immediately turned over to a shared pointer. A reference counter is used by the shared pointer to keep track of all memory block users. The shared pointer automatically releases the memory block when nobody is using this block any longer [4].

With the help of shared pointers, the users (in our case the threads) don’t need to release the memory by themselves, this will be done by the shared pointer always at the right time.

3 The Approach

3.1 Combination of Memory Pool and Shared Pointer

The goal is to benefit from both approaches, a combination of the memory pool with shared pointers.

1. The memory pool preallocates a number of objects (see figure 3).
2. During runtime threads acquire objects from the pool as shared pointers.
3. If nobody is using the object any longer, the shared pointer will automatically return the object back to the pool (see figure 4).

3.2 Execution Time of Shared Pointers

The common shared pointer implementation raises an interesting question: where’s the reference counter located and who allocates it? Unfortunately, it will be allocated with new() on heap by the shared pointer itself. That means the constructor call of a shared pointer is unpredictable in time.
Hence we have to modify the shared pointer in a way that we also preallocate the reference counter as follows.

1. Extend the memory pool. The memory pool must preallocate memory not only for the object, but also for the object’s reference counter.
2. Modify the standard shared pointer to work with the memory pool’s preallocated reference counter.

### 3.3 The Final Approach

We put 3.1 and 3.2 together, and provide here the final solution.

1. The memory pool preallocates memory for user data and its reference counter.
2. At the runtime the process acquires memory from memory pool as shared pointer.
3. The Shared pointer uses the preallocated reference counter. Its execution time is therefore predictable. The Shared pointer keeps also a pointer to the memory pool. It will be used to return memory in step 4.
4. When nobody is using the memory block any longer, the shared pointer automatically returns the memory to memory pool.

This solution works without memory allocation at runtime. The process and the shared pointer both use preallocated memory from the memory pool. In our approach we call the shared pointer `RtSharedPtr`.

### 3.4 Execution Time

Our memory pool is designed to be used in multi-threading environments. The maximum execution time of a memory acquisition from a memory pool is dependent on the maximum number of threads. For multi-threading environments the access to the memory pool is protected by a mutex. Let’s assume:

- $t_p = \text{execution time for a memory acquisition at a memory pool}$
- $t_m = \text{execution time of mutex lock + unlock}$
- $\text{thread}_{\text{max}} = \text{maximum thread number}$
- $t_{\text{all}} = \text{complete latency for a memory acquisition}$

#### Single-threaded environment:

$$t_{\text{all}} = t_p + t_m$$

The complete latency is constant in a single-threaded environment.

#### Multi-threaded environment:

Best case: no concurrent access to the memory pool

$$t_{\text{all}, \text{min}} = t_p + t_m$$

Worst case: full concurrent access, all threads access the memory pool at the same time.

$$t_{\text{all}, \text{max}} = (t_p + t_m) \cdot \text{thread}_{\text{max}}$$

Thus, the complete latency in a multi-threaded environment can be calculated as:

$$t_p + t_m \leq t_{\text{all}} \leq (t_p + t_m) \cdot \text{thread}_{\text{max}}$$

It is not constant, but predictable in time.

### 3.5 Performance Improvement

The mutex protection enforces that all parallel memory pool accesses will be serialised. This could be a performance bottleneck. A workaround is to create more memory pools to improve parallel memory acquisition.

If we create a memory pool for each dynamic data type in each thread, there is no concurrent memory pool access, thus, we don’t need the mutex any more. We can disable the mutex protection by creating a memory pool as thread local. To create a thread local memory pool, we call the memory pool constructor as follows

```cpp
RtMemoryPool(int size, bool tLocal=true, const QString& name)
```

We will reach the maximal performance. The execution time is minimal and constant.

$$t_{\text{all}} = t_p$$

Obviously in a single-threaded environment we should create the memory pool as thread local to get the best performance.
4 The Implementation

4.1 Memory Pool

Our implementation\(^3\) is based on C++ templates\(^5\), therefore all preallocated memory chunks of a concrete memory pool are of the same size. That means, we need a memory pool for each dynamic allocated data type but the template approach guarantees type-safety.

```
template<typename F>
class RtSharedPtr;

template<typename T>
class RtMemoryPool
{
    RtMemoryPool(int size, bool tLocal, const QString& name);
    RtSharedPtr<T> sharedPtrAlloc();
    ...
};
```

The preallocated memory blocks are organized as a linked list. Each block contains user data, its reference counter and a pointer to next block. For a memory acquisition the pool returns always the head block. (see figure 5)

```
FIGURE 5: Memory Acquisition
```

If a memory block returns, it will be added as the new head block in front of the list. (see figure 6)

```
FIGURE 6: Return Memory
```

Instead of a raw pointer the memory pool returns a shared pointer to the allocated memory block. The shared pointer uses the reference count in the allocated memory block, and keeps a pointer to the memory pool.

```
// usage
RtMemoryPool<SomeType> memPool(100, "MyPool");
RtSharedPtr<SomeType> sptr = memPool.sharedPtrAlloc();
```

If the template type is a class, the memory pool will call the class constructor\(^4\) to initialize the memory block. Accordingly the class destructor will be called when the memory block returns back to the pool. If the object holds some resources\(^5\), they are released by the class destructor to prevent resource leaks.

4.2 Shared Pointer

We modified the boost shared pointer\(^6\) as follows:

1. Make it possible to use the preallocated reference counter.
2. Add a boolean to indicate whether the managed memory block is from a memory pool.
3. Add a memory pool pointer, which will be used to return the memory block back to the pool.
4. Extend the release behavior; make the shared pointer also be able to return memory back to the pool.

---

\(^3\)Our implementation is based on the Qt library, so we use QString, which can be easily replaced by std::string.

\(^4\)Constructors of pool objects should be kept simple, because their execution affects directly the memory acquisition time.

\(^5\)Objects with dynamically allocated resources violate the real-time conditions because of their destruction in the object’s destructor.

\(^6\)Objects with dynamically allocated resources violate the real-time conditions because of their destruction in the object’s destructor.
The modified shared pointer can work in two modes.

Normal mode: it works identically as a standard shared pointer and allocates the reference counter on heap. Its execution time is therefore unpredictable.

Real-time mode: in this mode the shared pointer must work together with a memory pool. Since the reference counter is preallocated in the pool, the execution time is predictable. There is no heap operation during runtime.

```cpp
template<typename T>
class RtSharedPtr
{
    template<typename F>
    explicit RtSharedPtr(F* p) :
        pn(p), px(p)
    {
    }

    // a new constructor
    template<typename F>
    explicit RtSharedPtr(F* p,
                          ReferenceCount* count,
                          RtMemoryPool<T>* pool) :
        pn(p, count, pool), px(p)
    {
    }
}
```

We provide a new constructor. Compared to the boost version it takes two extra parameters. A pointer to a preallocated reference counter and a pointer to the related memory pool. With the new constructort the shared pointer will be bound to the memory pool and to its preallocated reference counter.

A complete implementation can be found at http://www.foerstergroup.de/files/TS/RtSharedPtr.tgz

4.3 Test Results

We tested our solution on:

- **Hardware:** Intel Atom N270, 1 GB RAM
- **Operation System:** Linux 3.0.0-rt3
- **Compiler:** gcc/g++ 4.4.5
- **C-Library:** glibc 2.11.2

As test case we started two threads, one high-priority thread which allocates memory – in test case A from the memory pool (figure 7) and in test case B from heap (figure 8). The second thread acts as a noise generator, it allocates and frees memory chunks of different size on heap to bother the glibc’s allocator.

The execution time for an allocation in the high-priority thread is measured in µs and illustrated in figure 7 and figure 8.

**FIGURE 7:** Test case A: pool allocation.

**FIGURE 8:** Test case B: heap allocation.

Both test cases run for approximately 10 hours and produce about $10^9$ records.

Test case A shows, that 90% pool allocation is accomplished up to 5 µs, and 99.999% pool allocation up to 10 µs. The curve is limited to 60 µs on the time axis, which is the worst execution time. Our approach satisfies therefore hard real time systems timing requirements.

Compared to pool allocation, the execution time of a heap allocation is evenly distributed between 80 µs and 900 µs. Its worst excution time is not limited.

5 Conclusions

The described approach provides a generic solution, which meets both: high system reliability by automatic memory deallocation and deterministic execution time by avoiding heap operations.

We have tested our approach on Linux and Windows. As a generic solution it can be easily migrated to other operation systems, which do not provide operation system level memory allocation with deterministic execution time.
References


pW/CS - Probabilistic Write / Copy-Select (Locks)

Nicholas Mc Guire
Disttributed & Embedded Systems Lab, Lanzhou University
Tianshui South Road 222,Lanzhou,P.R.China
mcguire@lzu.edu.cn,safety@osadl.org

Abstract

The initial problem of protecting data for concurrent access was relatively simple, the goal was exclusive access to a shared resource. Elaborate semantical variations of the atomicity theme have been developed over the past decades, followed by a ever increasing focus on scalability. At the same time a continuously increasing complexity of operating systems and applications has resulted in a steady growth of the complexity of the locking subsystem - one could speculate that the locking complexity has been growing faster than the overall complexity of operating systems, but it would be hard to put numeric evidence on this claim - suffice it to state that the development of Linux in the transition from 2.2 to 2.4 and to the now current 3.X series of kernels has been very much dominated by locking issues related to scalability [8], [7].

At the same time we have seen that locking semantics has become more complex, priority inheritance/priority ceiling, fine grain locking, lock types dependent on global state [5] and large lock dependencies (or lock chains [13]) becoming common. This growth in complexity has dramatically impacted the development of real-time OS like the Preempt-RT real-time extension to the Linux kernel - not too surprising considering considerable efforts related to real-time are lock related [4],[6]. The paradigm has roughly remained the same - explicit mutual exclusion to critical regions and atomicity of access in a functionally deterministic manner along with hardware support for more elaborate atomic instructions (i.e. cmov, cmpxch).

This approach has a serious draw back:
- it is hard to make locking scalable
- detecting and fixing locking problems is becoming more difficult
- the performance impact of locking - notably on real-time - is problematic
- The worst case behavior is only a miniscule sub-state-space hard to actually reach during testing
- Timing wise the worst case is always the loaded system, thus reliable prediction of load impact is limited.

The question is - is there an alternative ? Notably one that scales with growing complexity ? Its not yet time to give a simple yes or no answer, but we believe that we can state that for some locking problems there are solution that can actually inherently scale with growing complexity. The problem simply has to be approached from a different perspective.

Operating systems have been traditionally modeled as deterministic constructs - code is deterministic - but in system scope this simply does not hold. Non-determinism at the temporal level paired with preemptible operating systems inherently leads to the inability to predict the global state of an operating system even in the near future (lets say a few seconds into the future). Thus modeling a task as running on a "random" global state - the operating system - allows a new perspective for access to shared resources.

Taking one step back, locking was not introduce to provide exclusive access, locking was introduced to ensure consistency of access to a shared resources - locking being one way this can be done in a straightforward manner. At times where memory was a scarce resource this approach made a lot of sense - with RAM readily available, though with significant access performance differences depending on physical location, alternative solutions for consistent access to shared resources may make more sense - one of these methods is probabilistic locking.

In this paper we present the motivation for a simply probabilistic lock - arguably the term lock is inappropriate - but we retain it as it serves the same purpose as the traditional locks - guarantee
consistency of shared data. This lock is not a one-fits-all solution to the problem of shared data in concurrent systems - but rather it should be seen as an attempt to change the perspective and view contemporary systems as what they are - inherently random systems - and capitalize on this notion to resolve the scalability problem.

1 Introduction

Computer science has been much focused on deterministic methods - notably when it comes to synchronization methods we rely on correctness proofs to assure that the methods are sound. While this does guarantee that these methods will not fail as long as we actually are able to model them (that is we know all involved locks) allowing us to exclude inconsistency of data - they are causing serious problems in the transition to multi-core systems - they don’t scale well. A further issue with the deterministic approach is that the worst case is expected under heavy load and thus testing only has a limited significance in certifying correctness as the state space that would need to be covered by testing is simply too large.

Real-time systems have constraints on access patterns, i.e. SCHED_FIFO makes it impossible for a tight-loop sequence of preemption to happen on a single CPU (unless intentionally coded) - thus the measures to ensure consistency could be relaxed. Unfortunately historically it seems that locking was developed for the general case of arbitrary preemption patterns and then these general results were specialized for the real-time case. The effect being that though real-time locking should be in principle simpler than the general non-real-time case, it is in fact more complex.

The question we ask is simple if the methods in use are actually solving problems that exist or if they are not heavily involved in solving non-existing problems with considerable overhead to do so. Even worse - on very large systems forcing unnecessary determinism might well be one of the main problems. Is it reasonable to assume arbitrary defined task sequences or arbitrary preemption sequences at the temporal level? For any real life system this makes little sense - for full fledged general purpose OS it makes absolutely no sense. Even more, the inherent randomness of modern CPUs [9] makes it close to impossible to actually achieve synchronous sequences of concurrent access to unprotected global objects, even if one were to maliciously attempt to do so. Methods like WCET estimation are becoming (actually are at this point) prohibitively pessimistic and thus practically not usable for multicore systems.

The algorithm described in this paper contains a race - and it can be quite trivially shown under what conditions the race exists (i.e. a SPIN model would reveal this). We will introduce a lock-less/0-wait algorithm to read a register (of in principle arbitrary size) while concurrently writing it. We will show that this algorithm is reliable with the reader and the writer being non-atomic and then argue that while theoretically unsafe with a non-atomic reader and non-atomic writer it is practically - that is statistically - safe with an arbitrary reliability target. Thus the trade-off is spatial replication vs interprocess synchronization time.

The main contribution of this article is to demonstrate that the growing complexity of modern systems (complex hardware and software) needs answers to core questions, that rather than fighting complexity, capitalize on it and result in robust systems under real-world conditions.

1.1 Race Condition

A race is a access pattern on a shared object that can’t be judged from the context of the involved tasks only. It is important to note that unprotected shared access in itself does not suffice to create inconsistency - essentially occurrence of inconsistency is bound to access patterns. thus there are two options:

- unify the context - i.e. add a shared lock to join the context
- de-couple context - i.e. randomize access to minimize joined context

The first is the "traditional" deterministic locking scheme, the second is not actually that new, but maybe just not yet presented in the context of locking. The goal is to design synchronization that scales with complexity rather than trying to enforce simplicity at the local level by increasing the global complexity.

The properties of masking locks build on the notion of inherent non-determinism of concurrency in modern CPUs. Basically this non-determinism at
task level is precisely the cause for race conditions in the first place, if modern systems were strictly synchronous at the global level then we could pre-determine any access patterns and consequently protect. Sources of non-determinism are plentiful in modern systems, not only asynchronous interrupts, but also non-deterministic cache replacement strategies, ECC RAM and flash (the later with correction rates in the order of 1 out of 100 accesses projected [11]), complex dependencies of instruction execution times, etc. All of this leads to a non-deterministic timing - that is execution time jitter - and paired with pre-emptibility - to a non-deterministic global state from the perspective of the individual thread of execution.

In safety related or HA systems traditionally random faults have been mask by replication and redundancy - we take a similar approach here but at a much smaller scale - the critical object is a single data object and the "fault" is the writing process. We start with a well studied and simple class - a single writer multiple reader construct - similar to the one introduced by Peterson in his influential paper "Concurrent reading while writing" [15], whereby the assumptions about the read and write operations are very much relaxed to reflect the nature of modern super-scalar multicores, that is no memory barriers or volatile data types are assumed. The design goals for race masking are:

- lock-less / 0-wait
- hand-shake-free
- non-atomic reader/writer
- constant number of steps for read and write (\(O(1)\))
- concurrent multiple reader, single writer
- never later than a locking version
- reader and writer crash safe
  - none can be blocked
  - no reader will receive an inconsistent or old value if the writer crashes
  - No bounds on the number of crashing processes
- an arbitrary probability of success can be provided (level of replication)
- failure probability decreases with increasing system complexity.
- failure probability decreases with increasing system load.

Scalability is not mentioned here simply because we don’t yet have a good model to actually describe and analyze scalability but clearly scalability is a prime target. Wile we list non-atomic read-write it should be noted that we are assuming that single 32 bit entities are written atomically - that is a write of a word to a memory/register location will never permit an inconsistent concurrent read - either the old value is read in its entirety or the new value is read in its entirety but no "mix" of the two - any sane architecture will guarantee that (at least at present).

1.2 Concept

The concept is embarrassingly trivial, the writer simply writes to the shared object indiscriminent of the state of any reader. Obviously this would not be safe for a single object - as with safety related systems where random faults must be covered - we simply view the concurrent threads as "randomly" accessing the data object and the writer is viewed as the "fault-injector".

![Fault-injection model](image)

Thus the therapy is simply replication. Rather than writing one protected region, we simply write N unprotected regions and with the inherent randomness of complex hardware software systems we can provide an arbitrary high probability that at least one of the regions is consistent at any time and thus can be retrieve by the readers. Summarized this simply means:

- probabilistic guarantee of success that can be set to arbitrary value
- write operation: write replicated registers unprotected
- read operations: copies replicated registers and selects
hence the name probabilistic Write/Copy-Select pW/CS lock.

The principles are roughly modeled along the lines of loosely coupled replicated systems to mitigate random faults in safety related systems:

- temporal serialization is replaced by spatial "concurrency"
- atomicity is replaced by a probability of success.
- atomicity of single object updates must be guaranteed (that is the write of a single 32bit word must be guaranteed to be consistent (that is the single load or store is consistent - which should hold on any CPU I hope).

replication is done to guarantee that at least one register is consistent and complete at any time (with an arbitrary selectable probability) for a given assumed maximum synchronous preemptions of reader and writer thread. The value available is always the last complete value written (though an in-progress write may be incomplete) - in any case a reader always gets access to the last consistent register copy, thus never later than a locking solution would provide.

So pW/CS addresses consistency of concurrently accessed data - it does not address completeness nor ordering issues - it is the lowest level primitive for sharing non-atomic resources without introducing a joint context constraint.

1.3 Categorization

In Lamport’s taxonomy this is a regular 1-writer algorithm

A regular variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes.

though Lamport’s taxonomy [3] might not be applicable to a probabilistic locking scheme but it fulfills the criteria quite nicely. As this is a low-level primitive only, the motivation to build on such definitions is to allow deducing high-level constructs (i.e. monitors seem a quite natural option) to build on this primitive.

2 Register layout and protocol

In this section we describe the reader and writer protocols as well as the replicated register layout.

Replicated register set layout

The principle layout is simply a set of N registers with 2N markers guarding it, so a N-register for pW/CS protection would be:

\([\text{marker}, \text{reg}_1, \text{marker}] \ldots [\text{marker}, \text{reg}_N, \text{marker}]\)

Note that the markers are to be unique if unbounded reader delays are permitted, if reader delays are bounded then the markers type space must be sufficient to cover uniqueness within the reader delay for un-delayed writers (or register aliasing could occur - i.e. a role-over of a marker if the marker were only a char). In the proof of concept implementation the largest inherently atomically writable object, a 32bit value, is used as marker.

Read and write protocol

The readers and writers have a simple protocol to follow, basically the direction of access is inverted. The selection of direction is of course arbitrary - the essence only that readers and writers access in opposite direction. Now on weakly ordered architectures this might not hold (or require larger number of replicas) - but as the approach in itself is non-deterministic this does not matter as no consistency assumptions are actually made.

![access model](image-url)

FIGURE 2: access model

- writer protocol:
  - update replicas (left to right)
  - update protocol:
update leading (left) marker
update register
update the trailing (right) marker

• reader protocol:
  – copy register set (right to left)
  – select consistent register
  – selection protocol:

for(reg=right, reg<left, reg--){
  if(markers identical ){
    select register
  }
}
return register

protection by write and read being in opposite directions. It is not possible to get inconsistent data even with a single pass - it is though possible to get no data (all data is found to be inconsistent). The probability of the occurrence of all data is inconsistent can though be brought down to an arbitrary low value with sufficient replication.

Note that there are possibilities for "smarter" protocols than the above brute-force one. For the proof-of-concept implementation this simple minded approach was shown to work just fine - and as it allows simple modeling it is what we are currently using.

3 race-masking with implicit reader locking

The initial motivation for looking into race masking was to allow lock-less coordination of real-time and non-real-time tasks on a real-time enhanced GNU/Linux system. Essentially this section is to show that the introduction of real-time priorities will also only improve the situation but never aggravate it in the sense that the probability of success is never reduced.

A non-probabilistic variant is by implicit priority locking of the reader - if the reader has higher priority than the writer then it is not possible for the writer to preempt the reader and thus it is guaranteed that the reader will be able to copy the entire buffer uninterrupted - in this case it can also be guaranteed that the reader gets at least one consistent copy if $N \geq 3$ replicas of the register are used.

This is nothing really new, Lamport suggested this in 1985 [3] suggesting the idea actually stems from 1977 but uses it in a deterministic algorithm to implement a multivalent regular register. Dijkstra proposes a non-deterministic selection in his paper titled "Guarded Commands, Non-determinacy and Formal Derivation of Programs" 1975 [1] from which we use the idea of guards to protect a set of in principle non-deterministically selected actions (copying of the register). Interestingly enough Hoare in "Communicating Sequential Processes" [2] describes a number of situations based on Dijkstras guarded commands that resemble the pW/CS locks proposed here, though the context is quite different. Ultimately non-determinism has been proposed in many publications though we are not aware of examples where this non-determinism is actually utilized - this alone is the novelty of the proposed design here and we believe it is potentially useful in resolving scalability problems in at least some situations.

4 General race-masking with probabilistic locking

If the requirement of well set priorities and thus one-sided non-preemptibility is dropped then there is a possibility that the read will return with none of the registers in a detectable consistent state. That is actually we don't know the state of the register - we infer positively that the register is consistent if the markers are identical. At the same time we can not positively infer an inconsistency of the register in case of markers being unequal though. But taking the inconsistency of the markers as indication of inconsistency of the registers is a pessimistic assumption in all cases and thus safe.

To fail the access to the registers must be strictly in lock-step order for readers and writers - for $N$ replicas $N*2+1$ lock-step access would be needed to result in all registers being inconsistent. Note that the probability of such a lock-step behavior does increase with the size of the critical region (actually the uninterrupted time spent in the critical region).

A collision (all registers in an intermediate state) would require $2N+1$ synchronous preemption - so 13 synchronous preemption for a system using 6 replicas. With synchronous preemption we mean that the preemption of the reader must occur after a complete register with markers was read every time and the preemption of the writer must happen in the middle of the register region every time plus that last read must also be preempted to ensure that no register is read in a consistent state.

Such an aliasing for $N = 3$ requires synchronous
preemption of reader and writer in at least 6 consecutive cases - this means a sixfold synchronous race conditions is needed to result in inconsistent data - what is the probability of such a scenario if single race conditions are hard to reproduce?

In fact the race condition could be extended to arbitrary number of race hits to be needed to result in inconsistent data and thus one can provide arbitrary probability of success (at the expense of larger number of replications).

This solution could be described as, a somewhat paradox, "safe race" - safe to an arbitrary probability of successful reading of at least one consistent register.

The current proof-of-concept is for a register consisting of 3 integer values, but is extensible to any data structure - we note though that the prime interest is in resolving synchronization of small data objects, where race occurrence is very unlikely and thus traditional locking excessively wasteful.

The writer is simply an unconditional write to the register set.

```c
{  ui[i].w_enter++;  ui[i].period = period;  ui[i].duty = duty;  ui[i].bit = 1;  ui[i].w_exit++;  i++;}
```

The reader copies the register set in reverse order and then runs a selection loop on it:

```c
while(!exit_cond){  i = NUM_REPLICA-1;  do{    pwm[i].w_enter = ui[i].w_enter;    pwm[i].period = ui[i].period;    pwm[i].duty = ui[i].duty;    pwm[i].bit = ui[i].bit;    pwm[i].w_exit = ui[i].w_exit;  }while( i >= 0);  for(i=0;i<NUM_REPLICA;i++) {    if(pwm[i].w_enter - pwm[i].w_exit == 0) {      /* consistent register set found */    }  }

The selection can then simply set the pointer to the first valid entry found, note that the first found is the last written thus the most current of the N replicated registers so the selection can stop once a consistent register set was found. To ensure this the individual repicas though must be on cache line boundaries - if they were fit in a single cache line then the ordering implemented in the software would not necessarily be honored by the hardware.

5 Properties

5.1 Assessment of the randomness hypothesis

The most interesting issue in the experiments was to determine if the data can bolster the claim of a random fault scenario. If this assumption is false then obviously the underlying model would not be valid and consequently the conclusions also not - at least not at this point in time.

The random fault model is basically claiming that the writer actually has the same properties as a random fault injection - even though it is obviously systematic in nature, its timing is suspected to be truly random. If this holds then the mitigation of the fault also holds - with some constraints of course that will be developed a bit later.

To assess that the failures are actually random we take two main data samples into account.

- timing distribution
- distribution of single buffer inconsistencies

From this data, presented below, we can conclude that the writer process actually exhibits properties of a random fault (SEU).

FIGURE 3: timing distribution of one reader
5.2 probability of failure

In the above example a 6-tuple replica was in use, for N registers \(2^N + 1\) synchronous preemption are required - what is the probability of this happening?

To see this we look at the distribution of the race occurrence on a single unprotected global integer over the loop length. 10000 runs are done and then the occurrence of races is plotted, showing the approximation of the race probability.

The actual occurrence of a race is almost perfectly normal distributed, if one creates 1000 instances of two racing threads for a given fixed loop length and records the number of races that occurred the distribution is close to a perfect gauss curve.

5.3 Failure rates of pW/CS

To estimate the failure rate we implemented a pW/CS protected data object and ran tests where we record the distribution of inconsistent registers (actually inconsistent markers which is a conservative indication of inconsistent registers) and plot the distribution for different scenarios. From this data we derive a model of the distribution and estimate the probabilities involved.
pW/CS - Probabilistic Write / Copy-Select (Locks)

FIGURE 8: buffer inconsistency distribution load 16 (24 core AMD)

FIGURE 9: buffer inconsistency distribution load sweep from 0 to 16, 2 threads vs 256 threads (24 core AMD)

Note that the idle is the worst case (as expected). Further this code has a close to minimal loop body thus the probability of a preemption occurring in the critical section is very high and can be expected to be smaller in almost ever other case. Again this is quite the opposite of what you have in traditional locking where "keep it simple" is considered best-practice - with probabilistic locks increased complexity of access to the shared data is actually an advantage. The more erratic the system is the lower the probability of N lock-step preemptions leading to all buffers being inconsistent.

Notably current tests have shown that larger and thus more complex systems perform better than simpler systems - though we must note that we only had very limited access to large systems so the tests were generally only short runs and sometimes under not well specified load conditions.

5.4 Performance of pW/CS

The performance evaluation is quite preliminary, partially because we don’t have a sufficiently complete model yet, partially because the test-case is quite artificial and it needs to be demonstrated that this actually holds for a real problem. The performance assessment is done by looking at the time it takes to access the shared data object and plotting this time as a histogram - it shows that the time distribution is very favorable for the probabilistic lock even on a loaded system (note that readers and writer are SCHED_OTHER not RR or FIFO).

The comparison is done between code using pW/CS and code using a normal pthread_mutex to protect the shared object.

FIGURE 10: timing idle system (4 core Intel)

FIGURE 11: timing loaded system (4 core Intel)

Notably running the same test on a larger system - a 24 core AMD (2 CPUs) one can clearly see that the difference between the probabilistic approach and the deterministic approach widens.
5.5 Fairness

Fairness of any synchronization object is a critical issue. While locks don’t exhibit much unfairness on small systems (single to 4 core systems) and mild load scenarios, the lock-fairness can become a major issue on 16 or 32++ way systems. Any new locking proposal thus must exhibit fairness and scalability (note that fairness can also be achieved “brute-force” at the price of performance and/or scalability - i.e. server type approaches).

pW/CS lock can’t lead to reader nor writer starvation - if it does then the CPU was overloaded to begin with - but the locking regime it self does not contribute to un-fairness or starvation. with respect to scalability we are not yet sure if we can give it a thumbs-up, tests have been limited to a few systems up to now and only one was a 16 core (nehalem) and one 24 core AMD system, thus it is to early to say - from the conceptual side we believe that this is a scalable solution though.

5.6 Failure behavior

The maybe most interesting behavior of pW/CS is that a failure of one of the participating threads is simply irrelevant the writer can at best leave one data replica inconsistent, a reader would go entirely unnoticed as it does not alter the state of the shared object at any time.

pW/CS obviously does not have any double locking issues as there is no actual lock involved.

The prototype implementation used counters to check consistency, this is simple to implement but strictly not sufficient to guarantee correctness of the shared object, a better solution, though somewhat more involved computationally is to use a simple CRC to ensure consistency - first tests are running but were not ready on time for this paper (we need reasons to publish more papers on this any way...)

5.7 Testability

Traditional locking needs to be tested under load conditions, which are not only hard to generalize but never can cover all possible situations. Probabilistic approaches on the other hand can be designed to have their worst case probability of collision in the idle system - that is a high load improves the probability of a un-synchronized access and thus also decreases the probability of a collision which requires a complex synchronized access pattern. Thus in principle probabilistic locks are fully testable by testing on the idle system. We would like to emphasis that we don’t yet see this proof-of-concept as verified to carry this property though we do think that it is possible to build synchronization that exhibits the property of ”idle is the worst case”.

6 Possible advantages

While ”deterministic” code can’t be exhaustively tested, and it is common that while testing a number of profiles, maybe for extensive periods of time, that the problem surfaces in either an untested profile or simply as a matter of time.

The root problem is that we can’t reliably produce the ”worst-case” situation on a system, not even on a fairly simple hardware/software system, thus leaving the occurrence of the worst-case to the field.
With other words the problem becomes more likely with high-load situations and we can’t test all possible combinations of high-load situations.

What is now behind the problem is that the race condition becomes a rare but possible problem - a specific global state of the system - if it occurs we fail. In this sense the failure is deterministic (functional view) but its dependency is relatively complex so it is hard to test. On the other hand the state of the good case (looking at the successful execution of the synchronization object) is well defined "deterministic" in most states - but the state space is very large so it is hard to achieve coverage. All we need to do is turn it around - make the race depend on a complex deterministic global state and make the good case independent of a particular state - that is - the good case should have a very large state space coverage, and the bad case a small and testable state-space.

The mitigation proposed here is to use synchronization that exhibits its worst-case behavior on the idle system. A probabilistic lock will have it highest probability of aliasing, and thus failing, in an idle system, and the higher or more erratic the load situation is the better it gets - because the probability of synchronous preemption that cause the possible collision decreases. Thus we regain the ability to test synchronization potentially as we can now reliably provide the worst-case with one single profile - the idle system. At the same time the good case does not have a single deterministic global state constellation but rather happens in a large number of independent states (that is only one register must be consistent from N).

A second aspect of raciness is the temporal dimension - in traditional systems one could observe that something that worked well for a long time suddenly fails because of optimization or faster hardware - we had the "implicit ordering" simply by the execution flow that protected the unprotected critical region. Now the probabilistic lock has exactly the opposite qualities, the fast the system gets the lower the probability of the reader not achieving a consistent read before being preempted, and this also holds for optimization of compilers - so again we can test the worst case - slow system, unoptimized code - it can only get better for the probability of the race condition not occurring in all N replicas of the register set.

Finally the issues of Amdahl’s law, the longest serialized portion of code can quickly dominate the overall performance. As pW/CS has no serialization of readers and the writer there is no impact on concurrent threads by individual threads being preempted/blocked - the reader will always have access to the latest consistent buffer the writer was able to provide.

6.1 Next steps

The current proof-of-concept implementation is sub-optimal in that it creates a N-replica copy for each reader. This is an unnecessary overhead in that it would be at most suitable to create such a "scratchpad" per NUMA-node on a NUMA system, for non-NUMA a direct selection from the writers replicas is also an option. Further on the implementation side, the currently used counters should be replaced by a stronger consistency check - with the increasing overhead of accessing remote CPUs the overall local computation that can be expended if communication can be reduced to hand-shake-free (write-and-forget) semantics are considerable, equally the expendable spatial overhead is considerable before approaching a lock based implementation (on a 4x4 system that we had access to temporarily a implementation using 100 replicas was still faster than a locking version for a shared object of 16 bytes !). More work needs to be done to understand where the break-even point would lie and consequently where this approach would be suitable.

The second large area of future work is in the modeling of this concept. The current approach of a quite brute-force implementation to get a better understanding of the approach and its potential is hardly suitable for actual deployment in a real system if no formal model is available for assessment. Unfortunately the available models don’t fit the approach well. Maybe with the exception of Dijkstra’s guarded commands and non-deterministic if construct. The only real difference being that while Dijkstra’s guarded commands evaluate the guards to determin if execution should take place, pW/CS unconditionally copies the replicated register instance and then uses the "guards" to determin if the selection should take place or if the replica is abandoned - currently we intend to utilizing Dijkstra constructs to model pW/CS.

7 Conclusion

With ever growing complexity, designing deterministic while optimal systems is becoming increasingly hard (or actually impossible). In this paper we propose to look at potentially capitalizing on the growing complexity rather than fighting it - by utilizing
the inherent randomness of complex systems in combination with probabilistic locking methods.

We demonstrate the feasibility of this approach with an admittedly naive implementation of a critical section shared between a concurrently executing readers and writer of arbitrary priority. The results indicate that with growing complexity of the system, with higher system load, and with increased number of readers the probability of failing is reduced. Further faster systems have a higher probability of success than slower systems, and equally optimization of compiler plays to our advantage.

We are aware that this is too early to call this a sound and reliable result but the preliminary investigation does indicate that the proposed path - stop fighting complexity, use it! - is worth investigation in more detail.

The most notable obstacle to utilize such approaches in our opinion is the lack of appropriate models for probabilistic approaches. This clearly will be our next steps in this effort to capitalize on the inadvertable trend of growing hardware and software complexity. Further a systematic tradeoff study, comparing traditional locking options in relation to system complexity will be on our TODO list.

The main conclusion from this work though is simply that locking may not be the best solution for concurrent access to shared objects - rethinking the problem in the context of modern super-scalar multicore systems might well be worth the effort.

Acknowledgment

We would like to thank Silicon Graphics GmbH, Germany, specifically Mr. Heinz Moser for supporting this research effort by providing us with suitable multicore system for development and testing. This support allowed us to perform an initial evaluation of the scalability properties of pW/CS.

Sources used for this project are available on request under the GPL V2 from DSI Lab [14] and will be released as soon as they are cleaned up.

References


[15] Concurrent reading while writing, Gary L. Peterson, James E. Burns, 1983. ACM Transactions on Programming Languages and Systems
On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems

Paulo Baltarejo Sousa  
CISTER-ISEP Research Center, Polytechnic Institute of Porto  
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal  
pbs@isep.ipp.pt

Konstantinos Bletsas  
CISTER-ISEP Research Center, Polytechnic Institute of Porto  
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal  
ksbs@isep.ipp.pt

Eduardo Tovar  
CISTER-ISEP Research Center, Polytechnic Institute of Porto  
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal  
emt@isep.ipp.pt

Björn Andersson  
Software Engineering Institute, Carnegie Mellon University  
Pittsburgh, USA  
baandersson@sei.cmu.edu

Abstract

In this paper we discuss challenges and design principles of an implementation of slot-based task-splitting algorithms into the Linux 2.6.34 version. We show that this kernel version is provided with the required features for implementing such scheduling algorithms. We show that the real behavior of the scheduling algorithm is very close to the theoretical. We run and discuss experiments on 4-core and 24-core machines.

1 Introduction

Nowadays, multiprocessors implemented on a single chip (called multicores) are mainstream computing technology and it is expected that the number of cores per chip continue increasing. They may provide great computing capacity if appropriate scheduling algorithms are devised. Real-time scheduling algorithms for multiprocessors are categorized as: global, partitioned and semi-partitioned.

Global scheduling algorithms store tasks in one global queue, shared by all processors. Tasks can migrate from one processor to another; that is, a task can be preempted during its execution and resume its execution on another processor. At any moment the m (assuming m processors) highest-priority tasks are selected for execution on the m processors. Some algorithms of this category can achieve a utilization
bound of 100%, but generate too many preemptions.

Partitioned scheduling algorithms partition the task set and assign all tasks in a partition to the same processor. Hence, tasks cannot migrate between processors. Such algorithms involve few preemptions but their utilization bound is at most 50%.

Semi-partitioning (also known as task-splitting) scheduling algorithms assign most tasks (called non-split tasks) to just one processor but some tasks (called split tasks) are assigned to two or more processors. Uniprocessor dispatchers are then used on each processor but they are modified to ensure that a split task never executes on two or more processors simultaneously.

Several multiprocessor scheduling algorithms have been implemented and tested using vanilla Linux kernel. Litmus$^{\text{RT}}$ [1] provides a modular framework for different scheduling algorithms (global-EDF, pfair algorithms). Kato et al. [2] created a modular framework, called RESCH, for using other algorithms than Litmus$^{\text{RT}}$ (partitioned, semi-partitioned scheduling). Faggioli et al. [3] implemented global-EDF in the Linux kernel and made it compliant with POSIX interfaces.

In this paper we address the Real-time TAsk-Splitting scheduling algorithms (ReTAS) framework [11] that implements a specific type of semi-partitioned scheduling: slot-based task-splitting multiprocessor scheduling algorithms [4, 5, 6]. Slot-based task-splitting scheduling algorithms assign most tasks to just one processor and a few to only two processors. They subdivide the time into equal-duration timeslots and each timeslot processor is composed by one or more time reserves. These reserves are used to execute tasks. Reserves used for split tasks, which execute on two processors, must be carefully positioned to avoid overlapping in time.

The remainder of this paper is structured as follows. Section 2 provides a description of the main features of the slot-based task-splitting scheduling algorithms. Section 3 discusses some challenges and design principles to implement this kind of algorithms. A detailed description of our implementation is presented in Section 4 while in Section 5 we discuss the discrepancy between theory and practice. Finally, in Section 6 conclusions are drawn.

2 Slot-based task-splitting

Slot-based task-splitting algorithms have two important components: (i) the task assigning; and (ii) the dispatching algorithm. The task assigning algorithm executes prior runtime and besides assigning tasks to processors is also responsible for computing all parameters required by the dispatching algorithm. The dispatching algorithm works over the timeslot and selects tasks to be executed by processors.

The Sporadic-EKG (S-EKG) [4] extends the periodic task set model of EKG [7] to sporadic task set models. This approach assures that the number of split tasks is bounded (there are at most $m - 1$ split tasks), each split task executes on only two processors and the non-split tasks execute on only one processor. The beginning and end of each timeslot are synchronized across all processors. The end of a timeslot of processor $p$ contains a reserve and the beginning of a timeslot of processor $p + 1$ contains another reserve, and these two reserves supply processing capacity for a split task. As non-split tasks execute only on one processor they are scheduled according to the uniprocessor EDF scheduling algorithm. A detailed description of that algorithm with an example can be found at [8].

While EKG versions are based on the task, the NPS-F [5, 6] uses an approach based on bins. Each bin is assigned one or more tasks and there is a one to one relation between each bin and each notional processor. Then, the notional processor schedules tasks of each bin under the EDF scheduling policy. The time is split into equal-duration timeslots and each timeslot is composed by one or more time reserves. Each notional processor is assigned one reserve in one physical processor. However, up to $m - 1$ notional processors could be assigned to two reserves, which means that these notional processors are implemented upon two physical processor reserves, while the remaining notional processors are implemented upon one physical processor reserve.

There is one fundamental difference between S-EKG and NPS-F algorithms. NPS-F can potentially generate a higher number of split tasks than S-EKG. Another difference is related to the dispatching algorithm. The S-EKG allows non-split tasks to be executed on the split task reserves (in the case when these tasks are not ready to be executed) while NPS-F does not; that is, each notional processor executes only on its reserve(s).

Fig. 1 shows a generic execution timeline produced by these scheduling algorithms. The time is divided into equal-duration timeslots of length $S$. Each timeslot is divided up to 3 reserves: $x[p]$, $y[p]$ and $N[p]$. Reserve $x[p]$ is located in the beginning of the timeslot and is reserved to execute the task or notional processor split between processors $p$ and $p - 1$. Reserve $y[p]$ is located in the end of the timeslot and
is reserved to execute the task or notional processor split between processors $p$ and $p+1$. The remaining part ($N[p]$) of the timeslot is used to execute non-split tasks or notional processors that execute on only one processor.

FIGURE 1: Execution timeline example.

In the remainder of this paper, we will discuss the implementation of S-EKG an NPS-F algorithms in 4-core and 24-core machines supported by the Linux 2.6.34 kernel version.

3 Challenges and design principles for implementing slot-based task-splitting

In [9] a set of challenges and a set of design principles for the S-EKG implementation were discussed. However, in this paper we will implement NPS-F as well and for this reason we will need to adapt the design principles. In this section, we will do so as follows: each processor should have its own runqueue (the queue that stores ready jobs). The runqueue of each processor should map the ready tasks with its reserves; that is, which ready tasks are allowed to execute on each reserve. Since some tasks may execute on two processors, what is the best approach for that? If tasks are stored locally on each processor, whenever a task migrates from one processor to another processor, it requires locking both processor runqueues for moving that task from one runqueue to the other runqueue. However, in the case of the NPS-F this could imply moving more than one task. Since the frequency of migration may be high, it turns out that this is not the best approach; so we adopted a different approach. We defined a runqueue per notional processor so each notional processor stores all ready tasks assigned to it. Then, we map each notional processor to processor reserves.

As it is intuitive from observing two consecutive timeslots in Fig. 1, whenever a split task consumes its reserve on processor $p$, it has to immediately resume execution on its reserve on processor $p+1$. Due to many sources of unpredictability (e.g. interrupts) in a real operating system, this precision is not possible. Consequently, this can prevent the dispatcher of processor $p+1$ to select the split task because processor $p$ has not yet relinquished that task. In order to handle this issue, one option could be that processor $p+1$ sends an inter-processor interrupt (IPI) to processor $p$ to relinquish the split task, and another could be that processor $p+1$ sets up timer $x$ time units in future to force the invocation of its dispatcher. Two reasons have forced us to choose the latter. First, we know that if a dispatcher has not yet relinquished the split task it was because something is preventing it from doing so, such as, the execution of an interrupt service routine (ISR). Second, the use of IPIs will create some dependency between processors that could embarrass the scalability of the dispatcher.

4 Implementation of slot-based task-splitting

4.1 Assumptions about the architecture

We assume identical processors, which means that (i) all processors have the same instruction set and data layout (e.g. big-endian/little-endian) and (ii) all processors execute at the same speed.

We also assume that the execution speed of a processor does not depend on activities on another processor (for example whether the other processor is busy or idle or which task it is busy executing) and also does not change at runtime. In practice, this implies that (i) if the system supports simultaneous multithreading (Intel calls it hyperthreading) then this feature must be disabled and (ii) features that allow processors to change their speed (for example power and thermal management) must be disabled.

We assume that each processor has a local timer providing two functions. One function allows reading the current real-time (that is not calendar time) as an integer. Another function makes it possible to set up the timer to generate an interrupt at $x$ time units in the future, where $x$ can be specified.
4.2 Why vanilla Linux kernel?

The vanilla Linux kernel 2.6.34 was chosen to implement the scheduling algorithms S-EKG [4] and NPS-F [5,6]. That kernel version provides the required mechanisms to satisfy the previously mentioned design principles: (i) each processor holds its own runqueue and it is easy to add new fields to it; (ii) it has already implemented red-black trees that are balanced binary trees whose nodes are sorted by a key and most the operations are done in $O(\log n)$ time; (iii) it has the high resolution timers infrastructure that offers a nanosecond time unit resolution, and timers can be set on a per-processor basis; (iv) it is very simple to add new system calls and, finally, (v) it comes with the modular scheduling infrastructure that easily enables adding a new scheduling policy to the kernel.

4.3 ReTAS implementation

The vanilla Linux kernel 2.6.34 has three native scheduling modules: RT (Real-Time); CFS (Completely Fair Scheduling) and Idle. Those modules are hierarchically organized by priority in a linked list; the module with highest priority is the RT, the one with the lowest is the Idle module. Starting with the highest priority module, the dispatcher looks for a runnable task of each module in a decreasing order of priority.

We added a new scheduling policy module, called ReTAS, on top of the native Linux module hierarchy, thus becoming the highest priority module. That module implements the S-EKG and NPS-F scheduling algorithms. The ReTAS implementation consists on a set of modifications to the Linux 2.6.34 kernel in order to support the S-EKG and NPS-F scheduling algorithms and also the cluster version of the NPS-F, called C-NPS-F [5]. These scheduling policies are identified by the SCHED_S_EKG and SCHED_NPS_F macros.

Since the assigning algorithm is executed prior to runtime, in the next sections we will focus only on the kernel implementation; that is, on the dispatching algorithms.

4.3.1 ReTAS tasks

To differentiate these tasks from other tasks present in the system, we refer to these tasks as ReTAS tasks. Listing 1 shows the pseudo-algorithm of ReTAS tasks. They are periodic tasks and are always present in the system. Each loop iteration is considered a job. Note that the first job of each task appears in the system at $time0 + offset$ ($time0$ is set equal to all tasks in the system) and the remaining jobs are activated according to the period. The delay_until function sleeps a task until the absolute time specified by arrival.

```c
arrival := time0 + offset;
while (true) {
    delay_until (arrival);
    arrival := arrival + period;
}
```

Listing 1: ReTAS task pseudo-algorithm.

In the Linux operating system a process is an instance of a program in execution. To manage all processes, the kernel uses an instance of struct taskstruct data structure for each process. In order to manage ReTAS tasks, some fields were added to the struct taskstruct data structure (see Listing 2). notional_cpu_id field is used to associate the task with the notional processor. Fields cpu1 and cpu2 are used to set the logical identifier of processor(s) in which the task will be executed on. The absolute deadline and also the arrival of each job are set on the deadline and arrival fields of the retas_jobparam data structure, respectively.

```c
struct retas_task {
    int notional_cpu_id;
    struct retas_task_param task_param{
        unsigned long long deadline; //D_i
    } task_param;
    struct retas_job_param job_param{
        unsigned long long deadline; //D_i
        unsigned long long arrival; //a_i
    } job_param;
    int cpu1;
    int cpu2;
};
```

Listing 2: Fields added to struct taskstruct kernel data structure.

4.3.2 Notional processors

As mentioned before, ReTAS tasks are assigned to notional processors. Therefore, notional processors act as a runqueue. Each notional processor is an instance of struct notional_cpu data structure (see Listing 3), which is identified by a numerical identifier (id). Field cpu is set with the logical identifier of the physical processor that, in a specific time instant, is executing a task from that notional processor. The purpose of the flag will be explained in
Section 4.3.5. Each notional processor organizes all ready jobs in a red-black tree, whose root is the field root_tasks, according to the job absolute deadline. The lock field is used to serialize the insertion and remotion operations over the red-black tree specially for notional processors that are executed by two processors. edf field points to the task with the earliest deadline stored in the red-black tree. Note that notional_cpu is a vector defined as global variable.

```c
struct notional_cpu{
    int id;
    atomic_t cpu;
    rawpinlock_t lock;
    struct rb_root root_tasks;
    struct task* struct edf;
    ...
}
```

Listing 3: struct notional_cpu data structure.

### 4.3.3 Timeslot reserves

Each processor needs to know the composition of its timeslot. So, per-processor an instance of the struct timeslot (Listing 4) is defined. Fields timeslot_length, begin_curr_timeslot, reserve_length and timer are used to set the time division into time reserves. They are also used to identify in each time reserve a given time instant t falls in. When a timer expires, the timer callback sets the current task to be preempted and this automatically triggers the invocation of the dispatcher. And taking into account the current reserve, the dispatcher (that will be described on the next section) tries to pick a task from either the notional processor pointed by notional_cpu (for the first option) or notional processor pointed by alt_notional_cpu (for the second option).

```c
struct timeslot_reserve{
    struct notional_cpu notional_cpu; //first option
    struct notional_cpu all_notional_cpu; //second option
    unsigned long long reserve_length;
    
    struct timeslot{
        unsigned long long timeslot_length;
        unsigned long long begin_curr_timeslot;
        struct timeslot_reserve notional_cpu;
        NR_CPUS_PER_CPU;
        struct hrtimer timer;
    }
```

Listing 4: struct timeslot data structure.

### 4.3.4 ReTAS scheduling module

In the vanilla Linux kernel each processor holds a runqueue of all runnable tasks assigned to it. The scheduling algorithm uses this runqueue to select the “best” process to be executed. The information for these processes is stored in a per-processor data structure called struct rq (Listing 5). Many functions that compose the Linux’s modular scheduling framework have an instance of this data structure as argument. Listing 5 shows the new data structures required by the ReTAS scheduling module added to the Linux native struct rq. The purpose of the struct timeslot data structure was described in the previous section.

```c
struct retas_rq {
    int post_schedule;
    struct timeslot timeslot;
    struct release release;
    struct resched_cpu resched_cpu;
};
```

Listing 5: struct retas_rq added to struct rq.

According to the rules of the Linux’s modular scheduling framework, each module must implement a set of functions specified in the sched_class data structure. Listing 6 shows the definition of retas_sched_class, which implements the ReTAS module. The first field (next), is a pointer to the next sched_class in the hierarchy. Since retas_sched_class is declared as the highest priority scheduler module that field points to the rt_sched_class, which implements the two POSIX real-time policies (SCHED_FIFO and SCHED_RR). The other fields are functions that act as callbacks to specific events.

```c
static const struct sched_class
    retas_sched_class = {
        next = &rt_sched_class,
        .enqueue_task = enqueue_task_retas,
        .dequeue_task = dequeue_task_retas,
        .check_preempt_curr = check_preempt_curr_retas,
        .pick_next_task = pick_next_task_retas,
        ...
    }
```

Listing 6: retas_sched_class scheduling class.

The enqueue_task_retas (Listing 7) is called whenever a RETAS job enters into a runnable state. It receives two pointers, one for the runqueue of the processor that is running this code (rq) and another to the task that is entering in a runnable state (p). Then, it updates the job absolute deadline by suming the job arrival time (this field is updated
through the job release procedure that will be de-
scribed Section 4.3.6) to the task relativedeadline,
and inserts it into the red-black tree of its notional
processor. Additionally, it checks if this job (in the
case of being a split task) could be executed by other
processor; that is, if it is a split task could happen
that when a job is released on this processor could
Correspond to its reserve on the other processor. If
that is the case, then an IPI is sent to the other pro-
cessor, using the resched function.

static void
equeue_task_retas(struct rq *rq, struct
task_struct *p, int wakeup, bool flag)
{
    int cpu;
    p->retas_task_job_param.deadline=p->retas_task
    .job_param.arrival=p->retas_task.
task_param.deadline;
    insert_task(&notional_cpus[p->retas_task.
        notional_cpu_id], p);
    cpu=check_for_running_on_other_cpus(p, rq->cpu);
    if(cpu==-1)
    {
        resched_cpu(cpu);
    }
    return;
}

Listing 7: enqueue_task_retas function.

When a ReTAS job is no longer runnable, then
the dequeue_task_retas function is called that un-
does the work of the enqueue_task_retas function
(see Listing 8); that is, it removes the task from the
notional processor.

static void
dequeue_task_retas(struct rq *rq, struct
task_struct *p, int sleep)
{
    remove_task(&notional_cpus[p->retas_task.
        notional_cpu_id], p);
    return;
}

Listing 8: dequeue_task_retas function.

As the name suggests, the check_preempt_curr
retas function (Listing 9) checks whether the cur-
rently running task must be preempted or not. This
function is called following the enqueuing or de-
queuing of a task and it checks if there is any
ReTAS jobs to be executed. If so, it checks if
that job is available; that is, if it is not being
executed by another processor (to handle this is-
ssue, we use the atomic cpu field defined on the
struct notional_cpu) it sets a flag that in-
dicates to the dispatcher that the currently run-
ning task must be preempted, otherwise it sets up a
local timer (defined in the struct resched_cpu
resched_cpu) to expire some time later (throughout
set_resched_cpu_timer_expires, which will trig-
ger, at timer expiration, the invocation of the dis-
patcher).

static void
check_preempt_curr_retas(struct rq *rq, struct
task_struct *p, int sync)
{
    struct task_struct *t=0;
    int cpu;
    t=get_edf_task(get_current_notional_cpu(&rq->
        retas_rq.timeslot));
    if(!t)
    {
        t=get_edf_task(get_current_all_notional_cpu(&
            rq->retas_rq.timeslot));
    }
    if(t)
    {
        if(t!=rq->curr)
        {
            cpu=is_executing_on_other_cpu(t->retas_task.
                notional_cpu_id, rq->cpu);
            if(cpu==0)
            {
                set_task_need_resched(rq->curr);
            }
        }
        else{
        set_resched_cpu_timer_expires(rq);
        }
    }
}

Listing 9: check_preempt_curr_retas function.

The pick_next_task_retas function selects the
job to be executed by the current processor (see List-
ing 10). This function is called by the dispatcher
whenever the currently running task is marked to
be preempted or when a task finishes its execution.
First, it tries to get highest priority ReTAS job from
the first notional processor and, if there is no job it
checks the second notional processor. If there is one
ReTAS job ready to be executed (and it is not the
current executing job) then, the next step is to lock
the notional processor to that processor (this is done
in the lock_notional_cpu function and this locking
mechanism will be described in Section 4.3.5). If this
notional processor is locked it sets up a local timer to
expire some time later and returns NULL, otherwise
it returns the pointer to that job.

static struct task_struct *
pick_next_task_retas(struct rq *rq)
{
    struct task_struct *t=0;
    int cpu;
    t=get_edf_task(get_current_notional_cpu(&rq->
        retas_rq.timeslot));
    if(!t){ //it is assumed that these tasks (af
        off_notional_cpu) execute only on this cpu
    t=get_edf_task(get_current_all_notional_cpu(&
        rq->retas_rq.timeslot));
    }
    if(t)
    {
        cpu=lock_notional_cpu(t->retas_task.
            notional_cpu_id, rq->cpu);
        if(cpu==0)
        {
            set_resched_cpu_timer_expires(rq, t->
                retas_task.notional_cpu_id);
        t=0;
        goto pick_ret;
        }
    }
    pick_ret:
    return t;
}

Listing 10: pick_next_task_retas function.
4.3.5 Locking a notional processor

Whenever there is a shared resource there is the need to create a synchronization mechanism to serialize the access to that resource. In this implementation a notional processor can be shared by up two physical processors. So, to serialize the access to those notional processors two functions are used (see Listing 11): lock_notional_cpu and unlock_notional_cpu. The locking is done in the pick_next_task_retas function. As it can be seen from Listing 11, first it identifies the physical processor that is executing tasks from that notional processor. Next, it tries to add one to the atomic_t flag variable using the atomic_add_unless kernel function. But this operation only succeeds if the value is not one, that is, if it is zero, otherwise, it fails. In the first case, success, it atomically adds one to the flag and sets the cpu variable with the logical identifier of the current processor. And this way it locks the notional processor to that processor. In the second case, failure, the notional processor could be locked by other processor or by the current processor. If it is locked by the current processor nothing changes, otherwise the logical number of the processor is returned and the dispatcher cannot pick any job from this notional processor.

In order to unlock the notional processor, whenever a ReTAS job is the prev or the next task in the context of the context_switch function, which is invoked by schedule function (see Listing 13), it will enforce the execution of the post_schedule_retas function (see Listing 12) to unlock the notional processor. Unlocking means setting the flag variable equal to zero and the cpu variable equal to −1.

```c
int lock_notional_cpu(int ncpu, int cpu)
{
    int ret=atomic_read(&notional_cpus[ncpu].cpu);
    if(atomic_add_unless(&notional_cpus[ncpu].flag, 1))
        atomic_set(&notional_cpus[ncpu].cpu, cpu);
    if(ret==cpu)
        return 1;
    return ret;
}

void unlock_notional_cpu(int ncpu, int cpu)
{
    int x;
    x=atomic_read(&notional_cpus[ncpu].cpu);
    if(x==cpu){
        atomic_set(&notional_cpus[ncpu].cpu, -1);
        atomic_set(&notional_cpus[ncpu].flag, 0);
    }
}
```

Listing 11: Lock and unlock notional processor functions.

```c
post_schedule_retas(struct rq *rq)
{
    int i, ncpu=-1;
    if(rq->retas_rq.post_schedule)
        if(rq->curr->policy==SCHED_EKG || rq->curr->policy==SCHED_NPSF)
            ncpu=rq->curr->retas_task.notional_cpu_id;
    for(i=0;i<rq->retas_rq(timeslot.
        nr_notional_cpus[i]+1);
    if(likely(rq->retas_rq(timeslot.
        notional_cpus[i].notional_cpu))
        if(rq->retas_rq(timeslot.notional_cpus[i].notional_cpu->id==ncpu)
            unlock_notional_cpu(rq->retas_rq.timeslot.
                notional_cpus[i].notional_cpu->id,rq->cpu);
        }
    }
}
```

Listing 12: post_schedule_retas function.

```c
static inline void
context_switch(struct rq *rq, struct task_struct *prev, *next)
{
    ... if(prev->policy==SCHED_EKG || prev->policy==SCHED_NPSF)
        if(rq->policy==SCHED_EKG || prev->policy==SCHED_NPSF)
            rq->retas_rq.post_schedule = 1;
            switch_to(prev, next, prev);
    }
```

Listing 13: context_switch and schedule functions.

4.3.6 Job release mechanism

The job release mechanism is supported by the struct release and is set per-processor. It is composed by a red-black tree and a timer. The idea is the following: a job is put in the waiting state, next, it is inserted into a red-black tree ordered by the absolute arrival time and, finally, a timer is set to expire at the earliest arrival time of all jobs stored into the red-black tree. When the timer expires, the job with earliest arrival time is removed from the red-black tree and its state changes to running, consequently,
it becomes ready. The timer is armed to expire at the earliest arrival time of remaining jobs stored into the red-black tree. This procedure is triggered by the `delay until` system call that specifies the absolute arrival time of a job. One feature of this mechanism is that the next release of a job is done on the processor where it finishes its execution; that is, where it executes the `delay until` system call.

5 From theory to practice

Usually, real-time scheduling algorithms for multiprocessor systems are supported by a set of assumptions that have no correspondence in the real-world. The evaluation of the discrepancy between theory and practice is here addressed taking into account two real-world phenomena: jitter and overhead. For convenience we define jitter as being the time from when an event must occur until it actually occurs. We have identified three sources of jitter: reserve, release and context switch. We also define overhead as the time that the current executing task is blocked by something else not related to the scheduling algorithm. We have identified two sources for the overhead: interrupts and release.

5.1 Sources of jitter

Theoretically, when a reserve ends one assumes that jobs are instantaneously switched, but in practice this is not true. Fig. 2 shows ResJ\(_{t,i,k}\), which represents the measured reserve jitter of job \(\tau_{t,i,k}\) and denotes the discrepancy between the time when the job \(\tau_{t,i,k}\) should (re)start executing (at the beginning of the reserve A, where A could be \(x\), \(N\) or \(y\)) and when it actually (re)starts. It should be mentioned that the timers are set up to fire when the reserve should begin, but, unfortunately, there is always a drift between this time and the time instant at which the timer fires. Then, the timer callback executes and, in most cases, sets the current task to be preempted triggering the invocation of the dispatcher. The dispatcher selects a task according to the dispatching algorithm and switches the current task with the selected task.

![FIGURE 2: Reserve jitter.](image)

In theory it is typically assumed that a release of a job is instantaneous and becomes ready immediately. In practice, however, something very different is observed. RelJ\(_{t,i,k}\) (release jitter of job \(\tau_{t,i,k}\)) denotes the difference in time from when the job \(\tau_{t,i,k}\) should arrive until it is inserted in the ready queue (see Fig. 3). The following steps take place in practice. First, a timer expires to wake up the job and, as mentioned before, there is always a drift between the time instant when the timer should expire and when it actually expires. Next, the job is removed from the release queue and inserted into the ready queue.

![FIGURE 3: Release jitter.](image)

A final important source of jitter is the context switching. A context switch is the procedure for switching processor from one job to another.
In theory, the time required to switch jobs is usually neglected. However, switching from one process to another requires a certain amount of time for saving and loading registers, among other operations. $C_{tsw_{i,k}}$ (context switch jitter of job $τ_{i,k}$) denotes the difference in time from when the job $τ_{i,k}$ should start executing until it actually (re)starts (see Fig. 4). Note that, we consider the context switch time incurred by the EDF scheduling policy, because the context switch time incurred by reserves are accounted for by $ResJ$.

![FIGURE 4: Context switch jitter.](image)

5.2 Sources of overhead

Usually, in theory, we ignore interrupts, but in practice they are one of main sources of timing unpredictability in this kind of system. In practice, when an interrupt arises the processor suspends the execution of the current job in order to execute the associated ISR. $IntO_{i,k}$ (interrupt overhead of job $τ_{i,k}$) denotes the time during which job $τ_{i,k}$ is prevented from executing due to the execution of an ISR (see Fig. 5).

When a job is executing on a processor, jobs of other tasks could appear in the system. In our implementation, to release a job, the processor stops what it is doing to release that job. $RelO_{i,k}$ (release overhead of job $τ_{i,k}$) denotes the time during which job $τ_{i,k}$ is prevented from executing due to the releases of other jobs (see Fig. 6).

![FIGURE 5: Interrupt overhead.](image)

![FIGURE 6: Release overhead.](image)

5.3 Experimental setup

In order to evaluate the discrepancy between theory and practice, we have conducted a range of experiments with 4-core (Intel(R) Core(TM) i7 CPU @ 2.67GHz) and 24-core (AMD OPeron (TM) Processor 6168 @ 1.90GHz) machines with real-time tasks executing empty for-loops. In order to make the environment more controlled, we (i) set runlevel to 1 (that is no windowing system), (ii) disabled network interface and also the filesystem journal mechanism, (iii) all interrupts are handled by the last processor and (iv) setup one non real-time task per core, as a background task, to ensure that the kernel idle threads never start executing.

We generated 17 random task sets. The period and utilization of tasks varied from 5 $ms$ up to 50 $ms$ and from 0.1 up to 1, respectively. The number of tasks varied from 6 up to 28 tasks in the case of the 4-core machine. The time duration of each experiment was approximately 1000 $s$. All task sets were...
scheduled using the S-EKG and NPS-F scheduling algorithms (which have comparable jitter/overheads). Since each experiment took 1000 s, the whole set of experiments took 34000 s.

5.4 Discussion of results

We collected the maximum values observed for each type of jitter and also for each type of overhead irrespective of the algorithm used (S-EKG or NPS-F). Table 1 presents the experimental results for: reserve jitter ($ResJ$), release jitter ($RelJ$), context switch jitter ($CtswJ$), the overhead of interrupt 20 ($IntO - irq20$) (related to the hard disk) and the overhead of tick. Note that, we do not directly present the release overhead. Rather, since, the release overhead is part of what is experienced as release jitter, we simply present the worst-case $RelJ$ (which also accounts for $RelO$). The column identified with $MAX_{max}$ gives the maximum value observed in all experiments. The third column ($AVG_{\tau_i}$) gives the average value experimented by the task that experienced the $MAX_{max}$ value. The fourth column ($MIN_{max}$) gives the minimum of the collected values (note that is the minimum of the maximum values). The last column displays the average value of the task that experienced the $MIN_{max}$ value. Before analyzing the results, we draw the attention of the reader to the time unit, $\mu s$, which means that the impact of those jitters/overheads is relatively small. Recall that the period of tasks in the various experiments varied from 5 ms up to 50 ms.

The highest $ResJ$ values were constantly experienced by split tasks. This is due to the task migration mechanism required for split tasks (described in Section 4). In that mechanism, if a task is not available, a timer is set to expire some time later. The value chosen for this delay was 5 $\mu s$.

The $MAX_{max} RelJ$ value is too high (31.834 $\mu s$), but comparing both $AVG_{\tau_i} (0.329 \mu s$ and $0.369 \mu s)$ observed shows that something prevented the release mechanism of doing that job release. The reason for this is related to the unpredictability of the underlying operating system. There are many sources of unpredictability in a Linux kernel: (i) interrupts are the events with the highest priority, consequently when one arises, the processor execution switches to handle the interrupt (usually interrupts arise in an unpredictable fashion); (ii) on Symmetric Multi Processing (SMP) systems there are multiple kernel threads running on different processors in parallel, and those can simultaneously operate on shared kernel data structures requiring serialization on access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions; (iv) the high resolution timer infrastructure is based on local Advanced Programmable Interrupt Controller (APIC), disabling and enabling local interrupts can disrupt the precision of that timer and, finally, (v) the hardware that Linux typically runs on does not provide the necessary determinism, which would permit the timing behavior of the system to be predictable with all latencies being time-bounded and known prior to run-time.

The same reasons could be given to explain the $MAX_{max} CtswJ$ value. However, usually the magnitude of $CtswJ$ is not too high, because this operation is done by the scheduler, which executes in a controlled context.

In Table 1, we present the overhead results of two ISRs: $irq20$ and $tick$ ($tick$ is a periodic timer interrupt used by the system to do a set of operations, like for instance invoking the scheduler). The reason for this is, $irq20$ can be configured to be executed by one specific processor but $tick$ cannot. In our opinion, it does not make sense to present other values besides $MAX_{max}$, because in these experiments this is a sporadic and rare event. In contrast, $tick$ is periodic with a frequency of approximately 1 ms. The values observed show that this overhead is very small.

<table>
<thead>
<tr>
<th></th>
<th>$MAX_{max} (\mu s)$</th>
<th>$AVG_{\tau_i} (\mu s)$</th>
<th>$MIN_{max} (\mu s)$</th>
<th>$AVG_{\tau_i} (\mu s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ResJ$</td>
<td>8.824</td>
<td>5.255</td>
<td>7.919</td>
<td>5.833</td>
</tr>
<tr>
<td>$RelJ$</td>
<td>31.834</td>
<td>0.329</td>
<td>10.029</td>
<td>0.369</td>
</tr>
<tr>
<td>$CtswJ$</td>
<td>2.218</td>
<td>0.424</td>
<td>0.587</td>
<td>0.305</td>
</tr>
<tr>
<td>$IntO - irq20$</td>
<td>24.226</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$IntO - tick$</td>
<td>0.571</td>
<td>0.272</td>
<td>0.408</td>
<td>0.243</td>
</tr>
</tbody>
</table>

TABLE 1: Experimental results (4-core machine).
Because of space limitations, in this paper our analysis is focused on 4-core machine results, in [10] some results of a set of experiments with the 24-core machine are presented. Nevertheless, Table 2 shows some results related to the ResJ and RelJ on the 24-core machine. The explanation for the MAX max values is the same that was given for the 4-core machine results. The AVG τ i values are in all cases higher than those for the 4-core machines. This is due to the speed of the processors: 4-core processors operate at 2.67GHz while 24-core processors operate at 1.9GHz.

### 6 Conclusions

We have presented the Real-time TAsk-Splitting scheduling algorithms (ReTAS) framework that implements S-EKG and NPS-F slot-based task-splitting scheduling algorithms. The main purpose of this framework is to show that slot-based task-splitting scheduling algorithms can be implemented in a real-operating system (using the vanilla Linux kernel) and work in practice. Using this framework we have identified and measured the real-operating system jitters and overheads. In spite of the unpredictability of the Linux kernel we observed a good correspondence between theory and practice. These good results are due to: (i) the controlled experimental environment; (ii) the use of the local high-resolution timers and (iii) the fact that these scheduling algorithms only involve very limited synchronization on shared system resources between processors.

### Acknowledgements

This work was supported by the CISTER Research Unit (608 FCT) and also by the REHEAT project, ref. FCOMP-01-0124-FEDER-010045 funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness) and by National Funds (PT), through the FCT - Portuguese Foundation for Science and Technology.

### References


Experience with Sporadic Server Scheduling in Linux: Theory vs. Practice

Mark J. Stanovich, Theodore P. Baker∗, An-I Andy Wang
Florida State University
Department of Computer Science, Florida, USA
{stanovic,baker,awang}@cs.fsu.edu

Abstract

Real-time aperiodic server algorithms were originally devised to schedule the execution of threads that serve a stream of jobs whose arrival and execution times are not known a priori, in a way that supports schedulability analysis. Well-known examples of such algorithms include the periodic polling server, deferrable server, sporadic server, and constant bandwidth server.

The primary goal of an aperiodic-server scheduling algorithm is to enforce a demand bound for each thread - that is, an upper bound on the amount of CPU time a thread may compete for in a given time interval. Bounding the demand of a given thread limits the interference that thread can inflict on other threads in the system experience in the competition for CPU time. Isolating the CPU-time demands of threads, known as temporal isolation, is an essential requirement for guaranteed resource reservations and compositional schedulability analysis in open real-time systems. A secondary goal of an aperiodic server is to minimize the worst-case and/or average response time while enforcing the demand bound. The theoretical aperiodic server algorithms meet both goals to varying degrees.

An implementation of an aperiodic server can yield performance significantly worse than its theoretical counterpart. Average response time is often higher, and even temporal isolation may not be enforced due to factors not found or considered in the theoretical algorithm. These factors include context-switching overheads, imprecise clocks and timers, preemption delays (e.g., overruns), and limits on storage available for bookkeeping.

This paper reports our experience implementing, in Linux, variations of the sporadic-server scheduling algorithm, originally proposed by Sprunt, Sha, and Lehoczky. We chose to work with sporadic-server scheduling because it fits into the traditional Unix priority model, and is the only scheduling policy recognized by the Unix/POSIX standard that enforces temporal isolation. While this paper only considers sporadic server, some lessons learned extend to other aperiodic servers including those based on deadline scheduling.

Through our experience, we show that an implemented sporadic server can perform worse than less complex aperiodic servers such as the polling server. In particular, we demonstrate the effects of an implementation’s inability to divide CPU time into infinitely small slices and to use them with no overhead. We then propose and demonstrate techniques that bring the performance closer to that of the theoretical sporadic-server algorithm. Our solutions are guided by two objectives. The primary objective is that the server enforce an upper bound on the CPU time demanded. The secondary objective is that the server provide low average-case response time while adhering to the server’s CPU demand bound. In order to meet these objectives, our solutions restrict the degree to which the server’s total CPU demand can be divided. Additionally, we provide mechanisms to increase the server’s ability to provide more continuous allocations of CPU demand.

Through a network packet service example, we show that sporadic server can be effectively used to bound CPU demand. Further, the efficiency of jobs served by sporadic server can be improved in terms of both reduced average-case response time and increased throughput.

∗Dr. Baker’s contributions to this paper are based on work supported by the National Science Foundation, while working at the Foundation.
1 Introduction

The roots of this paper are in experiments we did in 2007 on trying to schedule Linux device-driver execution in a way that conforms to an analyzable real-time scheduling model [3]. We found that the Unix SCHED_Sporadic scheduling policy is a potential improvement over SCHED_FIFO at any constant scheduling priority. Then, in subsequent studies we discovered that we needed to correct some technical defects in the POSIX definition of SCHED_Sporadic, which are reported in [5]. The paper describes our more recent efforts to deal with another phenomenon, having to do with preemption overhead and trade-offs between server throughput, server response time, and the ability to guarantee deadlines of other real-time tasks.

In a broader sense, this paper is about narrowing a gap that has developed between real-time operating systems and real-time scheduling theory. While a great deal is known about real-time scheduling in theory, very little of the theory can be applied in current operating systems. We feel that closer integration of operating systems implementation and scheduling theory is needed to reach a point where one can build open systems that reliably meet real-time requirements.

After some review of real-time scheduling theory and aperiodic servers, we discuss our experiences with implementing sporadic server scheduling, the problem of properly handling preemption overhead, and how we addressed the problem. We compare the performance of serving aperiodic workload by a polling server, a sporadic server, and a hybrid polling-and-sporadic server, using our implementation of the three scheduling algorithms in Linux. We conclude with a brief discussion of lessons learned and some further work.

2 Background

Any implementor of real-time operating systems needs to understand the basics of real-time scheduling theory, in order to understand the implications of implementation decisions. While this paper is not primarily about scheduling theory, we try to establish some theoretical background as motivation for our discussion of implementation issues.

Real-time scheduling theory provides analysis techniques that can be used to design a system to meet timing constraints. The analyses are based on abstract scheduling algorithms and formal models of workload and processing resources. The theory can guarantee that a set of timing constraints will always be satisfied, but only if an actual system conforms to the abstract models on which the analysis is based.

Real-time operating systems provide a run-time platform for real-time applications, including the mechanisms and services that schedule the execution of tasks on the processing unit(s). For timing guarantees based on real-time scheduling theory to apply to an application implemented using an operating system, there must be a close correspondence between the virtual execution platform provided by the OS and the abstract models and scheduling algorithms of the theory. The burden of achieving this correspondence falls on the OS and application developers.

The OS must provide mechanisms that allow development of applications that conform to the abstract models of the theory within bounded tolerances. In the case of a general-purpose operating system that supports the concept of open systems, such as Linux, the OS must go further, to provide firewall-like mechanisms that preserve conformance to the models when independently developed applications or components run alongside one another.

In real-time scheduling theory the arrival of a request for some amount of work is known as a job, and a logical stream of jobs is called a task. Some jobs have deadlines. The goal is to find a way to schedule all jobs in a way that one can prove that hard deadlines will always be met, soft deadlines will be met within a tolerance by some measure, and all tasks are able to make some progress at some known rate. To succeed, the theory must make some assumptions about the underlying computer platform and about the workload, i.e. the times at which jobs may arrive and how long it takes to execute them.

The best-behaved and best understood task model is a periodic task, whose jobs have a known worst-case execution time (WCET) and a known fixed separation between every pair of consecutive jobs, called the period. A periodic task also has an associated deadline, the point in time, relative to the arrival of a job, by which the job must complete execution. These workload parameters, along with others, can be used to determine whether all jobs can meet their timing constraints if executed according to certain scheduling algorithms, including strict preemptive fixed-task priority scheduling.

A key concept in the analysis of preemptive scheduling is interference. The nominal WCET of a job is based on the assumption that it is able to run to completion (i.e., until the corresponding thread
suspends itself) without interference from jobs of any other task. Showing that a job can complete within a given time window in the presence of other tasks amounts to bounding the amount of processor time the other tasks can steal from it over that interval, and then showing that this worst-case interference leaves enough time for the job to complete. The usual form of interference is preemption by a higher priority task. However, lower priority tasks can also cause interference, which is called priority inversion or preemption delay. Preemption delays may be caused by critical sections, imprecision in the OS timer mechanism, or any other failure of the kernel to adhere consistently to the preemptive fixed-priority scheduling model.

A system that supports the UNIX real-time API permits construction of threads that behave like a periodic task. The clock_nanosleep() function is one of several that provide a mechanism for suspending execution between one period and the next. Using the sched_setscheduler() function the application can request the SCHED_FIFO policy, and assign a priority. By doing this for a collection of periodic tasks, and choosing priorities sufficiently high to preempt all other threads, one should be able to develop an application that conforms closely enough to the model of periodic tasks and fixed task-priority preemptive scheduling to guarantee the actual tasks meet deadlines within some bounded tolerance.

Unfortunately, that is not enough. To support a reasonable range of real-time applications one needs to be able to handle a wider range of tasks. For example, a task may request CPU time periodically but the execution time requested may not be bounded, or the arrival of work may not be periodic. If such a task has high enough priority, the interference it can cause for other tasks may be unpredictable or even unbounded, causing other tasks to miss deadlines.

Aperiodic tasks typically have performance requirements that are soft, meaning that if there is a deadline it is stochastic, or occasional deadline misses can be tolerated, or under temporary overload conditions load shedding may be acceptable. So, while the CPU time allocated to the service of aperiodic tasks should be bounded to bound worst-case interference for other tasks, it should be provided in a way that allows the aperiodic task to achieve fast average response time under expected normal circumstances.

One example of an aperiodic task that requires fast average response time can be found in the paper by Lewandowski, et. al [3]. In this paper, a real-time task uses the network in its time-critical path to gather information. While it is desirable to receive all network packets, missing a few packets is not catastrophic. The difficulty lies in that the network receive path is shared by other tasks on the system, some with different deadlines and deadlines with no explicit deadlines.

Assuming a fixed-task-priority model, a priority must be chosen for the bottom level of network packet service. Processing the packets at a low or background priority does not work well because processing the packets may be delayed arbitrarily. Extended delay in network packet processing means that a real-time task waiting for the packets may miss an unacceptably large number of packets. Another option is to schedule the network packet processing at a high priority. However, the network packet processing now can take an unbounded amount of CPU time, potentially starving other tasks on the system and thereby causing missed deadlines. Therefore, a scheduling scheme is needed that provides some high-priority time to serve the aperiodic tasks; however, the high-priority time should be limited, preventing the packet processing from monopolizing the CPU. The bound on CPU time ensures other tasks have access to the CPU in a timely manner.

The key to extending analysis techniques developed for periodic tasks to this broader class of workloads is to ration processor time. It must be possible to force even an uncooperative thread to be scheduled in a way that the worst-case interference it causes other tasks can be modeled by the worst-case behavior of some periodic task. A number of scheduling algorithms that accomplish this have been studied, which we refer to collectively as aperiodic servers.

Examples of well-known aperiodic server scheduling algorithms for use in a fixed-task-priority scheduling environment include the polling and defeerrable servers [18], and the sporadic server [2]. There are also several examples for use with deadline scheduling, among which the constant bandwidth server has received considerable attention[17].

All these algorithms bound the amount of CPU time an aperiodic task receives in any time interval, which bounds the amount of interference it can cause other tasks, guaranteeing the other tasks are left a predictable minimum supply of CPU time. That is, aperiodic servers actively enforce temporal isolation, which is essential for an open real-time execution platform.

The importance of aperiodic servers extends beyond the scheduling of aperiodic tasks. Even

---

1 Of course, careful attention must be given to other details, such as handling critical sections.
the scheduling of periodic tasks may benefit from the temporal isolation property. Aperiodic server scheduling algorithms have been the basis for a rather extensive body of work on open real-time systems, appearing sometimes under the names virtual processor, hierarchical, or compositional scheduling. For example, see [4, 9, 10, 11, 12, 13, 14].

In this paper, we limit attention to a fixed-task-priority scheduling environment, with particular attention to sporadic-server scheduling. The primary reason is that Linux for the most part adheres to the UNIX standard and therefore supports fixed-task-priority scheduling. Among the well-known fixed-task-priority aperiodic-server scheduling algorithms, sporadic-server scheduling is theoretically the best. It also happens to be the only form of aperiodic-server scheduling that is recognized in the UNIX standard.

A polling server is one way of scheduling aperiodic workloads. The polling server is a natural extension to the execution pattern of a periodic task. Using a polling server, queued jobs are provided CPU time based on the polling server's budget, which is replenished periodically. If no work is available when the polling server is given its periodic allocation of CPU time, the server immediately loses its budget. Similarly, if the budget is partially used, and no jobs are queued, the polling server gives up the remainder of the budget.

FIGURE 1: Example usage and replenishment of sporadic server's budget.

A sporadic server is a thread that is scheduled according to one of the variants of the original sporadic server algorithm introduced by Sprunt, Sha, and Lehoczky [2]. While many variants exist, the basic idea is the same. A sporadic server has a budget, replenishment period, and scheduling priority. When the sporadic server uses the CPU, the amount of time used is deducted from its budget. The amount of CPU time consumed is restored to the budget one replenishment period in the future, starting from the instant when the sporadic server requested CPU time and had budget. The operation to restore the budget at a given time in the future, based on the amount of time consumed, is known as a replenishment. Once the server uses all of its budget, it can no longer compete for CPU time at its scheduling priority.

The objective of the sporadic-server scheduling algorithm is to limit worst-case system behavior such that the server's operation can be modeled, for schedulability analysis of other tasks, as if it were a periodic task. That is, in any given sliding time window, the sporadic server will not demand more CPU time than could be demanded by a periodic task with the same period and budget. A secondary goal of the sporadic server is to provide fast average response time for its jobs.

With regard to minimizing average response time, a sporadic server generally outperforms a polling server. The advantage with a sporadic server is that jobs can often be served immediately upon arrival, whereas with a polling server jobs will generally have to wait until the next period to receive CPU time. Imagine a job arrival that happens immediately after the polling server's period. The job must wait until the following period to begin service, since the polling server immediately forfeits its budget if there are no jobs available to execute. A sporadic server, on the other hand, can execute the job immediately given that its budget can be retained when the server's queue is empty. The ability to retain budget allows the server to execute more than once during its period, serving multiple jobs as they arrive. Aperiodic servers that can hold on to their budget until needed are known as bandwidth-preserving servers.

3 Implementation

Several variants of the original sporadic-server algorithm have been proposed, including the POSIX SCHED\_SPORADIC [7], and more recently two variants that correct defects in the POSIX version [5, 8]. Differences include how they handle implementation constraints such as limited space to store replenishment operations, overruns, and preemption costs.
The scheduling algorithm followed by our implementation is described in [15], which is an updated version of [5] including corrections for errors in the pseudo-code that were identified by Danish et al. in [4].

Correct operation of a sporadic server results in bounded interference experienced by lower-priority tasks. In order to measure the interference, we used Regehr’s “hourglass” technique [6], which creates an application-level process that monitors its own execution time without requiring special operating system support. The hourglass process infers the times of its transitions between executing and not executing by reading the clock in a tight loop. If the time between two successive clock values is small, the assumption is that the process was not preempted. However, if the difference is large, the thread was likely preempted. This technique can be used to find preemption points and thereby determine the time intervals when the hourglass process executed. From this information, the hourglass process can calculate its total execution time.

Using the hourglass approach, we were able to evaluate whether an implemented sporadic server actually provides temporal isolation. That is, if we schedule the hourglass task with a priority below that of the sporadic server (assuming there are no other higher-priority tasks in the system), the hourglass task should be able to consume all of the CPU time that remains after the sporadic server used all of its budgeted high-priority time. The CPU time available to the hourglass task should, ideally, be one hundred percent minus the percentage budgeted for the sporadic server, viewed over a large enough time window. Therefore, if we schedule a sporadic server with a budget of 1 millisecond and a period of 10 milliseconds, and there are no other tasks with priority above the sporadic server and hourglass tasks, the hourglass task should be able to consume at least 90% of the CPU time, i.e., 9 milliseconds in any window of size 10 milliseconds. In reality, other activities such as interrupt handlers may cause the interference experienced by the hourglass task to be slightly higher.

To evaluate the response time characteristics of our sporadic server, we measured the response time of datagram packets sent across a network. The response time of a packet is measured by the time difference between sending the packet on one machine, \( m_1 \), and receiving the packet by another, \( m_2 \). More specifically, the data portion of each packet sent from \( m_1 \) contains a timestamp, which is then subtracted from the time the packet is received by the UDP layer on \( m_2 \). In our setup, \( m_1 \) periodically sends packets to \( m_2 \). The time between sending packets is varied in order to increase the load experienced by the network receive thread on \( m_2 \). The receive thread on \( m_2 \) is scheduled using either the polling server, sporadic server, or SCHED_FIFO [7] scheduling policies. In our experiments, \( m_2 \) is running Linux 2.6.38 with a ported version of softirq threading found in the 2.6.33 Linux real-time patch. \( m_2 \) has a Pentium D 830 processor running at 3GHz with a 2x16KB L1 cache and a 2x1MB L2 cache. 2GB of RAM are installed. The kernel was configured to use only one core, so all data gathered is basically equivalent to a uniprocessor system.

![FIGURE 2: Response time using different scheduling policies.](image1)

![FIGURE 3: sirq-net-rx thread CPU utilization using different scheduling policies.](image2)

---

5 The clocks for the timestamps on \( m_1 \) and \( m_2 \) are specially synchronized using a dedicated serial connection.

6 SCHED_FIFO differs from the other in allowing thread of sufficiently high priority to execute arbitrarily long without preemption.
Scheduling the Linux network receive thread (i.e., \texttt{sirq-net-rx}) using various scheduling policies affects the average response time of received network packets. One would expect that the polling server would result in higher average response times than \textit{SCHED_FIFO} or sporadic server and that sporadic server and \textit{SCHED_FIFO} should provide similar average response times until sporadic server runs out of budget.

In our experiment, sporadic server and polling server are both given a budget of 1 millisecond and a period equal to 10 milliseconds. The sporadic server’s maximum number of replenishments is set to 100. The hourglass task is scheduled using \textit{SCHED_FIFO} scheduling at a real-time priority lower than the priority of the network receive thread. Each data point is averaged over a 10 second interval of sending packets at varied rates. The CPU utilization and response time for the described experiment are shown in Figures 2 and 3.

One would expect that if the sporadic server and polling server both were budgeted 10% of the CPU, the lower-priority hourglass task should be able to consume at least 90% of the CPU time regardless of the load. However, the data for the experiment shows that the sporadic server is causing much greater than 10% interference. The additional interference is the consequence of preemptions caused by the server. Each time a packet arrives the sporadic server preempts the hourglass task, thereby causing two context switches for each packet arrival. Given that the processing time for a packet is small (2-10 microseconds) the server will suspend itself before the next packet arrives. In this situation, the aggregate time for context switching and other sporadic server overhead such as using additional timer events and running the sporadic Sever-related accounting becomes significant. For instance, on the receiving machine the context-switch time alone was measured at 5-6 microseconds using the \texttt{latctx} LMbench program[1].

The overhead associated with preemption causes the additional interference that is measured by the lower-priority hourglass task.\textsuperscript{7}

A snapshot of CPU execution time over a 500 microsecond time interval was produced using the Linux Trace Toolkit (LTTng)[16] and is shown in Figure 4. The top bar is the \texttt{sirq-net-rx} thread and the bottom bar is the lower-priority hourglass measuring task. This figure shows that the CPU time of both tasks is being finely sliced. The small time slices cause interference for both the lower-priority and sporadic server thread that would not be experienced if the threads were able to run to completion.

### 3.1 Accounting for Preemption Overhead

To ensure that no hard deadlines are missed, and even to ensure that soft deadlines are met within the desired tolerances, CPU time interference due to preemptions must be included in the system’s schedulability analysis. The preemption interference caused by a periodic task can be included in the analysis by adding a preemption term to the task’s worst-case execution time (\textit{WCET}) that is equal to twice the worst-case context switch cost – one for switching into the task and one for switching out of the task.\textsuperscript{8}

Assuming all tasks on the system are periodic, this is at least a crude way of including context-switch time in the schedulability analysis.

A sporadic server can cause many more context switches than a periodic task with the same parameters. Rather than always running to completion, a sporadic server has the ability to self-suspend its execution. Therefore, to obtain a safe \textit{WCET} bound for analysis of interference\textsuperscript{9}, one would have to determine the maximum number of contiguous “chunks” of CPU time the sporadic server could request within any given period-sized time interval. The definition of sporadic server scheduling given in scheduling theory publications does not place any such restriction on the number of CPU demand chunks and thus imposes no real bound on the \textit{WCET}. In order to bound the number of preemptions, and thereby bound the time spent context switching, most implemented variations of sporadic server limit the maximum number of pending replenishments, denoted by \textit{max_repl}. Once \textit{max_repl} replenishments are pending, a sporadic server will be prevented from executing until one of the future replenishments arrives.

\textsuperscript{7}The lower-priority thread does not measure much of the cache eviction and reloading that other applications may experience, because its code is very small and typically remains in the CPU’s cache. When cache effects are taken into account, the potential interference penalty for each preemption by a server is even larger.

\textsuperscript{8}This is an intentional simplification. The preemption term should include all interferences caused by the sporadic server preempting another thread, not only the direct context-switch time, but also interferences such as the worst-case penalty imposed by cache eviction and reloading following the switch. For checking the deadline of a task, both “to” and “from” context switches need to be included for potentially preemtiting task, but only the “to” switch needs be included for the task itself.

\textsuperscript{9}From this point on we abuse the term \textit{WCET} to stand for the maximum interference that a task can cause for lower-priority tasks, which includes not just the maximum time that the task itself can execute, but also indirect costs, such as preemption overheads.
Using \textit{max repl}, the maximum context-switching time per period of a sporadic server is two times the \textit{max repl}. Using this logic, and assuming that the actual context-switch costs are added on top of the servers budget, a worst-case upper bound on the interference that can be caused by a sporadic server task could be written as:

\[ SS_{\text{budget}} + 2 \times \text{max repl} \]

Accounting for the cost due to preemptions is important in order to ensure system schedulability; however, adding preemption cost on top of the server’s budget as above results in over-provisioning. That is, if a sporadic server does not use \textit{max repl} number of replenishments in a given period a worst-case interference bound derived in this way is an over-estimate. At the extreme, when a sporadic server consumes CPU time equal to its budget in one continuous chunk, the interference only includes the cost for two context switches rather than two times \textit{max repl}. However, the server cannot make use of this windfall to execute jobs in its queue because the context switch cost was not added to its actual budget.

We believe a better approach is to account for actual context-switch costs while the server is executing, charging context switch costs caused by the server against its actual budget, and doing so only when it actually preempts another task. In this approach the \textit{SS budget} alone is used as the interference bound for lower-priority tasks. Accounting for context-switching overhead is performed on-line by deducting an estimate of the preemption cost from the server’s budget whenever the server causes a preemption. Charging the sporadic server for preemption overhead on-line requires that the preemption interference be known. Determining an appropriate amount to charge the server for preemptions can be very difficult, as it depends on many factors. In order to determine an amount to charge sporadic server for a preemption, we ran the network processing experiment under a very heavy load and extracted an amount that consistently bounded the interference of sporadic server to under 10%. While such empirical estimation may not be the ideal way to determine the preemption interference, it gave us a reasonable value to verify that charging for preemptions can bound the interference.

Charging for preemptions on-line requires that the preemption interference be known. Determining an appropriate amount to charge the server for preemptions can be very difficult, as it depends on many factors. In order to determine an amount to charge sporadic server for a preemption, we ran the network processing experiment under a very heavy load and extracted an amount that consistently bounded the interference of sporadic server to under 10%. While such empirical estimation may not be the ideal way to determine the preemption interference, it gave us a reasonable value to verify that charging for preemptions can bound the interference.

The network experiment was performed again, this time charging sporadic server a toll of 10 microseconds each time it caused a preemption. Figure 5 shows the results for the experiment and demonstrates that time interference for other lower-priority tasks can be bounded to 10%, that is, the server’s budget divided by the period.
3.2 Preemption Overhead

BOUNDING THE INTERFERENCE THAT AN APERIODIC WORKLOAD CAUSES FOR OTHER TASKS IS THE PRIMARY OBJECTIVE OF APERIODIC SERVER SCHEDULING; HOWEVER, ONE WOULD ALSO LIKE TO SEE FAST AVERAGE RESPONSE TIME. FIGURE 6 SHOWS THAT UNDER HEAVY LOAD, THE AVERAGE RESPONSE TIME OF PACKETS WHEN USING SPORADIC-SERVER SCHEDULING IS ACTUALLY WORSE THAN THAT OF A POLLING SERVER WITH THE SAME PARAMETERS. FOR THIS EXPERIMENT, NOT ONLY IS THE SPORADIC SERVER’S AVERAGE RESPONSE TIME HIGHER, BUT AS THE LOAD INCREASES UP TO 45% OF THE PACKETS WERE DROPPED.

The poor performance of the sporadic server is due to a significant portion of its budget being consumed to account for preemption costs, leaving a smaller budget to process packets. If all of the packets arrived at the same time, the processing would be batched and context switching would not occur nearly as often. However, due to the spacing between packet arrivals, a large number of preemptions occur. A polling server on the other hand has a much larger portion of its budget applied to processing packets, and therefore does not drop packets and also decreases the average response time.

Based on the poor performance of a sporadic server on such workloads one might naively jump to the conclusion that, in general, a polling server is a much better choice. Actually, there is a trade-off, in which each form of scheduling has its advantage. Given the same budget and period, a sporadic server will provide much better average-case response time under light load, or even under a moderate load of large jobs, but can perform worse than the polling server for certain kinds of heavy or bursty workloads.

It turns out that the workload presented by our packet service example is a bad one for the sporadic server, in that a burst of packet arrivals can fragment the server budget, and then this fragmentation becomes “locked in” until the backlog is worked off. Suppose a burst of packets arrives, and the first $max_{repl}$ packets are separated by just enough time for the server to preempt the running task, forward the packet to the protocol stack, and resume the preempted task. The server’s budget is fragmented into $max_{repl}$ tiny chunks. Subsequent packets are buffered (or missed, if the device’s buffer overflows), until the server’s period passes and the replenishments are added back to its budget. Since there is by now a large backlog of work, the server uses up each of its replenishment chunks as it comes due, then suspends itself until the next chunk comes due. This results in a repetition of the same pattern until the backlog caused by the burst of packets has been worked off. During this overload period, the sporadic server is wasting a large fraction of its budget in preemption overhead, reducing its effective bandwidth below that of a polling server with the same budget and period. There is no corresponding improvement in average response time, since after the initial $max_{repl}$ fragmentation, the reduced bandwidth will cause the response times to get worse and worse.

3.3 Reducing the Impact of Preemption Overhead

A hybrid server combining the strengths of polling and sporadic servers may be a better alternative than choosing either one. In this approach, a sporadic server is used to serve light loads and a polling server to serve heavy loads.

Sporadic-server scheduling supports a polling-like mode of operation. When the $max_{repl}$ parameter value is one, only one preemption is permitted per period. Switching to the polling-like mode of operation is just a matter of adjusting $max_{repl}$ to 1.

When changing modes of operation of the sporadic server in the direction of reducing $max_{repl}$, something must be done if the current number of pending replenishments would exceed $max_{repl}$. One approach is to allow the number of pending replenishments to exceed $max_{repl}$ temporarily, reducing it by one each time a replenishment comes due. Another approach is to implement the reduction at once, by coalescing pending replenishments. This is

---

No packets were dropped by the other servers.

---

FIGURE 6: Accounting for context-switching overhead.
similar to the classical mode-change scheduling problem, in that one must be careful not to violate the assumptions of the schedulability analysis during the transition. In the case of a sporadic server the constraint is that the server cannot cause any more interference within any time window than would be caused by a periodic task with execution time equal the server budget and period equal to the server’s budget replenishment period, including whatever adjustments have been made to the model to allow for context-switch effects. We call this the sliding window constraint for short.

In order to maintain the sliding-window constraint during the mode change, one can think in terms of changing the times associated with pending replenishments. Consolidating the replenishment times would allow the creation of a single replenishment with an amount equal to the server’s initial budget. To guard against violating the sliding-window constraint, the replenishment time of any replenishment must not be moved earlier in time. One approach is to coalesce all replenishments into the replenishment with a time furthest in the future, resulting into a single replenishment with an amount equal to the server’s initial budget as shown in Figures 7 and 8.

Switching from sporadic server to a polling-like server should be performed if the server is experiencing heavy load. The ideal switching point may be difficult to detect. For instance, a short burst may be incorrectly identified as the onset of a heavy load and the early switching may cause the server to postpone a portion of its budget that could have been used sooner. Conversely, delaying the switch may mean that time that could have been used to serve incoming jobs is wasted on preemption charges.

While an ideal switching point may not be possible to detect beforehand, one reasonable indicator of a heavy load is when sporadic server uses all of its budget. That is the point when a sporadic server is blocked from competing for CPU time at its scheduling priority. At this point the server could switch to its polling-like mode of operation.

A possible event to indicate when to switch back to the sporadic server mode of operation is when a sporadic server blocks but still has available budget. This point in time would be considered as entering a period of light load and the max_repl could be reinstated.
Implementation of the switching mechanism described above is relatively simple. The replenishments are coalesced into one when the server runs out of budget but still has work. The single replenishment limit will remain enforced until the sporadic server is suspended and has budget, a point in time considered to be an indication of light load. So, the polling-like mode of operation will naturally transition back to the original sporadic server mode of operation.

Immediately coalescing all replenishments may be too eager. Loads that are between light and heavy may experience occasional or slight overloads that require only slightly more CPU time. In this case, converting all potential preemption charges, by delaying replenishments, into CPU time to serve packets is too extreme. Therefore, to perform better under a range of loads one approach is to coalesce only two replenishments for each overload detection. Using this method allows the sporadic server to naturally find an intermediate number of replenishments to serve packets efficiently without wasting large portions of its budget on preemption charges.

The performance data for the two coalescing methods, immediate and gradual, are shown in Figures 9 and 10. These figures show the advantage of transitioning between sporadic-server and polling-like mode of operation. Under light load until approximately 4500 pkts/sec, the sporadic server has the same response time as \texttt{SCHED\_FIFO} scheduling. However, once the load is heavy enough the sporadic server is forced to limit the amount of CPU demand and therefore the response time begins to increase to that of a polling server. There is not enough CPU budget to maintain the low average response time.

Immediate coalescing provides a gradual transition to polling-like behavior with the immediate coalescing has a much faster transition to the polling server’s response time performance. The better performance of the gradual coalescing is due to the server making better use of the available budget. With immediate coalescing, when the server transitions to the polling-like mode the CPU utilization drops, as one would expect of sporadic server where the \texttt{max\_repl} is set to 1. However, with gradual coalescing the server continues to use its available budget to pay for preemption costs and serve some jobs earlier, which results in lower response times.

4 Conclusion

Any open real-time operating system needs to provide some form of aperiodic-server scheduling policy, in order to permit temporal isolation of tasks, and to provide a real-time virtual processor abstraction that can support fault-tolerant compositional schedulability analysis. The only standard Unix scheduling policy with these properties is Sporadic Server.

We have described our experiences implementing and using a variation of the Sporadic Server scheduling algorithm in Linux. Our experience demonstrates that sporadic server scheduling can be an effective way to provide a predictable quality of service for aperiodic jobs while bounding the interference that the server thread can cause other tasks, thereby supporting schedulability analysis. However, this goal cannot be achieved without consideration of some subtle implementation issues that are not addressed in the theoretical formulations of sporadic server that have been published.

Neither the published theoretical versions of Sporadic Server nor the POSIX/Unix formulation consider all the interference effects we found on a real implementation. In particular, fine grained time slicing degrades the performance of the sporadic server thread and can cause interference for other threads on the system to significantly exceed the assumptions of the theoretical model. This interference is mainly due to a sporadic server being able to use CPU time in arbitrarily small time slices. Such fine time slicing not only increases the interference that the server inflicts on tasks that it preempts, but also degrades the throughput of the server itself. Through network service experiments we showed that the interference caused by a sporadic server can be significant enough to cause other real-time threads to miss their deadlines.

Charging a sporadic server for preemptions is an effective means to limit the CPU interference. The charging for preemptions can be carried out in several ways. We chose an on-line approach where the server is charged when it preempts another thread. Charging the server only when it actually preempts not only bounds the CPU time for other tasks, but allows the server to use its budget more effectively. That is, rather than accounting for the additional interference by inflating the nominal server budget (over the implemented server budget) in the schedulability analysis, we charge the server at run time for the actual number of preemptions it causes. In

\footnote{While a literal implementation of the abstract description of \texttt{SCHED\_SPORADIC} in the Unix standard is not practical and would not support schedulability analysis, we feel that the corrections described in this paper and [15] fall within the range of discretion over details that should be permitted to an implementor.}

228
this way the server’s actual interference is limited to its actual CPU time budget, and we do not need to use an inflated value in the schedulability analysis. Since the preemption charges come out of the server’s budget, we still need to consider preemption costs when we estimate the worst-case response time of the server itself. However, if we choose to over-provision the server for worst-case (finely fragmented) arrival patterns it actually gets the time and can use it to improve performance when work arrives in larger chunks.

The ability to use small time slices allows a sporadic server to achieve low average response times under light loads. However, under a load of many small jobs, a sporadic server can fragment its CPU time and waste a large fraction of its budget on preemption charges. A polling server, on the other hand, does not experience this fragmentation effect, but does not perform as well as sporadic server under light load. To combine the strengths of both servers, we described a mechanism to transition a sporadic server into a polling-like mode, thereby allowing sporadic server to serve light loads with good response time and serve heavy loads with throughput similar to a polling server. The data for our experiments show that the hybrid approach performs well on both light and heavy loads.

Our recent experiences reinforce what we learned in prior work with sporadic-server scheduling in Linux [5]. There are devils in the details when it comes to reducing a clever-looking theoretical algorithm to a practical implementation. To produce a final implementation that actually supports schedulability analysis, one must experiment with a real implementation, reflect on any mismatches between the theoretical model and reality, and then make further refinements to the implemented scheduling algorithm until there is a match that preserves the analysis. This sort of interplay between theory and practice pays off in improved performance and timing predictability.

We also believe our experience suggests a potential improvement to the “NAPI” strategy employed in Linux network device drivers for avoiding unnecessary packet-arrival interrupts. NAPI leaves the interrupt disabled so long as packets are being served, re-enabling it only when the network input buffer is empty. This can be beneficial if the network device is faster than the CPU, but in the ongoing race between processors and network devices the speed advantage shifts one way and another. For our experimental set-up, the processor was sufficiently fast that it was able to handle the interrupt and the sirq

net-rx processing for each packet before the next arrived, but the preemption overhead for doing this was still a problem. By waiting for several packets to arrive, and then processing them in a batch, the polling server and our hybrid server were able to handle the same workload with much less overhead. However, the logical next step is to force a similar waiting interval on the interrupt handler for the network device.

While we have not experimented with deadline-based aperiodic servers in Linux, it appears that our observations regarding the problem of fitting the handling of context switch overheads to an analyzable theoretical model should also apply to the constant bandwidth server, and that a similar hybrid approach is likely to pay off there.

In future work, we hope to explore additional variations on our approach to achieving a hybrid between polling and sporadic server, to see if we can improve performance under a range of variable workloads. We are considering several different mechanisms, including stochastic, for detecting when we should change modes of operation as the system moves between intervals of lighter and heavier load. We also plan to explore other aperiodic servers and determine how much interference preemptions cause. For example, it appears that a constant bandwidth server would suffer the same performance problems as a sporadic server when the workload causes budget fragmentation. We also plan to investigate the preemption interference due to cache eviction and reloading. The threads used in our experiments access relatively small amounts of data and therefore do not experience very large cache interferences. This is not true for all applications, and the cache effects on such applications will need to be bounded. While limiting the number of replenishments does reduce the cache effect, better mechanisms are needed to reduce the ability of sporadic server to cause cache interferences.

Other questions we are considering include whether it is practically feasible to schedule multiple threads using a single sporadic-server budget, and how well sporadic-server scheduling performs on a multi-core system with thread migration.

References


How to cope with the negative impact of a processor’s energy-saving features on real-time capabilities?

Carsten Emde
Open Source Automation Development Lab (OSADL) eG
Aichhalder Str. 39, 78713 Schramberg, Germany
C.Emde@osadl.org

Abstract

In the early days of using computers for determinism-critical tasks, processors mostly were suitable for this purpose, since instruction execution was in sync with the clock frequency. This made it possible to correctly predict the execution time of a given code segment. With the rapidly increasing need for processing power, deterministic execution was abandoned in favor of throughput. In consequence, the peak processing power of a today’s state-of-the-art multi-core processor is about 1,000,000 times greater than that of a standard processor 30 years ago. The worst-case performance, however, only improved by a factor of about 10 - and even this may require specific configuration of the processor and the operating system. The main reasons for the lack of progress of the worst-case performance are the introduction of caches and energy-saving features. While the negative impact of caching on the worst-case performance could be studied in recent years and can now be handled reasonably well, the details of energy-saving are less well known. Although energy-saving most of the time boils down to switching off or at least throttling down unneeded processor components, such mechanisms can be implemented in various ways and locations and are often undocumented. It was, therefore, the aim of this project to investigate the latency behavior of modern energy-saving processors and to provide recommendations how to disable or circumvent energy-saving.

To investigate the impact of energy-saving, latency measurements no longer were performed in a short closed loop such as when using the cyclic test utility but with randomly occurring interrupt triggers at idle state of the processor. This could lead to long latencies which were then attempted to be reduced by specific processor and Linux kernel configurations.

As a result, we now can recommend a number of individual processor settings and configuration items of the Linux kernel to optimize the worst-case system latency when modern energy-saving processors are used. In some cases, however, it was not possible to disable any deleterious effect of energy-saving on a processor’s latency, although we tried very hard. Thus, we urgently appeal to semiconductor manufacturers to - whatever mechanisms they invent to reduce power consumption - provide a way to switch them off, if they adversely affect response time. In many cases, power-saving and fast reaction to asynchronous events exclude each other. It is, however, still possible to obtain a deterministic response while powersaving is enabled, but it must be taken into account that the worst-case latency of such systems may be considerably prolonged.

1 Mechanisms of power-saving

There are a number of good reasons to reduce the power consumption of microprocessors

- Reducing the energy consumption as part of the general ecological imperative
- Allowing the creation of battery-powered devices
- Prolonging the lifetime of systems through reduced thermal stress
- Reducing the need of fans and preventing damage from defective fans
Reducing the need of dust filters and preventing damage from filters that were forgotten to clean or to replace.

The semiconductor industry is using two completely different approaches to provide processors with reduced energy consumption:

- Using smaller structures and more efficient isolation material to reduce current demand and leakage current
- Slowing down or switching off parts of the processor when idle

While the former approach is generally welcome and normally does not have any impact on the response time of a processor, the latter is relevant in the context of real-time systems that try to achieve a minimum worst-case latency. It is, therefore, important to analyze a processor with respect to the implemented mechanisms for energy saving when selecting and configuring it for a system that relies on real-time computing.

2 Slowing down or switching off parts of the processor

To slow down or switch off parts of the processor, three different general mechanism are employed:

- Throttling
- Sleep states
- Undisclosed internal mechanisms

This article will present methods to analyze the various mechanisms and present procedures to prevent them from interfering with a system’s real-time capabilities. In all cases, mainline Linux with the PREEMPT_RT patches was used, kernel version 2.6.33.15-rt31.

3 Material and methods

To study the effect of the various power-saving mechanisms, the cyclic test utility was used. It is available from a git repository but also part of many Linux distributions where the related package usually is called rt-tests.

For the analysis of the effect of throttling on the worst-case latency of a processor, the cyclic test utility could be used in its original version. The utility was run as usual on systems with and without enabled power-saving. At least 100 million test cycles were run to obtain reliable results of the worst-case latency.

For the analysis of the effect of idle states on the worst-case latency of a processor, the cyclic test utility was expanded. In a first step, the -i or --interval= option that is used to define the duration of the measurement cycle was allowed to accept a range of a lowest and a highest interval duration. When a range is specified using this newly defined option format, an individual duration is determined for every test interval by using a uniformly distributed logarithmic random value between the lowest and and the highest duration specified. The interval is displayed in the output line to monitor the behavior of this functionality.

In a second step, a two-dimensional histogram was implemented that is activated when the --histogram= option is specified. This histogram stores the frequencies of latency samples per interval duration. This makes it possible to differentiate recorded latency values with respect to the duration of the preceding idle time of the processor. It is expected that the latency values would not depend on the duration of the test interval in processors without any (or completely disabled) power-saving mechanism, whereas in processors with active power-saving, the latency values would be the higher, the longer the preceding period of quiescence was.

4 Throttling

4.1 Principle

The clock frequency of many modern processors can be adjusted to reflect a system’s load requirements which reduces power consumption when the processor is less busy or even idle. The Linux kernel can optionally be compiled with the cpufreq subsystem to manage CPU frequency scaling also known as throttling. It provides, among others, the ondemand and the performance scaling governor. Manipulation of the CPU frequency scaling is done through the sys virtual file system in the /sys/devices/system/cpu/cpuN/cpufreq directory. The available frequencies are listed in the file scaling_available_frequencies and the available governors in the file scaling_available_governors. Writing the name of one of the available governors to the file...
scaling governor will select it, e.g. to set CPU #0 to full speed:

cd /sys/devices/system/cpu/cpu0/cpufreq
echo performance >scaling_governor

4.2 Effect of throttling setting on latency

Many of the processors tested show a distinct increase of the worst-case latency, if throttling is enabled and its clock frequency is allowed to be decreased to the specified minimum. A typical result is shown in Figure 1 that was obtained on an AMD two-core G-Series processor running either at 1,400 MHz (performance governor) or at 583 MHz to 1,400 MHz (ondemand governor) depending on the load.

![Figure 1: Typical worst-case latency analysis with throttling disabled (performance scaling governor with a constant clock frequency of 1,400 MHz) and throttling enabled (ondemand scaling governor with a minimum clock frequency of 583 MHz)](image)

4.3 Recommendation

If the fastest possible reaction of a processor to external events is required, any throttling setting should be disabled, and the clock frequency of the processor should be set to full speed. Power consumption will then most probably reach the specified maximum, and care must be taken that sufficient cooling capacity is available. However, not all tested processors revealed a distinct effect of the clock frequency on the worst-case latency. It may, therefore, be advisable to determine the worst-case latency under the various settings of the CPU frequency scaling to ensure that this has the expected effect.

5 Sleep states

Another mechanism to reduce energy consumption when computing power is not needed are sleep states. A processor may implement a certain number of sleep states going from snoozing to deep sleep. The deeper the sleep state is, however, the longer the processor may need to wake up and to react to an unpredictable asynchronous event. It is, therefore, important to take care of the sleep state setting when the worst-case latency is crucial.

The sleep states are controlled through BIOS settings and kernel parameters. Unfortunately, there is not a standardized naming convention of BIOS settings that affect sleep states, but every processor and BIOS manufacturer may call the related settings differently. There is a kernel command line parameter that can be used to limit the sleep state to a certain maximum, but this does not necessarily prevent the BIOS from sending the processor to sleep state. If in doubt, the kernel parameter processor.max_cstate can be used. If set to 1, the kernel will not send the processor to any sleep state. Figure 2 exemplifies a long-term latency recording where this parameter was used during certain periods of time. A total of more than 220 latency plots are stacked horizontally with the time running backwards from back to front. It is clearly visible that the worst-case latency dropped from about 200 to about 130 microseconds when this parameter was specified.

![Figure 2: Effect of the kernel command line parameter processor.max_cstate=1 to prevent the processor from entering a sleep state (data obtained on an AMD six-core processor Phenom II X6 1090T)](image)
5.1 Effect of cycle interval on latency

To investigate whether a processor uses sleep states, the modified `cyclictest` utility was used that allows to specify a range of test intervals. A range of 100 microseconds to 1 second was specified, the number of histogram cells was set to 100 which is equivalent to 100 microseconds, and a total of 40,000 cycles per core was preset:

```
cyclictest -m -Sp99 -i100-1000000 -l40000 \ -h100
```

Figure 3 shows the result of such a measurement on an Intel Core i3-2100T processor running at a maximum clock frequency of 2,500 MHz. All power-saving features were enabled at BIOS level. The interval is scaled logarithmically and multiplied by 10; thus, the scale value of 20 is equivalent to $10^2$ microseconds and the scale value of 60 is equivalent to $10^6$ microseconds = 1 second. It can be seen that the worst-case latency increases with the increase of the duration of the preceding measuring interval. This is very probably the result of entering a sleep state when the processor is idle for a certain amount of time.

![Figure 3: Frequency of latency samples with respect to the duration of the preceding interval of quiescence (data obtained with a modified version of the cyclictest utility), Intel Core i3-2100T @2,500 MHz, all power-saving features enabled](image)

When the same command was run on the identical hardware but with BIOS power-saving features disabled, a completely different result was obtained. First, the worst-case latency was lower in general; secondly, there was no increase of the worst-case latency with increasing interval duration (refer to Figure 4).

![Figure 4: Frequency of latency samples with respect to the duration of the preceding interval of quiescence (data obtained with a modified version of the cyclictest utility), Intel Core i3-2100T @2,500 MHz, all power-saving features disabled](image)

5.2 Recommendation

When a minimum worst-case latency must be achieved, the BIOS settings should be investigated for the occurrence of terms such as "power saving", "energy saving", "speed steps", "power now", "green computing" etc. If possible, anything that is suspicious to reduce the computing speed while in idle state should be disabled. In addition, the kernel parameter `processor.max_cstate=1` could be tried.

6 Undisclosed internal mechanisms

In addition to throttling and sleep states, there probably are undisclosed internal mechanisms that reduce the power consumption of a processor and conversely affect the worst-case latency. Not all mechanisms, however, are related to idle state. One of the tested processors, for example, is used in a notebook along with a chip set that is optimized for mobile systems. When the modified version of the `cyclictest` utility was run on this processor, elevated worst-case latency values were obtained, but they were independent from the duration of the test interval (refer to Figure 5).
7 Conclusion

Power-saving features of state-of-the-art microprocessors certainly have a negative impact on the worst-case latency. If a very fast response to external events is crucial, e.g., in the range of single-digit microseconds, power saving should better be disabled whenever possible. If the constraints on the worst-case latency are less tight, power-saving may well coexist with real-time execution, since the determinism of the real-time response is not affected per se. It must, however, be considered that the latency may increase by an order of magnitude.

It is not difficult to predict that battery-driven devices such as smartphones will have an important influence on the development of future processors. And it is well conceivable that the semiconductor industry will develop even more sophisticated mechanisms of power saving that may interfere with real-time execution even more than they already do. It remains to appeal to semiconductor manufacturers to - whatever mechanisms they invent to reduce power consumption - provide a way to switch them off, at least if they adversely affect response time. On the other hand, field-programmable gate arrays have become very powerful in recent years; this makes it possible to create 32-bit software processors. Related softcores are already available and can be used under an Open Source license [3]. These processors may be designed individually in such a way that they provide optimum conditions for operating in a real-time environment.

Most of the findings presented in this article are not specific for PREEMPT_RT mainline real-time Linux but may be applicable to real-time extensions and to other operating systems as well.

Some of the measurements were made on processors that are under test at the quality assurance farm of the Open Source Automation Development Lab (OSADL). In addition to continuous latency recordings and cyclic latency plots on a large number of different systems, selected systems are running under particular and documented power-saving conditions. All data are publicly available on the organization’s Website at osadl.org/QA [4].

References


[2] GIT repository of the RT Tests that includes, among others the cyclicstest utility, maintained by Clark Williams http://git.kernel.org/?p=linux/kernel/git/clrwllms/rt-tests.git;a=summary - may not be accessible


How to cope with the negative impact of a processor's energy-saving features on real-time capabilities?
On integration of open-source tools for system validation, example with the TASTE tool-chain

Julien Delange and Maxime Perrotin
European Space Agency
ESAAESTEC, TEC-SWE
Keplerlaan 1
2200AG Noordwijk, The Netherlands
julien.delange@esa.int, maxime.perrotin@esa.int

Abstract

Design and implementation of safety-critical systems is very difficult because they must ensure continuous correct operational state whereas they are deployed in hostile environments. An error either during design or implementation phases can have significant impacts and consequences. To avoid such issues, failure cases must be clearly identified and handled by software engineers to prevent any propagation from one faulty component to another. For that purpose, good practices and standards are applied during the development process, from the specifications to the implementation.

However, despite all existing efforts, bugs are still introduced. They are introduced at different levels of the development process: either in the specifications (as in the Mars Climate Orbiter mission - failure was due to a mix-up of metric units) or in the implementation (as in the Ariane 5 launch - wrong assumption was made about a data type so that the system generate an overflow).

Over the years, several solutions have been designed to address such issues. However, they rely on different system representations and are applicable at different levels of the design process, so that their use could be difficult and may lead to design inconsistencies. In consequence, we have to avoid these problems and make their use more consistent.

In this paper, we present our tool-chain for system design, validation, implementation and certification. It relies on a modeling notation to capture both software and hardware concerns. The use of a single notation ensures specification consistency and avoiding potential errors when using different language to specify the same system aspect. We detail the support of this process in The Assert Set of Tools for Engineering (TASTE) development tool-chain.

1 Introduction

Safety-critical systems design and implementation is very difficult: they must operate correctly and continuously whereas they are usually deployed in hostile environments. Misconception or errors may have significant consequences and are potentially mission or life critical. To avoid such issues, failure cases must be clearly identified and handled by software engineers to avoid any propagation from one faulty component to another. For that purpose, several guidelines and standards have been designed and are currently used during all system development phases.

Safety-critical systems development is usually splitted in several phases, as illustrated in figure 1:

1. **Validation**: from specifications, engineers/developers check system feasibility and requirements enforcement
2. **Code production**: developers implements the system by translating the specifications into code (Ada/C) that can be compiled.
3. **Certification**: implementation execution is validated against its specifications and/or established standards (DO178B, ECSS, etc).

Even with a such careful process, errors remain either in the specifications (as in the Mars Climate Orbiter mission - failure was due to a mix-up of metric units) or in the implementation (as in the Ariane 5 launch - wrong assumption was made about a data type so that the system generates an overflow)
type so that the system generate an overflow). In particular, errors may be introduced at each step:

1. **Validation**: because it often rely on different system representations, a requirement validated using one notation may not be validated using another one.

2. **Code production**: developers often introduce errors or bugs either by misunderstanding system specifications or just by making syntax or semantic errors.

3. **Certification**: this also relies on a manual process where engineers can make errors by misunderstanding specifications.

To cope with these problems, the development process must be more consistent and automated as much as possible. It would automatically check requirements from specifications and ensure standards compliance enforcement at lower levels. As a result, this would also reduce development cost (system is verified by appropriate tools), ensure the development process reliability and robustness due to the automation of each step.

Next section details in details identified problems and presents our approach to address them.

---

2 Problem & approach

2.1 Problem statement

Despite all existing initiatives, errors/bugs are still introduced during development. To reduce them as much as possible, one has to:

1. **check and validate specifications automatically**

2. **verify implementation correctness regarding specifications**

Different tools already address these issues. However, they are loosely coupled and rely on a different notations that lead to potential semantic issues so that:

1. a requirement R1 validated using the specification language L1 and the validation tool T1 can be not validated when using the specification language L2 and the validation tool T2.

2. even if the user manages to translate system specification from one language to another, the process will require a manual translation which is error-prone: the user can introduce specification errors by himself (syntax error, misunderstanding of system specifications, etc.).

As a result, there is a strong need for a more consistent approach that strengthen system development with:

1. **A single notation** for all system aspects so that we avoid several representations of the same concepts and so, prevent any specification inconsistency.

2. **An automation of development steps** with tools that process specifications and produced development output without human guidance.

2.2 Proposed approach

First, we propose to capture system architecture with its requirements, properties and constraints using a single modeling language. It would specify both software and hardware concerns with a unique notation, avoiding all usual semantic issues.

From this high-level representation (models), development steps (same as in figure 1) are automatically processed by appropriate tools. In particular:

---

FIGURE 1: Generic development process

work-flow
1. **Validation**: tools automatically process models to check system correctness and feasibility so that designers can fix specifications errors before further development efforts.

2. **Code Production**: code generators transform specifications/models into implementation code (such as Ada or C). This generated code is automatically compiled and linked against a Real-Time execution platform that supports system entities (tasks, mutexes, etc.)

3. **Certification**: implementation is either simulated or executed on the target to check its behavior correctness and standards (such as DO178B or ECSS) compliance.

We implement this process in The ASSERT Set of Tools for Engineering [1] (TASTE) by using AADL as a single specification language. We also design tools that process AADL models and support each development step. Next sections details the both language, its tailoring for our needs and the specific tools we develop in that context.

### 3 Overview of TASTE

The Assert Set Of Tools for Engineering [1] (TASTE) is the outcomes of the European ASSERT project [2]. It [1] aims at providing a complete functional tool-chain for safety-critical system development, from the specifications to the certification/validation.

It relies on the Architecture Analysis and Design Language (AADL) [9] to represent both software and hardware concerns, their properties and constraints. First, software aspects are specified by the **Data View** and the **Interface View**, two AADL models that represent system functions (C/Ada code) and the data they share (using ASN.1 [19], a standardized notation for describing data types). Hardware and deployment concerns are described with the **Deployment View**, an AADL model that describes the execution platform (processors, devices, memories, etc.) and its association with system functions.

![TASTE process work-flow](image)

FIGURE 2: TASTE process work-flow

From these models, our tool-chain automates the development, as illustrated in figure 2:

1. **Validation**: it checks specification correctness by processing models and using appropriate tools:
   - (a) Cheddar [7] or MAST [8], two scheduling analysis tools released under free software
   - (b) REAL [15], an AADL [9] query tool integrated in Ocarina [10] that checks specification correctness

2. **Code production**: it transforms AADL models into C code using Ocarina [10], an AADL tool-suite released under GPL licensing terms. In particular, Ocarina is able to automatically generate code that targets real-time embedded platforms (RTEMS [12], VxWorks[13], etc.) and standards (RT-POSIX[14]). The code is then integrated on top of a real-time execution platform: Ocarina [10] actually supports the following free-licensed platforms: RTEMS [12], Xenomai [11] and POK [18].

3. **Certification**: it executes the code either on the target or a simulator (such as QEMU), checks:
   - its performance (using gprof, a performance analysis tools included in the GNU binutils suite [3])
reproduces its behavior (by instrumenting the code and produce a Value Change Dump (VCD) [6]) file to be used with GTKWave [5]).

• produces code coverage reports using the COUVERTURE tool-set [4] (specific free-licensed tools from Adacore that aims at supporting code coverage using a specific tailored version of QEMU [16]).

The use of a single notation (AADL [9]), processed by dedicated tools for each development aspect makes the overall process more consistent. In addition, automation of model processing avoids issues of usual development process and ensures requirements traceability. Finally, while system feasibility and requirements are automatically checked during the development process, these tools also provides metrics (such as code coverage) that can be used for system certification.

Next sections focus on validation and certification functions of our tool-chain:

• Section 4 describes our system validation functions from using AADL specifications.
• Section 5 details the automatic certification process with respect to the implementation.

4 Model Analysis & Validation

The TASTE tool-chain relies on AADL [9] to specify software (Data View and Interface View) and hardware (Deployment View) aspects. AADL is a component-based language to specify hardware and software concerns with their execution constraints. It has all necessary constructs to express safety-critical system concerns and supports mechanisms to address specific modeling needs. They can be written using either a textual or a graphical notation and is supported by a large tool-set, from command-line interface tools (such as Ocarina [10]) to tools with advanced graphical interface (like OSATE [9]).

As this article does not aim at providing a full overview of AADL, readers that would like to learn more about it can refer to the introduction written by its designers [20].

4.1 AADL modeling benefits

By introducing a single specification notation for both software and hardware concerns, we strengthen the overall development process, making it more consistent. Extensions mechanisms allow us to tailor the language to our needs:

• Properties extension mechanism is used to define specific requirements from textual specifications to the AADL model (for example, to model memory concerns such as stack or heap size, etc.).
• Annex languages mechanism is used to associate our in-house AADL validation tool (REAL) to check requirements enforcement. It processes processes models according to its components hierarchy and check for system requirements validation (for example: can a process P1 with 1Mb of RAM contain three threads that require a stack of 800Kb?).

If several modeling languages already exist for the specification of real-time embedded systems, no one provides the ability to capture both hardware and software aspects with such a flexibility. That is why our choice was focused on this language.

4.2 REAL validation tool-set

REAL (Requirements Enforcement Analysis Language) [15] is a language that associates validation theorems to AADL components. A dedicated solver analyzed model components with their theorems and check their enforcement.

To use REAL, users have to:

• Map properties and constraints from the textual specification to the AADL model (for example, execution time for each system function, period/deadline of each task, etc.).
• Design theorem to check requirements feasibility (for example: functions can be executed within task period).

One key aspect is the genericity of this approach: users can keep existing theorems in a library that would be reused for later projects.

Listings 1 and 2 give an example of the definition of a REAL theorem and its application on an AADL model. Listing 1 defines a (incomplete, due to lack of space) model with one main system containing:

• One process component with two tasks (thread components). The first one requires 35Kbytes of memory and the other 57Kbytes;
• One memory component with 40000 bytes.
process implementation p.i

subcomponents

task1 : thread t.i
{ SourceStackSize => 10 Kbytes;
SourceDataSize => 20 Kbytes;
SourceCodeSize => 5Kbytes; }

```
task2 : thread t.i
{ SourceStackSize => 2 Kbytes;
SourceDataSize => 50 Kbytes;
SourceCodeSize => 5Kbytes; }
```
end p.i;

system implementation s.i

subcomponents

```
mem : memory ram.i
{ WordCount => 10000;
WordSize => 4 bytes; }
```
prs : process p.i;
properties

```
ActualMemoryBinding =>
(reference (mem)) applies to prs;
```
end s.i;

Listing 1: AADL model example to be processed by the REAL validator

The REAL theorem (listing 2) checks that for each AADL process component of the model in listing 1, the amount of memory required by its tasks (lines 10 to 12 of listing 2) is less than the size of its associated memory (lines 14 and 15 of listing 2). Regarding the model of listing 1, this theorem is not be validated and validation tool would issue an error.

```
theorem check_memory
foreach prs in process_set do
  t := { x in ThreadSet | is_subcomponent_of (x, prs)};
  m := { x in MemorySet | u is_bound_to (Prs, x)};
  check {
    (sum (property (t, "SourceStackSize")) +
    sum (property (t, "SourceDataSize"))) +
    sum (property (t, "SourceCodeSize"))) <
    (sum (property (m, "word_count")) +
    (property (m, "word_size")));
  }
end check_memory;
```

Listing 2: REAL theorem that checks task memory requirements

4.3 Scheduling validation

Scheduling is a very intensive topic in the context of embedded and real-time systems. Numerous scheduling analysis techniques and methods have been designed over the years trying to evaluate system scheduling feasibility.

TASTE interfaces AADL specifications with two scheduling analysis tools: Cheddar [7] and MAST [8]. Both are available under the GPL license terms. Next sections give an overview of these tools and explain how AADL models are exported to them.

Overview of Cheddar and MAST

Cheddar [7] is a scheduling analysis tool written in Ada that provides command-line as well as graphical interface (shown in figure 3). It validates timing constraints either by simulating system execution or performing feasibility tests. To do so, the user must describe system architecture (processor, task, scheduling policy, etc.). Cheddar supports state-of-the-art scheduling algorithms (RMS, EDF, LLF), as well as standardized algorithms (like the one available in POSIX 1003b). Cheddar analysis also takes inter-tasks dependencies into accounts with an analysis of different sharing methods such as PIP, PCP or IPCP.

![Cheddar scheduling validation](image)

FIGURE 3: Cheddar scheduling validation

As for Cheddar, MAST [8] is a tool (shown in figure 4) that aims at validating scheduling feasibility of a system. It can analyzes system using several algorithms either for task scheduling (RMS, EDF, etc.) or data locking (PIP, PCP, etc.). On the other hand, MAST takes into account distributed systems concerns, which is especially critical for real-time systems. For example, when a task execution is triggered an incoming data from another task, analysis has to take into account scheduling concerns since a delay on the sender side would have an impact. In addition, network-related aspects (such as latency, jitter, etc.) may also impact system execution as a whole. A system specification in MAST takes these
aspects into account, offering a convenient way to analyze distributed systems.

Connect scheduling analysis tools with AADL

To automate scheduling analysis, TASTE transforms system specifications (Interface View and Deployment View) into a new description that can be processed by MAST or Cheddar. First, it translates the AADL models into a Concurrency View: a single AADL model that merges both software and hardware aspects with all execution entities (tasks, shared variables, etc.) with their scheduling constraints (scheduling algorithm of processors, locking policy for shared variables, etc.).

Then, an appropriate code generator (Ocarina [10]) transforms this concurrency view into a new representation suitable for Cheddar or MAST. In fact, it consists in translating AADL language constructs into an XML representation that can be processed either by Cheddar or MAST. As a result, this export function of our tool-chain bring the ability to automate scheduling analysis with both tools from the same specification (AADL models).

5 Implementation analysis

TASTE automatically creates system implementation from its Interface View, Deployment View and Data View (AADL models) by generating code that targets Real-Time operating systems. However, even if this automatic process offers many benefits (error avoidance, requirements enforcement & traceability, etc.), system implementation still has to be validated and also met certification requirements.

5.1 Performance analysis

Once system implementation is generated, developers can deploy it on the execution target. Then, appropriate tools trace/monitor system behavior to evaluate its performance. For that purpose, several tools already exist and are released either under proprietary or free-software license.

To assess generated application performance, TASTE uses gprof, an execution profiling program available in the binutils [3] tool-set. Its main advantage is its integration within the GNU compilation tool-chain: just by adding a flag in the compilation options enable application profiling that can be later processed by analysis tools.
TASTE provides its own interface method with gprof, as illustrated in figure 6. It parses profiling results by its own and produces an execution report with the execution time and the number of execution for each function. By using this report, engineers check execution traces compliance with system requirements.

5.2 Specifications compliance enforcement

Execution profiling provides metrics and data that could detect some erroneous execution case (a function called too many times, a call that would not happen, etc.), but may be not sufficient to check implementation correctness. In particular, implementation validation requires to check implementation consistency with the specifications (AADL models). This consist in monitoring system events, and check their compliance with the model. For that purpose, TASTE provides functions to monitor system events at run-time and create appropriate metrics that can be compared with its specifications. To do so, it instruments generated application with profiling instructions that produces VCD [6] files at run-time (example of events reported is available in figure 7) with the following metrics:

- Task activation time
- Data sent/received through tasks port
- Shared data usage (semaphore/mutex acquisition and release)

Once produced, programs such as GTKWave [5] as used to depict system events with a graphical interface and provide the ability to analyze system behavior. It offers the ability to check run-time behavior consistency with system specifications (for example, the task activation time is correct regarding specified period and deadline).

5.3 Code coverage analysis

Standards such as DO178B [21] (for avionics systems) or ECSS [22] (for aerospace applications) requires that safety-critical systems enforces a predefined code coverage, depending on their criticality level.

To do so, different methods are commonly used, but most of the time, they require a manual instrumentation or inspection of application code. Code instrumentation is intrusive: the code under inspection is the one that would be deployed and so, validation results may not be relevant. In addition, a manual inspection is still error-prone, due to the human-factor errors.

![ FIGURE 6: Gprof interface of TASTE](image)

TASTE provides its own interface method with gprof, as illustrated in figure 6. It parses profiling results by its own and produces an execution report with the execution time and the number of execution for each function. By using this report, engineers check execution traces compliance with system requirements.

![ FIGURE 7: System behavior description with GTKWave](image)

5.3 Code coverage analysis

Standards such as DO178B [21] (for avionics systems) or ECSS [22] (for aerospace applications) requires that safety-critical systems enforces a predefined code coverage, depending on their criticality level.

To do so, different methods are commonly used, but most of the time, they require a manual instrumentation or inspection of application code. Code instrumentation is intrusive: the code under inspection is not the one that would be deployed and so, validation results may not be relevant. In addition, a manual inspection is still error-prone, due to the human-factor errors.

![ FIGURE 8: Work-flow of the couverture tool-set](image)

To cope with these issues and provide an accurate coverage analysis, TASTE relies on the COUVERTURE tool-set [4], a code coverage analyzer released under free-software license terms. It relies on two main tools that produce coverage reports (as shown in figure 8):

1. A tailored version of QEMU [16] traces all executed instructions when executing the system.
2. An analysis tool, xcov, compares executed instructions with the program under execution and produces a coverage analysis report.

By tracking executed instructions and establishing a mapping with the source file, xcov produces a complete coverage report, as shown in figure 9.
details the execution of each line of code so that developers are able to assess if some block could be removed or not.

However, to evaluate system implementation, this coverage functionality would be integrated with a test framework that would execute generated applications with different input values that are representative of a real environment. This would provide a better assessment of system quality, force each condition/decision of the code to be executed and lead to a better coverage analysis.

FIGURE 9: Example of a coverage report produced by xcov

One particular interest of the COUVERTURE tool-set is its non-intrusive characteristic: coverage analysis is representative of the quality of applications that are finally deployed. In addition, COUVERTURE supports several coverage methods that are required by certification standards (Statement Coverage - SC, Decision Coverage - DC or Modified Condition Decision Coverage - MCDC). Using these different methods, we could evaluate and potentially certify generated systems at different levels.

6 Case study

The following sections illustrate the use of our TASTE tool-chain through a case-study that deploys several functions into a heterogeneous and distributed architecture.

6.1 Overview

This case study consists in simulating a temperature sensor with a basic forecast management system. It is composed of three functions:

1. A sensor for temperature acquisition.
2. A filter for bad data detection.
3. An average computer that receives each new temperature value from the filter and prints the average temperature.

Each function is deployed on top a Real-Time Operating System, executed on a single processor:

- The sensor function is deployed on a LEON2 processor with RTEMS.
- The filter function is executed on an Intel i386 processor with a Linux operating system.
- The average function is deployed on a LEON2 processor with RTEMS.

Finally, to enable the communication between functions sensor/filter and and filter/average, the processors are connected using a SpaceWire bus, as shown in figure 11.
it filters the data and send it when it is considered as valid. For the needs of this simulation, 50% of received data are considered as correct so that this function sends data to the average function every two seconds.

- The average function is also sporadic and activated when receiving incoming data from the filter. As the filter function sends data every two seconds, this function execution follows this period.

**FIGURE 11:** Deployment view of the system

Then, once these functional aspects have been described, their allocation on the platform has to be specified using the Deployment View. It defines the hardware to be used and its association with the previously defined functions of the Interface View. The deployment view of the case study (illustrated in figure 11) is composed of three processors (two LEON using RTEMS - acq_board and avg_board and one intel i386 that uses Linux - filter_board) interconnected using SpaceWire links.

**FIGURE 12:** Scheduling analysis result for the node acquisition

Finally, from these both models, we can validate some of its aspects prior to implementation efforts. In this case-study, we run a schedulability feasibility using Cheddar [7], as illustrated in figure 11. The scheduling feasibility test is based on simulation and is performed for each processor of the system (figure 11 illustrates the result for the acq_board).

6.3 Implementation validation

Then, TASTE processes system specifications (Interface view and Deployment view, to generate binaries that would be executed on the target. Then, it analyzes the implementation and verifies its compliance with the specifications in order to check certification requirements enforcement.

First of all, system profiling is performed using gprof, as illustrated in figure 13. Profiling report shows how many time each function has been executed. In the following example, the results refer to the execution of the first ten seconds of the filter function. We can see that the __po_hi_delay_until() function (called by each function at the end of each cycle) has been invoked almost 30 times, which can seem to be inconsistent: this function is triggered each second so that function would be activated at most 10 times. However, the node that hosts it communicates using two SpaceWire bus and each one is using a task that polls the bus for incoming data periodically each second. Consequently, if the functional aspects generates only one task in the system, the deployment concerns (device drivers) add additional resources that have an impact on system execution.

**FIGURE 13:** Profiling of the filter node using gprof

FLOSS in Safety Critical Systems

245
Then, as detailed in section 5.1, TASTE analyzes system implementation to check system behavior compliance with its specifications. Figures 14 and 15 report run-time events that occur when executing the implementation of the node filter_board. Figure 14 reports the events at a coarse grain: we can see that system activity happens on a periodic basis, each second. Then, figure 15 details the events at a finer grain, each second:

1. The poller task of the SpaceWire driver (task_0 on figure 15) is activated. It receives incoming data from the acq_board (that acquires temperature from the sensors).
2. When receiving data, poller task from the SpaceWire driver (task_0 on fig 15) transfers data to system port (port_2_0 on fig 15).
3. New data instance triggers the execution of the filter function and its associated task (task_2 on figure 15) which retrieves the data (so that the size of the port port_2_0 fallbacks to 0), executes its code and waits for new data.

We can see that these events are consistent with system specifications: the SpaceWire driver receives data each second and triggers the execution of the sporadic function filter. However, we didn’t detail the execution of the task task_1 which seems always active. In fact, this task corresponds to the poller function connected to the other SpaceWire bus. As it never receives data (this node only sends data through this bus), the associated task is always waiting for incoming data and never go to the sleep mode.

Finally, our tool-chain produces a code coverage analysis report of the generated application. As explained in section 5.3, safety-critical standards (such as DO178B [21] or ECSS [22]) require that applications enforce quality criteria such as a predefined code coverage value. To do so, TASTE automatically produces coverage report using the COUVERTURE [4] tool-set, as shown in figure 16.

Reports can be produced in different formats (text, HTML, etc.). They detail, for each file and function, the coverage information, so that engineers assess system quality based on execution metrics. In our case-study, most functions of the RTEMS executive are not used so that it significantly decreases the code coverage level of produced applications.

7 Conclusions & Perspectives

This article gives an overview of open-source tools that provide help and guidance for safety-critical systems design. They are used as early as possible to support each step of the development process. Such tools are usually loosely-coupled and require manual efforts to be tailored to the development process of each system. To cope with these issues, TASTE makes their use more consistent by linking them with a single specification notation.

For that purpose, AADL models describes system architecture with its execution constraints using. Then, tools translates this specification notation to be processed by validation programs that check architecture correctness and requirements enforcement. This process automates the process, avoiding issues of usual development methods.
Use of such a tool-chain strengthens the development process and makes it more robust and reliable. Moreover, as potential errors are discovered early in the development process and integration issues would likely be reduced, development cost are expected to decrease significantly.

Further work would cover other aspects of safety-critical systems development. In particular, our tool-chain could also support additional guidance for safety-critical standards (such as DO178 or ECSS) enforcement by providing documentation generation facilities or additional implementation code validation (coding rules to be checked, etc.).

7.1 Perspectives

Automation of AADL models production from usual text-based specifications or linking these two notation would be particularly useful and integrates our tool-chain with traditional design methods. Such a translation process will require that:

- all system entity and its associated requirement from the initial specifications are correctly translated into AADL components
- there is no specification inconsistency (due to semantic issues, mapping error, etc.) between the AADL model and the initial specifications.

Tools that address these issues are currently being developed. For example, the TOPCASED [23] requirements importer tool provides the ability to connect a requirement from a textual document (with an extension such as .odt, .pdf, .txt, etc.) to a model (potentially AADL). This work is emerging and available tools are still considered as experimental, but this topic is a particular interest and would be a major interest to trace a requirement from its description (in the textual document) to the implementation (the code).

Then, another idea is to strengthen our tool-chain by improving its functions. In fact, some TASTE functions are limited to several architectures or platforms (for example, system analysis that produces VCD files is limited to Linux platforms). An important improvement would consist in supporting all potential deployment platforms.

References

[1] The ASSERT Set of Tools for Engineering
http://www.assert-project.net/taste

[2] The ASSERT project
http://www.assert-project.net

[3] GNU binutils
http://www.gnu.org/software/binutils/

[4] The Couverture project

[5] GTKWave
http://gtkwave.sourceforge.net/

[6] Value Change Dump file format

[7] Cheddar scheduling analyzer

[8] MAST scheduling analyzer
http://mast.unican.es/

http://www.aadl.info

[10] Ocarina AADL Toolsuite
http://ocarina.enst.fr


[14] POSIX 1003.1b

[15] Olivier Gilles and Jérôme Hugues - Validating requirements at model-level in Ingénierie Dirige par les modèles (IDM08)


[18] Partitioned Operating Kernel - POK
http://pok.safety-critical.net

[19] Gerald W. Neufeld and Son Vuong - Overview of ASN1 in NetISDN, 1992


[22] ECSS-E-40, Space Engineering Software

On integration of open-source tools for system validation, example with the TASTE tool-chain
Safety logic on top of complex hardware software systems utilizing
dynamic data types.

Nicholas Mc Guire
Distributed and Embedded Systems Lab
SISE, Lanzhou University
mcguire@lzu.edu.cn

Abstract

Utilizing computers for safety critical systems, notably contemporary super scalar multi-cores, let alone NUMA systems running general purpose operating systems like GNU/Linux, is quite contended in the safety community - their hopes still rest on determinism and KISS. While keeping things simple in the safety related components is undoubtedly preferred, it is questionable if keeping the hardware model simple is realistic - notable with the divergence of reality from model with respect to determinism already being dramatic for widely used general-purpose single-core CPUs. Further actually deterministically covering the impact of all complex software components is not doable with an economically tolerable effort (if it is technically doable is a different issue).

The consequence of this belief in determinism, is an, in our opinion useless, fight against complexity and non-determinism - two inherent properties of modern hardware/software systems. Quite to the contrary, we propose to utilize the properties of complex systems to enhance safety related systems. This seemingly paradox approach can be seen as an attempt to take the bull by the horns as it seems inevitable that the time of simple CPUs and black-box proprietary operating systems, that continue to entertain the illusion of determinism, is coming to and end.

Safety mechanisms, drawing enhancements from underlying complexity, we see as potentially suitable for building safety related systems are:

- computation: Inherent diversity
- data: mapping value domain to complex data representations
- time: loos coupling: inherent randomness

and we are quite sure that this little list is incomplete at this point.

In this article we will describe an attempt at the second category called dynamic data types, which essentially combine the value domain with the temporal properties of data to map data to a value in the frequency domain rather than to a value in the time-domain. We outline the concept of dynamic data types and a rational for why it seems a promising approach for covering of particular fault classes. Finally we describe how building simple logic utilizing dynamic data types on complex systems can yield a safe system never the less and thus allow to co-locate safety related logic with non-safety related general purpose applications and services on a single contemporary system.

Keywords: Safety logic, complex systems, dynamic data types

1 Introduction

Digital systems have a number of well known advantages, notably with respect to signal rectification and complexity of signal interaction without degeneration - simply speaking the absence of noise. Though this absence of noise has great technical advantages, it has some interesting side-effects with respect to safety properties. During failure mode and effect analysis for digital systems one regularly stumbles across a number of fault classes seemingly inherent to digital systems:
• permanent bit faults (a memory cell, a shorted relay, etc)
• transient bit faults (the infamous cosmic ray - or more earthly EMV issues)
• systematic operational faults (i.e. the FOOF bug in Pentium I)
• transient operational faults (i.e. critically low voltage)
• temporal faults (i.e. clock drifts or clock updates)

This is a bit course grained but serves well for the discussion here. It also should be noted that failures are only one possible cause of hazards, so this is in principle incomplete [5].

All of these fault classes are then mitigated by different technologies (see [1] IEC 61508 part 7 for a overview of available technologies) , starting from redundancies and different levels, by adding diversity in hardware or software, and by introducing protocols that mitigate against consequences of these faults (i.e. using sequence numbers, CRC, etc.[4]). All of this surrounding safety related mitigation’s of the respective fault classes may well be technically suitable, but the question to ask first is - why do these faults exist in the first place - and if they are inherent to digital systems can they not be mitigated at a generic level rather than growing the complexity at the application level ?

There have been some indirect efforts to mitigate these issues at a generic level, think of the many IEC 61131 [3] runtime systems allowing to focus again on the relatively simple logic and handling the safety related issues below-the-hood. The assumption being that for a simple set of operations a complete list of potential faults can be established and mitigated. The methods though used for this mitigation again are relatively specific and in general limited to ladder-logic or boolean-logic constructs - so the question remains - what is the root cause and can it be mitigated at a principle level?

2 Naive case study

This case study might seem quite trivial but from a safety perspective we believe it demonstrates the potential advantage of the concept introduced in this paper - dynamic data types.

if this is implemented with common means then we have a signal level (voltage or digital makes no difference here) indicating the switch position of up/down, we have an actuator that operates on such a signal and we have the indicator lights that will provide operator feedback on current actions.

**possible single faults (simple model)**

• up stuck:
  actuator “sees” up pushed and lifts the door, it is fed back to the operator via indicator light but the operator might not be able to close it, and the actuator can’t actually determine that the input is unintended or invalid. The hazard is probably limited as the indicator is correct and humans thus can respond according to preset procedures to achieve a “safe state” - long term consequences of ”up” being persistent are obviously only meaningfully interpreted in a particular context.

• down stuck:
  actuator “sees” down permanent and closes the door - feedback to the operator indicates closed door - safe state assumed - thus no hazard, but availability is impacted.

• actuator stuck:
  the actuator does not respond to input, it stays in what ever state it happens to be - this may be visible in the indicator light but the operator would not see that the actuator is damaged in the signal - an additional diagnostic input would be needed (i.e. indicator for the actuator operation status it self). Depending on the actuator stuck position this may or may not be safe - again context would be needed.

• indicator stuck green: - this would not allow the operator to detect a potential hazardous situation and exposure to a hazard could occur based on procedures (green indicating entering permitted). The problem would not be detected until an operator attempts to change state by requesting “up” and not getting any response (indicator light changing to red) - but it would rely on human observation and even if detected the diagnostics would require additional means.

• indicator stuck red:
  impacts availability primarily.

Note that we are not considering any temporal issues here - there are of course a whole set of temporal failure modes as well, nor are we concerned with
the (and often dominant) non-technical failures like management and safety culture failures.

2.1 Adding diagnostics

As noted above there are a number of faults that could be diagnosed if additional information were available from additional sensors - but the obvious disadvantage of this is that additional sensors mean reduced availability on the one hand and the problem of accumulated faults for any indicators of "rare-events" - notably the later is problematic from a safety perspective. A further issue is simply that the system intended to be kept simple starts becoming more complex than would be needed for the pure logic.

The above example exhibits two main problems:

- lack of indication of "invalid" state for all components
- inability to identify the cause without additional sensors

If we look at the potentially hazardous situations then these are associated with the problem of single points of failure as well as the inability to signal "invalid state".

Before considering methods to implement redundancy, signal-diversity, safety communication, etc. we would like to look at the root causes and then derive requirements that could potentially eliminate them all together.

3 Root cause

Paradoxically, from a safety perspective, one of the problems of digital systems is the absence of noise - strictly speaking it is not absence, it just is a "rare" phenomena and thus perceived as being absent - we don’t notice the rare event of bit-flips on the wire, either it is covered by detection and retransmission (i.e. ECC RAM, or noise cancellation on ethernet wires) or it leads to a failure that we then "fix" by rebooting our PC...

3.1 Value domain

The assumed absence of noise on the signals in digital systems has the irritating consequence that the rare event of bit-level alteration (SEUs) does not lead to a violation of value domain constraints. That is to say if a register contains an integer and a bit is flipped it still contains an integer and is within range - the bit-flip is not noticeable in the data-type. The reason for this is that the value domain is dense (even though it is discrete - dense in the sense that there are no "holes" in the representation) - conversely analog signals had to be constructed with a granularity to ensure detectability of divergence, so the analog signal spectrum, effectively usable, could not be dense.

**SEU**

Single event upsets are random alterations of some state of the system that can’t be predicted, they only can be covered by some form of fault-tolerance or fault detection/reaction - they are in principle not testable as they are not visible until they actually occurred. In case where the affected resource may stay in the altered state, the alteration may be detectable if additional information about the expected state is available.

```
logic high .----- perceived value
   |
logic low------- -- intended value
   SEU
```

**Omission**

Omission faults are due to the inavailability of some resource. They are in principle detectable, though the effort for this may be high - note that omission faults may also have systematic characteristics (i.e. wire cross-talk).

```
logic high . - - - intended value
   |
logic low ------------ perceived value
bit failure output is a legal value
wire breakage
```

So the SEU or omission occurrence is not visible in the output (think of this as a register or memory cell in which 0 was originally stored - due to wire cross-talk it is flipped to 1 and then read as 1), the bit failure on the other hand could also be due to accumulated faults or physical failures of memory/register/wire but stay unnoticed until activated by accessing. Testing for this fault is also not simple as it need not have caused a permanent damage, so writing to the SEU affected location "clears" the fault aside from the problem of some paths not being easily testable until you need them (think of fault
handling or emergency shutdown procedures). With other words the problem is that this type of fault is neither detectable in the value domain, nor is it mitigated in the data type.

This is in fact inherently a property of digital systems, and a number of mitigation’s have been proposed. Some of these mitigation’s are:

- inverse replicated logic (2-out-of-2)
- diverse replicated logic (2-out-of-2, N-out-of-M for availability reasons)
- periodic testing and statistical SEU occurrence models ”protecting” the intermediate intervals.
- safe data objects, protected by additional representations (i.e. CRCs)

All of these protections have one thing in common, they tackle the problem at the symptomatic level, and all of them have the potential for common-cause faults, though the probabilities can be argued to be sufficiently low (if reality correlates with these arguments is not the topic here).

A further serious problem, from a safety perspective, of digital systems is that the absence of an active signal, due to failure, may also be a legal value - i.e. a solid 0.

The root cause of the problem though lies in the data type it self not becoming invalid by the alteration, and thus on-the-fly modification, and even permanent modification (i.e. a stuck relay) are not visible/detectable in the data-type it self.

3.2 Time domain

A similar situation is given for the time domain. Basically the primitive data types don’t have any encoding of the time-axis. Even if we pair values with some for of timestamps (also just a value representing some point in time) we don’t substantially change the issue, we just have added a level of complexity for a correlated fault to occur (that is change of the value and synchronous change of the time-stamp to fit the expected value - think of a zero-crossing detection for sinusoidal signal used frequency monitoring on power-lines, the failure now would require a value and time-stamp fault to lead to a false positive, so the probability is reduced but it is not eliminated).

Clock fault

logic low  .-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.---

<table>
<thead>
<tr>
<th>logic low</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1'</td>
<td></td>
</tr>
</tbody>
</table>

| clock fault | output is late but not detectable by the externally perceived value. |

4 dynamic data types

In this section we introduce a novel concept (at least in the context of safety related logic) of encapsulating the general signal in a way that does not exhibit the above mentioned limitations, and would not require mitigation by additional means but rather enforces inherent mitigation of faults. At this point we are not (yet) claiming that the fault coverage is actually complete - but we do think that the majority of relevant faults (notably single fault hypothesis) are covered without impacting availability by false-negative.

The system model behind dynamic data types roughly is to view the input signals as streaming through the system and accumulating changes based on the operations done on them. There is no reliance on the correctness of operation, there only is a reliance of the effects of operation being unique and visible in the signal. This is in some ways similar to coded mono-processors [7] that focus on the operations rather than on the data, and ensure that operations carry such a unique property. The basic operations that are currently being studied are:

- Signal expansion: i.e. adding of signals - extending the signals content to contain the difference and sum of the inputs.
- Signal reduction: i.e. filtering of signals - reducing the spectrum it represents.
- Delaying the signal: i.e. phase shifting - altering the temporal correlation to other signals in the system.

4.1 Signal Requirements

The requirements listed here are only from a safety perspective and not from a functional perspective. further more they are most likely not yet complete as this project is in an early stage.

- all valid states must be active states - inactivity of any component may not lead to a legal value
• all elements may only react to valid states (implying active states)
• SEUs shall not impact the value domain (general resilience against SEUs)
• signals must encapsulate temporal properties along with value properties
• Operations on signals may not include a single-point-of-failure (i.e. comparison to fixed value or fixed offset addition or filter parameters of which modification of only one leads to a false-positive filter)

To satisfy these - admittedly very crude - requirements, we propose mapping logical values to discreet frequencies, with a tolerated range. In this example we mapped:

<table>
<thead>
<tr>
<th>Logic Dynamic representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
</tbody>
</table>

Each value needs a specific tolerance - so the value domain which is mapped to the frequency domain is sparse, and thus tolerant against a certain deviation (i.e. transient SEUs). Further signal interaction is by multiplication, thus any signal omission would lead to multiplication with a 0 value and thus no valid output.

![Figure 1: response to input omission failure](image)

Stuck at failures are a bit more critical, selecting proper frequencies in the generated intermediate spectrum in combination with band-pass filtering (in this case butterworth filters) allows covering both omission and stuck-at faults.
4.2 Core concepts of dynamic data types

From a safety perspective dynamic data types concept can be summarized as

- variable values are bound to a frequency spectrum and so conserve temporal information of the signal
- only active states are legal (recognizable) states - that is signal omission or stuck-at does not yield a valid signal
- combinations of signals lead to valid signals or no signal
- Complex logic can be described by signal composition

- compromising dynamic data types would require arbitrarily complex correlated alterations of data/code to yield a valid signal
- increased system complexity reduces the probability of a false-positive signal generation

Essentially the goal of dynamic data types is to “scale” with system complexity growth which we consider inevitable.

4.3 Hardware elements

The concept of dynamic data-types does not only allow implementing computer based logic on top of complex hardware/software systems safety but also would lend it self to low-complexity system that can be fully integrated with computer based systems without reduction of generality.
FIGURE 3: Realization of logic element as low-complexity hardware

Also as the whole point of dynamic data types is to encapsulate not only state information but also status and temporal information in the data representation this only makes sense if the end-elements, the actuators in the system, also can utilize these inherent safety properties.

A basic model of an end-element for the use in dynamic data types, is outlined here. The basic concept is to process the bandpass signal as separate half-waves that provide a diverse representation of the signal - thus any stuck at at best could trigger one side but not both - finally utilizing both half-waves to generate the output action. Naturally such an end-element is quite problem specific thus we can’t generalize this, but this scheme does show how the transition from the digital logic representation/processing to final elements can be done retaining the technical safety properties of the hardware/software system.

FIGURE 4: Realization of safe end-element

While this is still active research, this preliminary hardware implementation does show the generality of the concept allowing to establish a uniform safe signaling model within a safety related system.

4.4 Composable logic

A further issue for any control system is compositability. While we are restricting the model here to boolean logic at present extensions do seem possible. Never the less the first step is to demonstrate compositability of basic logic elements.

Even though the presented example is not a wildly impressive level of complexity - many times one can find relatively simple safety logic involved in monitoring operations of complex systems. Basically the majority of the control is not safety related - think of this as a CNC machine where the only real safety related issue is if the protection doors are closed during operations or in case of operation with open doors (for positioning and maintenance) the movement of components must be limited to very low speeds - the essence of this example is that the actual safety conditions are relatively simple and it is typically resolved by monitoring the system and intervening in case of constraint violation rather than relying on the entire control software to be safety related and "bug free". This relatively simple logic components (i.e. doorALock & doorBLock) can now be monitored by a dedicated safety re-
lated system - hat is some MCU or could be run on
the available computing capacity provided adequate
isolation could be enforced as stated in IEC 61508-
3 Clause 7.4.2.8/7.4.2.9 [2]. To provide such simple
logic one can use dynamic data types that allow cre-
cating reliable logic from signal summations followed
by digital filtering to extract the intended logic oper-
ation. This is achieved by multiplying the inputs and
the resulting frequency summation/difference again
by a constant helper frequency.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>A</th>
<th>B</th>
<th>A*B</th>
<th>H1</th>
<th>Spectrum of A<em>B</em>H1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>0,12</td>
<td>27</td>
<td>, ,15 , ,27 , ,39</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>6,18</td>
<td>27</td>
<td>,9 , ,21 , ,33 , ,45</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>6</td>
<td>6,18</td>
<td>27</td>
<td>,9 , ,21 , ,33 , ,45</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12</td>
<td>0,24</td>
<td>27</td>
<td>3 , , , ,27 , , , ,51</td>
</tr>
<tr>
<td>XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The output frequency spectrum now allows to
use a bandpass to extract A XOR B from the above
spectrum. By changing the input H1 (a helper fre-
cuency) the frequency spectrum generated by multi-
plying the two inputs with the helper frequency al-
lo ws extraction of A AND B. By following the input
multiplication stage by a digital filter (in our case we
use a 4th or 5th order FIR filter) one then can ex-
tact the desired compositional logic term from the
signal.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>A</th>
<th>B</th>
<th>A*B</th>
<th>H1</th>
<th>Spectrum of A<em>B</em>H1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>0,12</td>
<td>9</td>
<td>3 , ,9 , ,21 , ,</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>6,18</td>
<td>9</td>
<td>3 , 9,15 , ,27 ,</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>6</td>
<td>6,18</td>
<td>9</td>
<td>3 , 9,15 , ,27 ,</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>12</td>
<td>0,24</td>
<td>9</td>
<td>9,15 , , , ,33</td>
</tr>
<tr>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While this may seem quite a complex way of
doing this, it is precisely the complexity introduced
that is the protective mechanism against systematic
and stochastic faults in the system as the signals logic
value is encoded in the frequency of the signal and
thus to transform a signal into a different logical sig-
nal (i.e. a signal that should be N Hz representing
on to a frequency of N Hz representing off) would
require a highly complex sequence of synchronous
faults to appear - all faults that appear randomly will
either be in the tolerance of the signaling system (fil-
ter bandpass tolerances) or will cause the system to
enter an identifiable invalid state and lead to a safety
reaction. For systematic faults it may be a bit more
complicated to argue such an approach but equally
the impact of any systematic fault at the lower level
(i.e. operating system, system libraries) would have
to generate a very complex and synchronous response
and not merely a local fault like the infamous FOOF
bug [6].

The claim here thus is that such relatively low-
complexity safety logic elements can be run on a
unsafe OS while retaining the safety properties of
the logic. It should be stressed though that we are
not claiming any mitigation of systematic faults in
the implementation of the safety logic itself - rather
only mitigation of systematic faults in the underly-
ging generic components is claimed. With the overall
complexity of the safety logic and the signal process-
ing beneath it being relatively low we think that it
is an absolutely realistic target to implement such a
logic according to suitable standard procedures and
under control of adequate safety management to en-
sure with adequate probability that the residual fail-
ure rate of the safety logic is sufficiently low. As an
e xample the DDT logic presented here takes two in-
puts A and B and a control input H and provides an
output of A o B and NOT A o B, with the operation
being AND or XOR depending on the setting of H.

Note that the frequencies selected here during
simulation, technically are not sensible, but the prin-
ciples don’t change when transformed to other ranges
of the spectrum - any real-life system would be op-
erating in the kHz range.

FIGURE 5: composable logic element
structure
Inspecting the generated frequency spectrum one can extract further logic operations on the inputs from the output spectrum through adjusted filters as well as additional logic inputs. At this point we neither know the scalability of this composition nor do we know its limits - but the current state of experiments is sufficient to already produce some basic boolean safety logics that could be practicable for safety related applications running on complex (non-safety) hardware software platforms.

4.5 Masking through complexity

As noted this is in the context of efforts to utilize complexity of underlying hardware/software systems to enhance safety rather than fighting them. With the the mapping presented in the last section we obtain a process to construct outputs that is not susceptible to faults in the underlying software/hardware systems because any false-positive requires an arbitrarily long correlated sequence of faults to lead to the false positive. Any fault in the underlying system can impact the processing of dynamic data types at arbitrary points, but as there is no single point of failure in the transformation of the input stream to the output stream, no such single fault could impact the logical correctness - it can of course diminish availability - which is not of concern here. Regarding multiple faults, there is of course the possibility of accumulated dormant faults emerging and these then striking at some mode change - i.e. in the above example when switching the mode-input from AND logic to XOR logic - but any such fault would have to again generate a complex sequence of values to lead to a false positive output which is not impossible but unlikely - how unlikely it is is still under active investigation.

As the data stream though always depends on multiple manipulations (one could view dynamic data types as a N-value diverse data representation) SEUs are fully mask (both in signals as well as parameters for i.e. the filters) and at the same time would be detectable by independent signal monitoring. In many ways what is being proposed here is the re-invention of quite simplistic analogue technol-
ogy in a digital world - but with the advantage of retaining the signal transmission properties and the ability to manipulate the signals in software.

5 Conclusion

Essentially we believe that safety needs methods to build on complexity - ideally appreciate complexity of underlying systems, without compromising safety principles. We see little point in a continuous fight against ever increasing complexity of system software and hardware - the question of "should it be done in software" is legitimate in many cases, but there is also little point in denying the trend towards complexer software/hardware systems and aggregation of functions of different safety levels in systems. If the safety community does not find answers to the complexity growth that fit current technologies and goes on recommending simple systems with simple or no operating system, this will eventually backfire as we will see the expertise and experience as well as field data of these systems fade into negligence - resulting in degraded safety in the long run.

We are not claiming that dynamic data types are THE answer to the problem, but we do believe that they are an example of the potential that lies in the paradigm change when switching to "capitalize on complexity" rather than "fight complexity".

As Leveson notes that software can affect system safety in two principle ways:

- it can exhibit behavior in terms of output value and timing that contributes to the system reaching a hazardous state
- it can fail to recognize and handle hardware failures that it is required to control or respond to in some way.

As safety is a system property dynamic data types can’t in principle guarantee software safety - but we believe dynamic data-types can give reliable mitigation’s to both of the above mentioned potential of software to impact system safety. Thus while the implementation of dynamic data types (the software components as well as the frequency/spectrum-gap selection) are directly safety critical, such a system can be allowed to run on a un-safe operating system/hardware due to the inherent detectability of induced faults. Thus the required independence from the co-located software components can potentially be provided.

While we see dynamic data types as a potential solution to some of the typical safety logic needs, we are well aware of this not yet being a mature concept - rather we hope that we will find opportunities to study this approach in more detail. Independent of the suitability of the dynamic data type concept though, we believe that it is time to start thinking about how to utilize complexity and inherent non-determinism for safety related systems rather than continuing to rely on an abstract model of computing that is crumbling - if not vanishing.

References


[3] IEC 61131,


[6] Pentium F00F bug ,Wikipedia, May 2010 HTTP://EN.WIKIPEDIA.ORG/WIKI/0XF00FC7C8A

POK, an ARINC653-compliant operating system released under the BSD license

Julien Delange
European Space Agency
Keplerlaan 1, 2201AG Noordwijk, The Netherlands
julien.delange@esa.int

Laurent Lec
MakeMeReach
23 rue de Cléry, 75002 Paris, France
lec.laurent@gmail.com

Abstract

High-integrity systems must be designed to ensure reliability and robustness properties. They must operate continuously, even when deployed in hostile environment and exposed to hazards and threats. To avoid any potential issue during execution, they are developed with specific attention. For that purpose, specific standards define methods and rules to be checked during the development process. Dedicated execution platforms must also be used to reduce potential errors. For example, in the avionics domain, the DO178-B standard defines the quality criteria (in terms of performance, code coverage, etc.) to be met according to the software assurance level. ARINC653 specifies services for the design of safe systems of avionics systems by using partitioning mechanisms.

However, despite those specific methods and tools, errors are still introduced in high-integrity systems implementation. In fact, their complexity due to the large number of collocated functions complicates their analysis, design or even configuration & deployment. In addition, an error may lead to a safety or security threats, which is especially critical for such systems.

In addition, existing tools and software are released under either commercial or proprietary terms. This does not ease identification and fix of potential security/safety issues while also reducing the potential users audience.

In this paper, we present POK, a kernel released under the BSD license that supports software isolation with time & space partitioning for high-integrity systems implementation. Its configuration is automatically generated from system specifications to avoid potential error related to traditional code production processes. System specifications, written using AADL models are also analyzed to detect any design error prior implementation efforts.

1 Introduction

Safety-critical systems are used in domains (such as military, medicine, etc.) where security and safety are a special interest. They have strong requirements in terms of time (enforcement of strong deadlines) and resources consumption (low memory footprint, no dead code, etc.). These systems must ensure a continuous operational state so that each potential failure must be detected before generating any defective behavior. However, they often operate in hostile environments so that they must be especially tested, validated to ensure absence of potential error.

Last years, processing power of such real-time embedded systems increased significantly so that many functions previously implemented using dedicated hardware are now provided by software, which facilitates updates and reduces development costs. In addition, this additional processing power gives the ability to collocate several functions on a same
computation node, reducing the hardware to deploy and thus, the overall system complexity.

However, by collocating several functions on the same processor brings new problems. In particular, this integration must ensure that each function will be executed as it was deployed on a single processor. In other words, the execution run-time must provide an environment similar to the one provided by a single processor. In addition, impacts between collocated functions must be analyzed, especially for safety-critical systems where integrated functions may be classified at different security/safety levels.

Next section details the problem and our proposed approach. It also presents other work related to this topic.

2 Problems & Approach

The following paragraphs details identified problems related to high-integrity systems design and development with the increasing number of functions and their integration on the same processor. Then, it details our approach and each of its functionality.

2.1 Problems

High-integrity systems host an increasing number of functions. In addition, due to new hardware and performance improvements, the trend is to exploit additional computing capacity and collocate more software components on the same processor. However, this introduces new problems, especially when systems have to enforce safety or security requirements:

1. The platform must provide an environment for system functions so that they can be executed as if they run on a single processor.

2. Functions collocated on the same processor must be analyzed so that an error on a function cannot impact the others. For example, this would ensure that a function classified at a given assurance level cannot impact another at a higher one.

3. The execution platform configuration must be error-free to ensure functions isolation and requirements enforcement. Otherwise, any potential mistake would lead to misbehavior and make the whole system unstable.

First problem relates to function isolation: while collocating several functions on the same processor, designers must ensure their isolation. For that purpose, functions are separated within partitions. These are under supervision of a dedicated kernel that provides partitions execution separation using time & space isolation:

- **Time isolation** means that processing capacity is allocated to each partition so that it is executed as if it was deployed on a single processor.

- **Space isolation** means that each partition is contained in a dedicated memory segment so that one partition cannot access the segment of another. However, partitions can communicate through dedicated channels under kernel supervision so that only authorized channels are granted at run-time.

Partitions integration analysis is especially important since a partitioned kernel may collocate several functions classified at different security/safety levels. Communication channels between partitions must be verified to check that system architecture does not break safety/security requirements. For example, in the context of safety, we have to check that a partition at a low safety level cannot block a device used by a partition at a higher security level. On the other hand, on security level, we have to check that a partition cannot establish covert channel to read or write data from/to partition classified at higher security levels.

Thus, automatic kernel and partitions configuration would be a particular interest because these systems host partitions that provide critical functions and must be carefully designed. In consequence, we have to avoid any manual code production process, because it is error prone and have important economic and safety impacts [6]. This is also especially critical when creating the most important code - the one that configures the isolation policy.

2.2 Approach

To cope with identified problems, we propose the following approach focused on the following aspects:

1. A dedicated run-time, POK, that ensures time & space partitioning so that functions are executed as if they run on a single processor.

2. An analysis framework that analyzes system functions and detects potential errors.

3. An automatic partitions and kernel configuration tool from validated specifications.
This ensures enforcement of validated safety and security requirements at run-time.

Use of these three steps altogether makes a complete development process that would ease integration of heterogeneous functions, as illustrated in figure 1. First, the designer describes its functions with its properties (criticality/security levels, execution time, etc.) using an appropriate specification language (AADL). Then, we analyze system architecture to ensure that each of their requirements would be enforced while integrating (step 1 on 1). From this analysis, deployment & configuration code is automatically produced so that partitions and kernels are correctly configured according to their requirements (step 2 on 1). Finally, generated code is integrated with the POK execution platform that supports time & space isolation so that functions are integrated and separated as specified (step 3 on 1).

As a result, our approach ensures integration of several functions on the same processor while preserving all their requirements in terms of timing, security or safety.

2.3 Related Work

Partitioning operating systems that support time & space partitioning already exist. However, most existing solutions are released as commercial and proprietary software, such as Vxworks [8] or LynxOS [9]. On the free-software side, the Xtratum [13] project aims at providing partitioning support by isolating functions in RTEMS [11] instances. However, it isolates functions using virtualisation mechanisms (using a dedicated hypervisor), whereas POK relies on a traditional kernel model.

In addition, several standardization attempts were initiated over the years, to define the concepts behind partitioned operating systems. Among them, the ARINC653 avionics standard [2] specifies the services and the API that would be provided by such a system. The MILS [14] approach, dedicated to security, also details required services to integrate several functions on the same processor while ensuring a security policy.

On our side, the POK operating system provides services required by both ARINC53 & MILS: time & space partitioning support, real-time scheduling, device drivers isolation, etc. It supports ARINC653 API and provides a POSIX adaptation layer to ease application use on POK. Finally, it is released under the BSD license so that users are willing to use it as free-software and can easily improve it, depending on their needs.

On system modeling and analysis side, no framework provides the capability to both capture and analyze partitioned architecture requirements. However, this is very important for incoming projects when system architecture must be analyzed/validated and development automated, especially because traditional development methods costs are still increasing [6] and lead to security/safety issues.

Similarly, no tool automates configuration and deployment of partitioned kernel from their specifications. Usually, developers make it manually by translating system requirements into configuration/deployment code. This is still error-prone and a fault can have a significant impact (missed timing constraint/deadline, communication channel not allowed, etc.). Automating configuration is still an emerging need but would likely to take importance while system functions are still increasing.

3 The POK execution platform

The following sections gives a general presentation of the POK kernel, detailing its services and implementation internals.

3.1 Overview

POK is an operating system that isolates partitions in terms of:

- **time** by allocating a fixed time-slice for partitions tasks execution with its own scheduling policy.
- **space** by associating a unique memory segment to each partition for its code and data.
However, partitions may need to communicate. For that purpose, POK provides the inter-partitions ports mechanism that defines interfaces to exchange data between partitions. These are defined in kernel configuration so that only specified channels are allowed at run-time. Consequently, system designer has to specify the communication policy prior (which partition can communicate with another) executing the system.

Moreover, partitions may require to communicate with the external environment and so, access devices. However, sharing a device between two partitions leads to potential security or safety issues: a partition at a low security level may read data previously written by a partition classified at a higher level. To prevent this kind of issue, POK requires that each device is associated to one partition and the binding between partitions and devices must be explicitly defined during the configuration.

Partition-level services aims at supporting applications execution. It provides relevant services to create execution entities (tasks, mutexes, etc...) and helps developers:

- **Kernel interface** accesses kernel services (writing to/reading from inter-partitions communication ports, creating tasks, etc.). It relies on software interrupts (syscalls) to request specific kernel services.
- **Tasking functions** handles task-related aspects (thread management, mutex/semaphore locking, etc.).
- **Intra-partition communication** provides functions for data exchange within a partition. It supports state-of-the-art communication patterns: events or data-flow oriented, semaphores, events, etc. Intra-partitions services are the same than the one provided in the ARINC653 [2] standard.
- **Libc, Maths functions, POSIX & ARINC653 layers** are the mapping of well known standards in POK. It aims at adapting other APIs to help developers for reusing existing code in POK partitions. For example, by using this compatibility layer, developers can reuse existing software that performs POSIX calls.
- **Device drivers code** is a set of functions to support devices within partitions. It relies on the kernel-interface service when privileged instruction (see section 3.5) are used.
- **Cipher algorithms** is a set of reusable functions to crypt/decrypt data within a partition.

Finally, POK is available under BSD licence terms, supports several architectures (x86, PowerPC, SPARC/LEON) and has already been used by different research projects [15, 5, 18].

### 3.2 POK services

Figure 2 depicts the services of each layer (kernel/partition). The kernel contains the following:

- **The Hardware Abstraction Layer** provides a uniform view of the hardware.
- **Memory Management** handles memory to create or delete memory segments.
- **Time Management** counts the time elapsed since system start-up and provides time-related functions to partitions (for supporting scheduling algorithms for example).

- **Fault Handling** catches errors/exceptions (for example: divide by zero, segmentation fault, etc.) and calls the appropriate handler (partition or task that generates the fault).
- **Time Isolation** allocates time to each partition according to kernel configuration.
- **Space Isolation** switches from one memory segment to another when another partition is selected to be executed.
- **Inter-partitions communication** exchanges data from one partition to another. Inter-partitions communication is supervised by the kernel so that only explicitly defined communications channels can be established and exchange data.

![Figure 2: Services of Kernel and Partition layers](image-url)

Partition-level services aims at supporting applications execution. It provides relevant services to create execution entities (tasks, mutexes, etc...) and helps developers:

- **Kernel interface** accesses kernel services (writing to/reading from inter-partitions communication ports, creating tasks, etc.). It relies on software interrupts (syscalls) to request specific kernel services.
- **Tasking functions** handles task-related aspects (thread management, mutex/semaphore locking, etc.).
- **Intra-partition communication** provides functions for data exchange within a partition. It supports state-of-the-art communication patterns: events or data-flow oriented, semaphores, events, etc. Intra-partitions services are the same than the one provided in the ARINC653 [2] standard.
- **Libc, Maths functions, POSIX & ARINC653 layers** are the mapping of well known standards in POK. It aims at adapting other APIs to help developers for reusing existing code in POK partitions. For example, by using this compatibility layer, developers can reuse existing software that performs POSIX calls.
- **Device drivers code** is a set of functions to support devices within partitions. It relies on the kernel-interface service when privileged instruction (see section 3.5) are used.
- **Cipher algorithms** is a set of reusable functions to crypt/decrypt data within a partition.
3.3 Time isolation policy

The time partitioning policy requires a predictable method to ensure enforcement of time-slices allocation. For that purpose, POK schedules partitions using a round-robin protocol: each of them is executed periodically for a fixed amount of time. This introduces a scheduling hierarchy: partitions are first scheduled and then, tasks within the partition are scheduled according to the internal partition scheduling policy.

In order to ensure predictable communications (in terms of time), inter-partitions communication data are flushed at the major frame, which is the end of the partition scheduling period (sum of all partitions periods). In other words, once all partitions are executed, inter-partitions communications data are flushed and is available to recipients partitions during their next execution period.

An example of such a scheduling policy is provided in figure 3 with two partitions: one schedules its task using the Rate Monotonic Scheduling (RMS) policy while the other uses Earliest Deadline First (EDF). Each partition is executed during 100ms and inter-partitions communication ports are flushed each 200ms.

Within partitions, POK supports several scheduling policies: the basic FIFO algorithm or other state-of-the-art methods such as EDF or LLF. However, support for advanced scheduling algorithms is still considered as experimental.

3.4 Space isolation and inter-partitions communication

Partitions are contained in a distinct memory segment, which means that each of them has a unique space to store its code and data. Segments cannot overlap each other and a partition cannot access to memory address located outside its associated segment.

Segments properties (address, size, allocation) are defined at configuration time and cannot be modified during run-time. To avoid any access outside their segments, partitions are executed in a user context. They can only use a restricted set of instructions\(^1\), as for user processes on regular operating systems. To use privileged instructions\(^2\), they must call kernel services and performs system calls (see POK services - kernel interface in 3.2).

To guarantee space isolation, POK relies on the Memory Management Unit (known as the MMU), a dedicated hardware component usually embedded in the processor. Depending on the architecture, different protection mechanisms can be used (segmentation, pagination, etc.). For the x86 platform, it relies on the segmentation. This protection pattern has many advantages: it is very simple to program (the developer has to define memory areas using the Global Descriptor Table - GDT \([12]\)) and is predictable (other memory protection mechanisms require to perform special operations which execution time are difficult to bound). When switching from a partition to another, the POK kernel ensures that:

1. Execution context from the previous partition is clean (the elected partition cannot read data produced by the previous).
2. The memory segment associated with the elected partition is loaded.
3. Caches are flushed so that the execution context is clean as if the partition was freshly elected on a clear environment.

Finally, POK provides the inter-partition communication services, which aim at establishing and monitoring data exchange from one partition to another, as shown in figure 4, where partition 1 send data to partition 2 under the supervision of the POK kernel. These channels must be defined at configuration time so that the kernel allows only explicitly defined communication at run-time.

\(^1\)On Intel architectures, such code is usually executed in ring 3
\(^2\)On Intel architectures, privileged instructions are available for code executed in ring 0
In the example of figure 4, any attempt to send data from partition 2 to partition 1 will fail. As explained in section 3.3, inter-partitions communication are flushed when the major time frame is reached. To do so, the POK kernel copies the data of each initialized source ports to their connected destination ports, copying memory areas from one partition to another. However, to ensure that only relevant data are copied, the size of each inter-partition port is also defined at configuration time so that no additional data can be read by the destination partition.

3.5 Device drivers implementation

Compared to other approaches, POK does not execute device drivers within the kernel. Instead, their code is isolated within a single partition to:

- Avoid any impact from a device driver error (safety reason). If the driver is executed in the kernel context, a crash would lead to unexpected impacts, such as crashing the whole kernel or other partitions.

- Ensure data isolation (security reason): if the device is shared by several partitions, one classified at a low security level may read or write data on the device from another one classified at higher level.

Finally, when executing within a partition, a driver requires to have access to privileged instructions to control hardware. For that purpose, POK provides access to these privileged instructions only to the relevant partition (the one that accesses the driver). In that case, this additional access must be defined at configuration time so that the kernel grants/refuses access to privileged instructions according to its configuration policy. An example is shown in figure 6: in this system, two partitions are executed: one application partition and a driver partition that controls a device. The system defines three memory areas: one for each partition and an additional area mapped to the device memory. The driver partition may have access to this specific additional memory segment to control the hardware but the application partition would not be able to access it.

3.6 Configuration flexibility

One major advantage of POK compared to other partitioned operating systems is its fine-grained configuration policy: each service of POK, at each level (partition of kernel) can be configured and finely tuned. This would have several advantages: first, it reduces the set of functionality to the minimum and ensures a reduced memory footprint. In addition, reducing the number of functions and potentially useless code increases the code coverage of the whole system (for partitions and kernel).

However, this flexibility is particularly interesting when the system is automatically configured, according to its specification/description. Otherwise, the developer has to write the configuration manually, which would be time-consuming and error-prone. Hopefully, POK provides such configuration generation facility through its AADL code generator.
To auto-generate kernel and partitions configuration, designer must then first model its system with its requirements. This is explained in the next sections.

**AADL models**

- AADL-to-ARINC 653 code generator (Ocarina)
- Partition services (libpok)
- Kernel services (POK)

**Validation rules**

- Model validation rules
- Code production

**FIGURE 7:** Development and prototyping process using AADL models

### 4 Partitioned System Modeling

To define a system architecture that uses POK, its specification is written with an appropriate modeling language to:

1. Specify system functions using a rigorous notation (better than a simple text-based document)
2. Analyze system properties and requirements to check for potential errors before implementation efforts. (for example, check system architecture against a security policy)
3. Generate configuration & deployment code using automated tools so that we avoid hand-written code and its related problems.

For our needs, we make a specific process (as illustrated in figure 7) for the design and implementation or partitioned systems supported by POK. First of all, engineers create AADL [7] models that specify system constraints and requirements. These specifications are processed by validation tools that check for potential error/fault that could generate a problem either at run-time or even for system implementation. Finally, code generator automatically creates configuration & deployment code for both kernel and partitions according to system requirements (AADL models) so that verified properties are correctly translated into code.

Next sections present the language chosen (AADL [7]), its tailoring for partitioned architecture modeling and its use for specifications validation.

#### 4.1 AADL

The Architecture Analysis and Design Language (AADL) [7] is a modeling language that defines a set of components to model a system architecture. It provides hardware (processor, virtual processor, memory, etc.) and software components (thread, data, subprogram, etc.) as well as mechanisms to associate them (for example, model the association of a thread to a processor or a memory). The language itself can be extended using:

1. **Properties** to specify components requirements or constraints (for example, the period of a task). The core language provides a set of predefined properties but system designers can define their own.
2. **Annex language** to associate a piece of code of another language to a component. By doing so, it provides a way to interface AADL components with third-party language for the association of other system aspects.

AADL provides both graphical and textual notations. The graphical one is more convenient for system designers while the text-based one is easier to process by system analyzers/code generators. A complete description of the language, written by its authors, is available in [1].

#### 4.2 Tailoring the AADL for partitioned architectures

The AADL provides a convenient way to describe both software and hardware architectures but one may need guidance to know how to use it to describe partitioned architectures and especially how to use AADL components to describe POK run-time entities (partitions, memory segments, etc.) and their requirements/constraints (partitions scheduling, memory segments size, etc.).

For that purpose, we design modeling patterns, which consist of a set of modeling rules to be used by system designers to represent partitioned architectures. The main idea is to restrict the set of components to be used in a model and which properties
must be associated to the model in order to make a complete description of a partitioned architecture.

These modeling patterns has been first design for POK [15] and then, be standardized as an annex of the AADL standard [3]. Most important patterns define how to use AADL to model a partition (process bound to a memory and a virtual processor to model the partition, its allocated memory segment and its execution context), intra-partition communication (connections of AADL ports between thread located within the same process) or inter-partition communication (connection of AADL ports between process components). A full description of these modeling patterns is available in [15, 3, 5].

Next section presents our tools that check models to validate partitioned architectures requirements.

4.3 System validation

Once system architecture is specified using AADL models, analysis tools process models and check their compliance with several requirements/guidelines. In the context of POK, we design the following rules:

- **Modeling patterns enforcement rules** (rules described in previous paragraph). This ensures that models are complete, with all required components/properties and they can be processed to configure the kernel and partitions with code generators.

- **Handling of potential safety issues rule**. This one checks that all potential error will be recovered and report to the user which error is not handled by a partition or a task. For example, if the designer didn’t specify a recovery subprogram when a memory fault is raised while execution a partition, the analyzer will report an error.

- **Security analysis rules**. This aims at checking the architecture against a security policy. Depending on its own classification level and the one of the data they produce or receive, a partition may be compliant with a specific security policy. On the other hand, the security policy defines which operations are legal so that our tool can automatically checks for architecture compliance with them. Actually, our tool checks state-of-the-art security policies such as Bell-Lapadula [17] or Biba [16].

Other validation can be issued on the models to check either the correctness of the architecture (for example: resource dimensions with the analysis of memory segments size with respect to the associated partitions requirements - size of tasks size, etc.) or validation of a specific constraint (for example, partitions scheduling).

Once models are validated and the architecture considered as correct, our code generator, Ocarina, produces configuration & deployment code for both the kernel and its associated partitions. Next sections detail this process.

5 Automatic Configuration & Deployment of Partitioned Architectures

Once system requirements and constraints have been validated, the Ocarina AADL tool-suite processes models and generates configuration & deployment code for both kernel and partitions, as shown in figure 7. Next sections describe the process [5], highlighting its benefits regarding safety-critical systems needs (predictability, safety assurance, etc.) and constraints (code coverage, etc.).

5.1 Code generation Overview

Our code generator process [5] consists in analyzing AADL models, browsing its components hierarchy and, for each of them, generates appropriate code to create and configure system entities (as depicted in figure 7). The process creates the kernel configuration code (partitions time slots, memory segments assignment for each partition, etc.) and partitions configuration code (services to be used, required memory, intra-partition communication policy, etc.).

For example, when the code generator visits a memory component, it generates code to create a new memory segment. Then, it inspects the model to retrieve the associated partition and configure the kernel to associate the appropriate partition with this segment. Each AADL entity and property is used to produce system configuration code so that the resulting process creates a complete code, almost ready to be compiled. Next section explains which part of the code is automatically generated and which still requires some manual code to have a complete executable system.
5.2 Kernel and partitions configuration

At first, the code generator browses components hierarchy to create the kernel and partitions configuration code. For that purposes, it analyzes the following AADL components:

- **processor**: for the kernel configuration. The processor components specifies the time slots allocated for each partition so that they are translated into C code to configure the kernel time isolation policy.

- **virtual processor**: for partition run-time configuration. Based on the properties of this component, the code generator produces code that configures partitions services (need for memory allocation, POSIX or ARINC653 API support, etc.)

- **process** and its **features**: the process contains all partitions resources (thread and data) so that necessary amount of resources is allocated in the kernel and its partition. The process features represent inter-partitions communication ports. When analyzing such entities, the code generator configures these ports and their connection within the kernel so that only communication channels specified in the model would be granted at run-time.

- **memory**: as it represents a memory segment, the code generator produces configuration code that instantiates a new memory segment in the main system memory and associates it to its bound process component (that corresponds to a partition). In consequence, at run-time, each partition will be associated with a memory segment that has the properties (address, size, etc.) specified in the model.

Once this configuration code has been generated, Ocarina also generates the behavior code of the system, the one executed by each task. Next section details this step.

5.3 Behavior code generation

The behavior part corresponds to the code that uses partitions resources to execute application functions. It consists in getting/putting data from/to the execution environment (for example, by using inter-partitions communication ports), calling application functions and managing tasks execution (put a task in the sleep mode when its period is completed, etc.).

As for the configuration code generation, the behavior code of application is generated from a predefined set of AADL components:

- A **thread** component specifies a task that potentially communicates using its features. For each thread, Ocarina generates code that gets the data from its IN features, performs the call sequence to its subprogram and sends produced output using its OUT features.

- A **data** component represents a data located in a partition, shared among its tasks and protected against potential race conditions using locking mechanisms (mutex, semaphore, etc.). For each task that uses the data, the code generator produces code that locks it, modifies it and finally releases it.

- A **subprogram** component references object code implemented either in C, Ada or even assembly languages. This is just a reference to low-level implementation so when visiting such component, the code generator creates a function call to the object code. Finally, it also configures the build system in order to integrate the object code in the partition binary.

Then, almost all the code of the system is produced by Ocarina. The user has to provide the application-level code, the one referenced by AADL subprogram components and automatically called by the behavior code generated for each partition. Next section details the benefits of this process with respect to high-integrity systems requirements.

5.4 Benefits of Code Generation

First of all, the use of such a process requires to specify system architecture using a modeling language, which makes the whole process more rigorous than just using a text-based specification document. Moreover, the process brings the following benefits:

1. **Early error detection**
2. **Syntax/semantic error avoidance**
3. **Specifications requirements enforcement**

By using validated models as a language source for system implementation, the development process detects specification errors at the earliest when such problems are difficult to track and usually detected during tests (at best) or production (at worst)
phases. By identifying these errors prior to the implementation, we save a significant number of problems and save development costs.

Then, by automating code production with code generators such as Ocarina, we rely on established generation patterns that output the same block of code according a predefined AADL block. Use of such code patterns avoids all error related to hand-written code that usually introduce syntax/semantic errors that are difficult to track and require code analysis tools and reviews to be found.

Finally, a particular interest is the enforcement of the specifications. Implementation compliance with the specifications is usually checked during manual code review. However, this is long, costly and also error-prone [6] since it relies on a manual inspection. By automating the code production from the specifications and by using code generation patterns, implementation code ensures specifications enforcement and so, would reduce development costs while improving system safety and robustness.

6 Case-Study

We illustrate the POK dedicated design process through a basic example with two partitions: one that sends an integer to another which do some processing and outputs its result.

6.1 Overview

To focus on the design process, we limit the behavior of the system to a basic example: the sender partition executes a task each second, increments an integer (starting from 0) and sends the result to the receiver partition using an inter-partition communication channel. Then, the receiver partition runs a task activated each 500ms that retrieves the value sent by the other partition and triggers a divide by zero, depending on the received value. This, the second partition would execute the dedicated handler that recover from the fault it generates.

System architecture is made of two partitions each one executed during a 500ms time slot (so that the major frame is 1 second) and stored in a segment of 120Kbytes. Each partition contains a periodic thread (either the sender or the receiver) that calls one subprogram (the one that sends or receives the integer). Partitions communicate through two queuing ports, connection with a inter-partition channel. Finally, the recovery policy requires to stop the kernel when an error is raised at kernel-level and restart other components (partition, task) when one of them triggers an error/exception.

![FIGURE 8: AADL model of the case-study](image)

6.2 AADL modeling

The graphic version of the case-study is illustrated in figure 8. It maps the requirements previously described with specific execution components:

- **Partitions** use process (recv.prs and snd.prs for application aspects), virtual processor (part1 and part2 of pok.kernel for run-time concerns) and memory (segments specification) components. Arrows on the graphical model explicit their association.
- **Tasks** of each partition are defined with a thread component within each partition (either recv.thr or snd.thr).
- **Queuing ports** are added on each partition and connected within their containing system (case_study_osadl). It explicitly defines communication channel that can be requested by partitions at run-time.

Definition of system recovery policy uses AADL properties, included only in the textual representation. The following listing provides an example of the definition of partition run-times with their recovery strategy (the Recovery_Errors and Recovery_Actions properties). It also includes a processor component that represents the partitioning kernel with the two partition environments (part1 and part2) and its associated time isolation policy (Slots, Slots_Allocation and Major_Frame properties).

---

3For example, the C code if (c = 1) statement; is often an error even if legal and would likely be if (c == 1) statement;
1 virtual processor implementation partition.common
properties
3 Scheduler " => RR;
Additional.Features =: (console, libc_stdio);
5 Recovery_Errors = (Illegal_Request, Partition_Init);
7 Recovery_Actions = (Partition_Reboot, Partition_Stop);
9 end partition.common;
11 processor implementation pok_kernel.i
13 subcomponents
15 part1 : virtual processor partition.common;
17 part2 : virtual processor partition.common;
19 Major_Frame =: 1000ms;
21 Slots_Allocation =: (reference (part1), reference (part2));
23 Recovery_Errors = (Kernel_Init, Kernel_Scheduling);
25 Recovery_Actions = (Kernel_Stop, Kernel_Scheduler);
end pok_kernel.i;

The following listing also illustrates the definition of the complete system with its processor, partitions, memory and the connection of the inter-partitions queuing ports.

1 system implementation osal.i
subcomponents
3 cpu : processor pok_kernel.i;
5 receiver : process recvpri.i;
7 mem : memory ram.i;
connections
port sender.pdataout -> receiver.pdatain;
9 properties
11 actual_processor_binding => (reference (cpu.part1)) applies to sender;
13 actual_processor_binding => (reference (cpu.part2)) applies to receiver;
15 actual_memory_binding => (reference (mem.seg1)) applies to sender;
17 actual_memory_binding => (reference (mem.seg2)) applies to receiver;
end osal.i;

6.3 AADL model validation

We validate system architecture using the Requirements Enforcement Analysis Language (REAL [19]) tool. It analyzes AADL components hierarchy and validates it against theorems. By using this tool, we validate model structure and compliance with modeling patterns. We illustrate that using two theorems.

The first one (see listing below) checks compliance of system architecture regarding space isolation and memory segments definition. It processes memory components contained in the main system (that corresponds to the main memory - like RAM) and ensures that each one contains memory sub-components to specify memory segment with their size (by checking the definition of the Word_Count property).

theorem Contains_Memories
2 foreach s in system_set do
mainmem := (y in Memory_Set |
is_subcomponent_of (y, s));
6
partitionsmem := {x in Memory_Set |
is_subcomponent_of (x, mainmem)};
8 check ((cardinal (mainmem) > 0) and
10 (Property_Exists (partitionsmem, "Word_Count"))
end Contains_Memories;

Second validation theorem (see listing below) checks the major frame compliance (see section 3.3 for its definition) with partitions time slots. To do so, the theorem processes each processor component of the system and checks that the value of the Major_Frame property is equal to the sum of the Slots value.

1 theorem scheduling_major_frame
3 foreach cpu in processor_set do
5
check ( (property \exists (cpu, "Major_Frame") and
( (float (property (cpu, "Major_Frame")) =
5 sum (property (cpu, "Slots")))))
end scheduling_major_frame;

Other validation theorems can be designed and added to the process to automatically check for enforcement of other requirements or also verify specific modeling patterns. Interested readers may refer to the POK distribution available on the official POK website [10]: it contains a complete REAL [19] theorem library to check all modeling patterns related to partitioned architectures.

6.4 Code generation & metrics

Configuration and deployment code is automatically generated from AADL specifications. As models have been previously validated, produced output is expected to enforce system requirements.

Among all generated files, one especially important is deployment.h, which defines constant and macro that configure kernel services and set resources dimensions (amount of ports, partitions, etc.). The following listing provides an overview of the file generated from the model of this case-study.

2/* other configuration directives
4*/
6#define POK_NEEDS_CONSOLE 1
8#define POK_CONFIG_SLOTS (500, 500)
10#define POK_CONFIG_SLOTS (0, 1)
12#define POK_CONFIG_MAJ_FRAME 1000
14#define POK_CONFIG_PORTS 2

We can check and verify that configuration directives enforces model requirements: two partitions are defined, scheduling slots of each partitions (500ms) is correctly mapped, as well as the major frame (1s).

The code generation process not only configures
the kernel but produces partitions configuration and behavior code. Developers only have to write application code, which corresponds to the functional part of the system. In this case-study, it consists of two functions: one that outputs an integer and stores it as a function argument (the one used on the sender side) and another that takes one integer as argument and process it (receiver side). The code provided by the developer is shown in the following listing, demonstrating that code production automation reduces manual code production activities.

In the following application code, the receiver part raises a division by zero exception when result of \((t + 1)\%3 = 0\) (line 16 of the application code). According to the recovery policy, when such a condition is met, the partition restarts. To show graphically that the partition is correctly restarted, we also output the number of times the function is executed using variable step. Its initial value is stored in the data from the partition binary so that when reloading the partition, the initial value is set again in the variable.

```c
1 void user_send (int* t)
2 {
3     static int n = 0;
4     printf ("Sent value \n", n);
5     n = n + 1;
6     *t = n;
7 }
8 static int step = 0;
9
10 void user_receive (int t)
11 {
12     int d;
13     d = (t + 1) % 3;
14     printf ("Step\n", step++);
15     printf ("Received value\n", t);
16     printf ("Computed value\n", t / d);
17 }
```

Generated application is compiled for Intel (x86) architecture and produces the following output during execution:

```
...Step 3
Received value 5
[KERNEL] Raise divide by zero error
Step 0
Received value 0
Computed value 0
Sent value 8
...
```

One may notice that when the faulty condition of the application code \((t + 1)\%3 = 0\) (line 16 of the user application code) is reached (in that case, when receiving value 5), the receiver partition is restarted. Initial value of variable step (printed in the line Step 0) is set back to 0, showing that the partition binary has been re-loaded.

To assess the memory consumption of generated systems, we also report generated kernel and partitions sizes (see table below). Partitions size is similar: they contain the same functionality and differ only by their application code. Both of them have a small size: 11kB for a complete system that embeds run-time functions for the support of user application. This demonstrates the lightweight aspect of the approach. Kernel size is also very small, especially for such a system that provides critical functions regarding safety and security issues.

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>26 kB</td>
</tr>
<tr>
<td>Partition 1</td>
<td>11 kB</td>
</tr>
<tr>
<td>Partition 2</td>
<td>11 kB</td>
</tr>
</tbody>
</table>

7 Conclusions & Perspectives

This article presents POK, a BSD-licensed operating system that supports partitioning with time & space isolation. It also provides layers to ease deployment of existing code that uses established standards such as POSIX or ARINC653.

Beyond the operating system itself, POK relies on a complete tool-chain to automate its configuration & deployment and ease partitioned systems development. It aims at specifying system architecture and properties using a modeling language, AADL and verifying its requirements using dedicated analysis tools that process these specifications. Then, from this validated specifications, our tool-chain automatically generates code that configures/de部署 kernel/partitions and execute application code provided by the user. This ensures specifications requirements enforcement and avoid all errors related to usual development process.

7.1 Perspectives

The domain of partitioned architecture is still emerging and there is many potential open perspectives. On the kernel side, there is a need for more hardware support (devices, architectures, etc.) and a wider support of existing standards, as for the ARINC653 layer (for example, to support the second part of the standard).

On the modeling and analysis part, there is a strong need to connect AADL models with other
system representation or specifications. In particular, the production of AADL models could be automated from text-based specifications and model components/entities could be associated to external specifications such as DOORS. This would ease requirements traceability, which is a special interest for the design of high-integrity systems, when designers have to ensure that high-levels requirements are correctly mapped in the implementation.

References


[16] Biba, K.J. - Integrity considerations for secure computer systems. Technical report, MITRE


[18] Julien Delange - Intégration de la securité et de la sureté de fonctionnement dans la construction d’intergiciels critiques - PhDThesis

[19] Olivier Gilles and Jérôme Hugues - Validating requirements at model-level in Ingénierie Dirige par les modèles (IDM08)
POK, an ARINC653-compliant operating system released under the BSD license
D-Case Editor: A Typed Assurance Case Editor

Yutaka Matsuno
The University of Tokyo, Japan
JST, CREST
matsu@cc.u-tokyo.ac.jp

Abstract

System assurance has become an important issue in many system domains, especially in safety-critical domain. Recently, assurance cases [3] have been getting much attentions for the purpose. We demonstrate D-Case Editor [10], which is an assurance cases editor being developed in DEOS (Dependable Embedded Operating System for Practical Uses) project funded by Japan Science and Technology Agency. D-Case Editor has been implemented as an Eclipse plug-in using Eclipse GMF framework. Its characteristics are (1) supporting GSN (Goal Structuring Notation) [8], (2) GSN pattern library function and prototype type checking function [9], and (3) consistency checking function by an advanced proof assistant tool [13]. To achieve these characteristics, we have exploited types in several ways. In this paper, we briefly introduce assurance cases, and demonstrate the functions of D-Case Editor. Because it has been implemented on Eclipse, it is interesting to make a tool chain with existing development tools of Eclipse. D-Case Editor is available as an open source in the following web page: http://www.il.is.s.u-tokyo.ac.jp/deos/dcase/.

1 Introduction

System assurance has become a great importance in many industrial sectors. Safety cases (assurance cases for safety of systems) are required to submit to certification bodies for developing and operating safety critical systems, e.g., automotive, railway, defense, nuclear plants and sea oils. There are several standards, e.g. EUROCONTROL [5], Rail Yellow Book [12] and MoD Defence Standard 00-56, which mandate the use of safety cases.

There are several definitions for assurance cases [3]. We show one of such definitions as follows [1].

“a documented body of evidence that provides a convincing and valid argument that a system is adequately dependable for a given application in a given environment.”

Assurance cases are often written in structured documents using a graphical notation to ease the difficulty of writing and certifying them. Goal Structuring Notation (GSN) is one of such notations [8]. Writing assurance cases and reusing them in a cost effective way is a critical issue for organisations. Patterns and their supporting constructs are proposed in GSN for the reuse of existing assurance cases, which includes parameterized expressions.

Assurance cases have been recognized as a key method for dependability of systems. However, currently there have been not so much tools for assurance cases (very few in open source.) A notable tool is ASCE tools [1], which has been widely used in several areas such as defense, safety critical area, and medical devices.

To make assurance case more familiar to developers who are using open sources tools, we have released D-Case Editor, an open source assurance case editor implemented on Eclipse GMF. The web page is http://www.il.is.s.u-tokyo.ac.jp/deos/dcase/. The characteristics are as follows.

1. Supporting GSN (Goal Structuring Notation) [8],
2. GSN pattern library function and prototype type checking function [9], and
3. Consistency checking function by an advanced
To achieve these characteristics, we have exploited *types* in several ways. For example, we introduce types for variables used in GSN patterns [4]. Our intention is to make assurance cases to be shared among various tools for wider use. Introducing types is one attempt for the purpose.

The structure of this paper is as follows. In Section 2, we introduce assurance cases and patterns, and some standardization efforts for assurance cases. Section 3 introduces several functions of D-Case Editor. Section 4 states a few concluding remarks.

## 2 Background Knowledge

### 2.1 Goal Structuring Notation (GSN)

*Goal Structuring Notation (GSN)* is introduced by Tim Kelly and his colleagues at University of York [8]. It is a graphical notation for assurance cases. GSN is widely used for safety cases. Some safety cases written in GSN are publicly available [2]. We briefly explain constructs and their meanings in GSN. Arguments in GSN are structured as trees with a few kinds of nodes, including: *goal* nodes for claims to be argued for, *strategy* nodes for reasoning steps that decompose a goal into sub goals, and *evidence* nodes for references to direct evidences that respective goals hold. Figure 1 is a simple example of GSN. The root of the tree must be a goal node, called top goal, which is the claim to be argued (*G*1 in Figure 1.) For *G*1, a context node *C*1 is attached to complement *G*1. Context nodes are used to describe the context (environment) of the goal attached to. A goal node is decomposed through a strategy node (*S*1) into sub goal nodes (*G*2 and *G*3). The strategy node contains an explanation, or reason, for why the goal is achieved when the sub goals are achieved. *S*1 explains the way of arguing (argue over each possible fault: A and B). When successive decompositions reach a sub goal (*G*2) that has a direct evidence of success, an evidence node (*E*1) referring to the evidence is added. Here we use a result of fault tree analysis (FTA) as the evidence. For the sub goal (*G*3) that is not decomposed nor supported by evidences, a node (a diamond) of type *undeveloped* is attached to highlight the incomplete status of the case. The assurance case in Figure 1 is written with D-Case Editor.

### 2.2 GSN Patterns

Writing and certifying assurance cases are difficult because they tend to be huge and complex, and they require domain specific knowledge of target systems. To ease the difficulties, it has been recognized that assurance case patterns should be collected and available for reuse, similarly to design patterns in object oriented languages. There have been several publicly available GSN patterns ([7, 14, 4]).

Figure 2 is a simple example of GSN patterns in [4]. The top-level goal of system safety (*G*1) is re-expressed as a number of goals of functional safety (*G*2) as part of the strategy identified by S1. In order to support this strategy, it is necessary to have identified all system functions affecting overall safety (*C*1) e.g. through Functional Hazard Analysis (FHA). In addition, it is also necessary to put forward (and develop) the claim that either all the identified functions are independent, and therefore have no interactions that could give rise to hazards (*G*4) or that any interactions that have been identified are non-hazardous (*G*3).

Figure 2 includes main GSN extensions for GSN patterns, as defined in [6]:

- Parameterized expressions. \{System \textit{X}\} and \{Function \textit{Y}\} are parametarised expressions. We can instantiate \textit{X} and \textit{Y} by appropriate (possibly safety critical) system and function, respectively.

- Uninstantiated. Triangles (\triangle) attached to nodes indicate that the nodes contain uninstantiated parameterised expressions. To instantiate the GSN pattern as an assurance case, we need to instantiate the expressions.

- 1 to many expressions (multiplicity). Number of functions are different by the target system. We can instantiate the number of functions (n) for the target system.

- Choice. By this extension, we can choose appropriate goals for the target system.

### 2.3 Assurance Cases and Standardization Efforts

Two major graphical notations for assurance cases are GSN and CAE (*Claims, Arguments, and Evidence*) [1]. There are two standardization efforts for assurance cases; the system assurance task force at the OMG (Object Management Group) and GSN standardization effort [6]. OMG has standardized
Figure 1: A simple GSN Example

Figure 2: An example of GSN patterns [4]
the meta-model for assurance cases called ARM (Argument Metamodel) [11] by which both notations are in fact interchangeable. The main aim of the ARM is to align two major notations and facilitates the tool support. Unfortunately it only reflects main constructs between the two, and some specific features, which are not compatible are missing from it. For instance, patterns are not included in the ARM.

## Overview of D-Case Editor

Figure 3 shows a screen shot of D-Case Editor. Users can draw GSN diagrams in the canvas. In the right, there is a pattern library. From the library, users can choose already existing, good assurance case patterns and fragments, and copy to the canvas. Current D-Case Editor has the following functions (some functions are omitted in current version.) Consistency checking with an advanced proof assistant tool [13] will be available soon.

- Checks on the graph structure of D-Case (e.g. no-cycle, no-evidence directly below a strategy, etc.)
- External info via url can be attached to a goal.
- “Patterns” with typed parameters can be registered and recalled with parameter instantiations.
- Graphical diff to compare two D-Cases.
- A “ticket” in Redmine, a project management web application, can be attached to a goal; the ticket’s status can be reflected graphically in D-Case (color change.)
- Monitoring: a url to be polled by Editor can be attached to a node; the answer is dynamically reflected in D-Case (color change.)
- Scoring: calculates a weighted score for a D-Case indicating how much of it is completed.
- connection with uml2tools: generating a D-Case subtree for a component diagram data.

Among these functions, we show how patterns with typed parameters can be registered and recalled with parameter instantiations. Current implementation is limited that variables and types can only be declared in the top level of the GSN term. Declaration of variables and types are written in an XML file, as shown in Figure 4. In Figure 4, variables STATUS, CPU, USAGE, and MESSAGE are declared, and given types enum, int, double, and string, respectively. Furthermore, these types are given useful restrictions such that the value of CPU (this variable is intended as the CPU resource usage rate of the target system) is restricted within 0 – 100%. Users of D-Case Editor can assign values to these variables via the parameter setting window. If a user mis-assigned a value (e.g., 150 for CPU), then D-Case Editor reports the type error. As far as we know, there is not any assurance case editor which has such parameterized expressions and type checking mechanism. We plan to implement the type checking mechanism in Section 3.

## Concluding Remarks

We have presented our assurance case editor, called D-Case Editor. It has been implemented as an Eclipse plug-in using Eclipse GMF, and released as an open source. We hope that D-Case Editor would contribute to make assurance cases more familiar to developers by making a tool chain of D-Case Editor with Eclipse and other development tools. We plan to comply to OMG ARM [11] and other international standards related to assurance cases in next release.

## References

Figure 3: A Screen Shot of D-Case Editor

<?xml version="1.0" encoding="UTF-8"?>
<!-- all element names and attributes are case sensitive -->
<dataType>
  <parameter name="STATUS" type="enum">
    <items>
      <item value="NORMAL"/>
      <item value="ERROR"/>
      <item value="RUNNING"/>
      <item value="SATISFIED"/>
    </items>
  </parameter>
  <parameter name="CPU" type="int">
    <range min="0" max="100"/>
  </parameter>
  <parameter name="USAGE" type="double">
    <range min="0.00" max="999.99" digit="2" inc="0.02"/>
  </parameter>
  <parameter name="MESSAGE" type="string">
    <length min="0" max="20"/>
  </parameter>
</dataType>

Figure 4: Variables and Type Declarations XML file for D-Case Editor


A FLOSS Library for the Safety Domain

Peter Krebs, Andreas Platschek, Hans Tschürtz
Vienna Institute for Safety & Systems Engineering
FH Campus Wien - University of Applied Sciences
Favoritenstrasse 226, Vienna, Austria
{peter.krebs, andreas.platschek, hans.tschuertz}@fh-campuswien.ac.at

Abstract

Safety-critical software is usually implemented under the constraints of one or more standards which demand evidence that these constraints were honoured. This leads to higher implementation effort and require in-depth knowledge on the programming languages and interfaces used by each individual programmer – often to avoid making the same mistakes over and over again.

To facilitate development under such conditions, a library of frequently used functions and algorithms which adhere to certain safety constraints complemented by specific evidence suitable for proof against a standard would be of great help. Inside this paper we present such a library written in ANSI-C, named "safety_lib", which emerged as a by-product of an application developed for Safety Integrity Level (SIL) 2 certification according to IEC 61508 at the Vienna Institute for Safety & Systems Engineering.

The main intention of this paper is to show the benefits of using such a library in safety-critical development and the reasons for its planned release under a FLOSS license. Furthermore, we want to invite everyone to use the safety_lib and participate in its development to improve both its code and evidence base.

Our hypothesis is that the joint development of a library for safety-critical applications can not only save development and certification costs, but – even more important – increase safety through better and more intense reviews carried out by a community instead of just individual developers.

1 Introduction

Implementing a safety-critical system in software requires additional effort from the developers compared to the usual, non-safety, development process. Depending on the specific domain and the standards followed, this usually entails the adherence to certain implementation constraints and the availability of evidence to prove (with sufficiently high confidence) that the software was actually built under these requirements. Defining and obtaining suitable proof is in fact one of the most time-consuming and expensive tasks in the development of a safety-critical system.

The implementation of simple, well-understood algorithms is rarely of concern. However, the increasing demand for safety-critical functions purely implemented in software, as shown for example by [1], will pose a number of new problems. This especially concerns two important trends in the design of modern systems: the ability to communicate over a network via standardised and widely-available protocols such as TCP/IP and the utilisation of concurrency for enhanced performance and responsiveness. Both aspects significantly increase the complexity of a software and the probability of residual faults due to the difficulty in the design and testing of asynchronous and indeterministic behaviour. This is often aggravated by the available interfaces and their peculiarities which provide further opportunities to implement an incorrect or 'unstable' system.

The combined problem of increased software complexity and evidence for safety can not be solved by a single measure. However, to facilitate the implementation and satisfy the need for proof at the same time, a viable approach would be to coalesce complex and often used functions in a qualified software
library. Ideally, this 'safety library' would be easy to (re-)use, well-tested and complemented by a sufficient body of evidence to not only rise the safety of the system itself but also to speed up a certification process against a specific standard.

Inside this paper we present such a library written for ANSI-C, named 'safety.lib', to achieve three objectives: First, we want to point out an approach to satisfy the demands of safety standards in a meaningful way. Second, by pointing out the problems our library helps to solve during implementation, we hope to raise the awareness of certain ill-understood and often dangerous practices when implementing a safety-critical system in C. Third, it is our intent to leverage the know-how of the community to improve the safety of the library and its evidence through extensive reviews and enhancements. To this end we are preparing the release of the safety.lib under a FLOSS license and invite everyone to participate in its further development.

1.1 Content

The first part gives an overview on the requirements imposed by certain safety standards on the implementation of a safety-critical SW application and how they affect the source code. Furthermore, the impact on safety of portability and code reuse is discussed.

The second part explains how a software library can provide evidence for the above mentioned requirements and opportunities for improving safety at the same time. After this, the safety.lib is introduced via a number of examples that demonstrate certain dangerous and unsafe implementation practices and the approach of the library in preventing problems. This is complemented by some technical facts and an outline of concrete evidence the safety.lib offers.

The remainder of this paper deals with possible alternatives and the approach to improve the functionality and safety of the library via the FLOSS community.

2 Constraints

Developing a safety-critical system inevitably involves one or more standards and the effort to prove that the requirements of those standards are met. Regarding software implementation the requirements usually consist of a number of constraints for the design and coding of the source code. While certain standards might require specific methodology, there is a set of constraints common to most of them:

Coding guidelines - A coding guideline restricts the functionality of a given programming language to a certain subset by excluding functions and constructs which are deemed unsafe. A well-known example are the MISRA-C guidelines for the ANSI-C language which disallow a number of standard library functions (such as malloc() or printf()). It can be expected, that the adherence to a given coding guideline will force a programmer to abandon some ‘standard solutions’ in favour of code that complies to the restrictions (e. g. adding pre/postconditions to standard function calls, avoidance of dynamic memory allocation).

Coding style guide - In contrast to a guideline, a coding style guide only affects the format of the source code and does not restrict the feature set of a language. The primary purpose is to enforce a consistent style of the code in order to improve readability for code reviews and maintainability in case of changing developers. A style guide dictates, for example, the length and indentation of code lines.

Modularity - Splitting the code up into small, self-contained components is beneficial for reviewing and testing and facilitates the assessment of the impact of changes on the overall safety. The most common approach to ensure the modularity of code is by enforcing thresholds for code complexity metrics such as lines of code (LOC) and cyclomatic complexity[2].

Defined interfaces - Software modules and their functions should be consistent, easy to use and unambiguous in the meaning of their parameters. This can be achieved by enforcing limits for the number of parameters, avoiding overly generic functions and via documentation of the interfaces.

Static analysis - Static analysers work directly on the source code and can detect problems which are not considered by the translator such as out-of-bounds access or locking errors. In addition, manual code reviews provide coverage of problems which can not be adequately checked by a tool.

Testing - Testing complements static analysis by asserting the correct behaviour of a software during runtime. As there is no way to exhaustively test an even moderately complex system, test coverage metrics are usually defined
to gauge the thoroughness of a test suite (e. g. statement/branch coverage).

Table 1 lists the clauses of the important safety standards IEC 61508[3], EN 50128[4], DO-178B[5], ISO 26262[6] and ISO 13849[7] pertaining to the above mentioned points.

In addition to the normative requirements a software should exhibit additional properties:

<table>
<thead>
<tr>
<th>Standard</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEC 61508</td>
<td>CG/CSG: Table A.3, Point 3</td>
</tr>
<tr>
<td></td>
<td>M: Table B.9, Points 1/2/3/4</td>
</tr>
<tr>
<td></td>
<td>DI: Table B.9, Point 6</td>
</tr>
<tr>
<td></td>
<td>SA: Table B.8</td>
</tr>
<tr>
<td></td>
<td>T: Tables B.2 and B.3</td>
</tr>
<tr>
<td>EN 50128</td>
<td>CG: Table A.4, Point 10</td>
</tr>
<tr>
<td></td>
<td>CSG: Table A.12, Point 2</td>
</tr>
<tr>
<td></td>
<td>M: Table A.20, Points 1/2/3</td>
</tr>
<tr>
<td></td>
<td>DI: Table A.20, Point 5</td>
</tr>
<tr>
<td></td>
<td>SA: Table A.19</td>
</tr>
<tr>
<td></td>
<td>T: Table A.13</td>
</tr>
<tr>
<td>DO-178B</td>
<td>CG/CSG: 11.8</td>
</tr>
<tr>
<td></td>
<td>M: -</td>
</tr>
<tr>
<td></td>
<td>DI: -</td>
</tr>
<tr>
<td></td>
<td>SA: -</td>
</tr>
<tr>
<td></td>
<td>T: 6.4</td>
</tr>
<tr>
<td>ISO 26262</td>
<td>CG/CSG: 5.4.6</td>
</tr>
<tr>
<td></td>
<td>M: 7.4.3</td>
</tr>
<tr>
<td></td>
<td>DI: -</td>
</tr>
<tr>
<td></td>
<td>SA: 8.4.5</td>
</tr>
<tr>
<td></td>
<td>T: 9</td>
</tr>
<tr>
<td>ISO 13849</td>
<td>CG/CSG: Annex J, Clause 4</td>
</tr>
<tr>
<td></td>
<td>M: 4.6.2, 4.6.3</td>
</tr>
<tr>
<td></td>
<td>DI: 4.6.3, Point g</td>
</tr>
<tr>
<td></td>
<td>SA: 4.6.2</td>
</tr>
<tr>
<td></td>
<td>T: 4.6.2, 4.6.3</td>
</tr>
</tbody>
</table>

CG = Coding guideline, CSG = Coding style guide, M = Modularity, DI = Defined interfaces, SA = Static analysis, T = Testing

**TABLE 1:** Safety standards and their requirements for implementation.

Portability enables the usage of the software on different systems without the need to modify large parts. This not only saves time and money but is also beneficial for safety as changes to an existing code might easily introduce new bugs. One way to achieve this is to rely on standardised interfaces as much as possible - e. g. by exclusively using functions defined by the POSIX standard. The downside of portability is the increased effort of generic programming and the lack of certain non-standard features.

Reuse of software elements can speed up the development process and can be a strong safety argument - when the reused component can be proven to be safe enough. The necessary data for this 'proven in use' argument is, however, often difficult to retrieve especially with closed-source/proprietary code. FLOSS software plays an important role here as not only the code itself but also ancillary data (such as bug reports) is publicly available.

### 3 The Library Approach

Fulfilling the mentioned subset of requirements for a piece of safety-critical software obviously requires effort proportional to the amount of code. In order to reduce the workload, parts of the code can be substituted by pre-existing one in the form of a code library which already complies to the safety constraints. This approach is actually sanctified by most safety standards which refer to this pre-existing code as 'Commercial/Components off the Shelf' (COTS) and basically provide two ways to assess if a COTS component is acceptable:

**Proven in use** refers to a history of operational use without known safety-related failures for a specified minimum time \(^1\).

**Verification Evidence** proofs that the pre-existing code adheres to the safety requirements of a standard for a given safety integrity level (or similar metric).

Considering a software library developed by a 3rd party, both aspects can be covered as follows:

- The history of defects (e. g. bug reports) shows whether there are known safety-related issues in a specific version.
- The source code itself can be analysed and reviewed for the presence/absence of faults.
- Additional data like test results, metric reports and analyser output provide safety-specific evidence.

The first two points are usually not feasible with closed-source/proprietary libraries and make the use

\(^1\)For example, IEC 61508, part 7, clause C.2.10.1 states the minimum operating time as 1 year.
of FLOSS software in safety-critical systems attractive. However, most FLOSS libraries do not provide the specific evidence needed to justify their usage in the context of a safety standard. While it is theoretically possible to extract the necessary data – given the availability of the source code – time and budget constraints usually prevent this. Furthermore, FLOSS code is rarely written in accordance to safety requirements. Instead, developers might opt to write safety-critical code from the scratch, effectively reinventing the wheel over and over and wasting the benefits provided through the FLOSS approach – namely a large number of potential reviewers/testers and availability of defect history.

To cope with this problem, a software library is needed which fulfils the following requirements:

- The source code must be fully available.
- The history of changes and bugs must be obtainable.
- All library code must comply to the common set of safety requirements imposed by the standards.
- Sufficient evidence to proof the compliance of the code must be available.
- Modifications and corrections must follow a defined process.
- The library must be portable to a large number of platforms.
- The interfaces must be fully documented.

In the remaining parts of the paper we present a software library written in ANSI-C, named the ‘safety\lib’, developed in accordance with these requirements.

4 A safe library for C

The safety\lib was originally part of a Voice-over-IP (VoIP) client software developed according to IEC 61508 for SIL 2. As such, its original purpose was to mainly provide a safe interface to socket and thread synchronisation functions. However, its scope has increased steadily to incorporate other problematic aspects of ANSI-C and the POSIX function set. While many of these aspects are common programming knowledge, some might not be well-known. In the following, we discuss a number of these problems to further their awareness and describe how the safety\lib tries to solve them.

4.1 Undefined behaviour

C is a language which provides lots of freedom both to the programmer and the compiler. This freedom comes at a price, as it is not guaranteed how certain source code constructs actually behave during runtime. The C90 standard[8] includes a list in Annex G, Clause G.2 of such constructs said to invoke ‘undefined behaviour’ – the most famous examples are probably division by zero, dereferencing a NULL pointer and accessing an array out-of-bounds. Undefined behaviour should not be confused with unspecified and implementation-defined behaviour which are much more benign (but can still problematic).

The danger of undefined behaviour is the unpredictability of the program’s execution. Depending on the compiler, the state of the execution environment and other arbitrary factors it might crash, fail silently or actually show no erroneous behaviour at all. Due to compiler optimisation a program might even be affected prior to executing the actual operation which invoked the undefined behaviour, as demonstrated by [9].

Examining already existing source code for undefined behaviour can be very difficult. An alternative approach is to exclude programming constructs which may lead to undefined behaviour a priori – this is the motivation behind coding guidelines. The safety\lib adopts this by adhering to the MISRA-C:2004 coding guidelines[10] which prohibits invocation of undefined behaviour in general and for specific cases such as:

- Using identifiers that do not differ in the first \(n\) characters\(^2\)
- Shifting a value by more than the number of bits it possesses minus 1 (e.g. shifting a 16-bit integer by more than 15).
- Using the value of an automatic variable before a value has been assigned.

Especially the execution of certain standard library functions with invalid arguments is a classical source of problems. For example, the inability of the string handling functions in respecting array bounds

\(^2\)The number \(n\) of significant characters is implementation defined and usually 6 and 31 for identifiers with external and internal linkage, respectively.
can lead to out-of-bounds access when the terminating NUL-character is not present in the source string. The error might be prevented by using `strncpy()`. However, if there is no terminating NUL in the first \( n \) characters to copy, the destination string will not be terminated either – only for subsequent functions to fail as in listing 1.

```c
#define BUFSIZE 19

... char buf[BUFSIZE] = {'\0'};

strncpy(buf, "undefined behaviour", BUFSIZE);

/* will likely print 19 or crash */
printf("Size of string in buffer: %u\n", strlen(buf));
```

Listing 1: Undefined behaviour invoked through standard library string handling functions

The primary method to avoid these and other classes of mistakes is to provide wrappers for standard functions which ensure that certain pre- and postconditions hold during execution. For the above example a safe version would both ensure that a given destination buffer size is not exceeded and that the resulting string is terminated in all cases. Listing 2 shows the same situation using the `safety_lib`'s `safe_strncpy()` function which incorporates these checks to avoid undefined behaviour.\(^3\)

```c
#define BUFSIZE 19

... char buf[BUFSIZE] = {'\0'};

if (1 == safe_strncpy(buf, "undefined behaviour", BUFSIZE)) {
  /* no terminating NUL detected among BUFSIZE characters */
  printf("String too long\n");
} else {
  /* will always print less equal 18 */
  printf("Size of string in buf: %u\n", strlen(buf));
}
```

Listing 2: Using `safe_strncpy()` to avoid undefined behaviour

\(^3\) A similar function already exists with `strlcpy()` on several platforms. However, `strlcpy()` is not defined by POSIX and so not universally available – for example, glibc does not implement the function.

### 4.2 Complicated Interfaces

Due to historic growth the POSIX 2001 standard\(^{[11]}\) contains a number of functions whose interfaces are rather complicated to use correctly. This especially concerns the socket API used in network programming, which suffers from the need to combine several different socket types and provide support for IPv6. Applications requiring network traffic must take care to use the correct type of socket address structure in resolving addresses and creating sockets or risk unstable network behaviour.

Listing 3 shows how to get the IP address of a remote peer connected via TCP. However, this code does not work with IPv6 which would need another address structure type to store its larger address (namely `sockaddr_in6`). Instead of limiting itself to one of the types applications should use a `sockaddr_storage` structure suitable for both IPv4 and IPv6 addresses as recommended in \([12]\). This involves a conversion to yet another type (`sockaddr`) in order for `getpeername()` to work, since IPv6 support was added after the API was defined. Similar confusing behaviour can be found for other functions dealing with sockets and address structures (e. g. the output of `getnameinfo()` is inconsistent across platforms, `inet_ntop()` returning 0 for error).

```c
int sockfd_connected = -1;
struct sockaddr_in peer_addr;
socklen_t peer_addrlen = sizeof(peer_addr);
char peer_addrstr [INET_ADDRSTRLEN + 1U];

... getpeername(sockfd_connected, (struct sockaddr *) &peer_addr, &peer_addrlen);
inet_ntop(AF_INET, &peer_addr.sin_addr, peer_addr_string, (socklen_t) (INET_ADDRSTRLEN + 1U));
printf("IP of peer: %s\n", peer_addrstr);
```

Listing 3: Getting the IP address of a TCP peer in the traditional way

To facilitate the usage of sockets in a protocol independent, portable and non-confusing manner, the `safety_lib` includes wrappers for socket handling...
functions which enable the programmer to work directly with addresses in textual and binary form for IPv4 and IPv6. For example, listing 4 demonstrates the above scenario using the safe_get_peer_address() function instead.

```c
int socketd_connected = -1;
char peer_addrstr[INET_ADDRSTRLEN + 1U];
uint16_t peer_srcport;
enum ip_version peer_ip_version;
...
/* calls to socket(), bind() and accept() omitted */
...
safe_get_peer_address(socketd_connected, peer_addrstr, (INET6_ADDRSTRLEN + 1U), &peer_srcport, &peer_ip_version, FALSE);
printf("IP of peer: %s\n", peer_addrstr);
printf("Source port: %u\n", peer_srcport);
if (IP_V6 == peer_ip_version)
{
printf("Connection uses IPv6\n");
}
Listing 4: Getting the IP address of a TCP peer and more with safe_get_peer_address()
```

4.3 Dynamic allocation

Safety standards are quite clear in prohibiting dynamic allocation/deallocation of memory during runtime\(^4\). To avoid the associated dangers such as memory leak there is no explicit dynamic allocation in the safety\_lib (although some utilised standard library or POSIX functions allocate memory temporarily, e. g. getaddrinfo()).

Dynamic allocation is often used to implement abstract data types (ADT) such as lists. The safety\_lib provides an API to create ADTs using only statically allocated memory. In this way, there is no loss of flexibility in storing data when the maximum storage consumption is known.

In case dynamic allocation is absolutely needed a simple custom memory allocator is provided which acts on a block of statically allocated storage. Both the allocation and coalescing algorithms can be easily replaced if necessary.

4.4 Thread synchronisation

Sometimes it is necessary to force multiple threads to execute in a certain order or to let them wait for a specific event to occur. The principal way to achieve this with the POSIX pthread API is to use a condition variable (condvar). A condvar basically puts the current thread to sleep until it receives a signal from another thread and is usually associated with a predicate that determines if it is necessary to wait. Furthermore, each condvar is paired with a mutex that is atomically unlocked and locked when the thread starts and ends it sleep respectively. The proper use of a condvar requires certain steps:

- Before the predicate is checked, the mutex must be locked.
- If the predicate evaluates to false, the thread blocks on the condvar and releases the mutex.
- After getting signalled and waking up, the mutex is automatically locked and the predicate must be checked again. This is necessary, because condvars are subject to 'spurious wake-ups, i. e. a thread sleeping on a condvar might return without actually receiving a signal (see \cite{13} for details on this behaviour).
- If the predicate now evaluates to true, the mutex must be unlocked and the thread continues.

To reduce the possibility of errors when implementing the above procedure, the safety\_lib provides an extended version of POSIX condvars, termed 'signal gate', with the following semantics:

- Spurious wake-ups are handled internally and never lead to undesired thread continuation.
- A thread signalling another one can assign an ID and a priority (high or low) to the signal.
- A thread waiting on a signal gate can set a mask of valid signal IDs. Only signals with a matching ID can 'unlock' the gate and allow the thread to continue.
- If one or more signals are sent to a signal gate before a thread actually waits on it, the signals are buffered in the order of their arrival and processed according to their priority (high before low). This is not possible with raw condvars as signals are not buffered in this case but simply lost.
As with condvars, a timed wait can be used which automatically unlocks when no signal was received for a certain time.

The signal gate primitive was extensively used in the design and implementation of a multi-threaded application and proved to be very useful and robust.

4.5 Signals

POSIX signals are a classical tool for asynchronous inter process communication and used explicitly or indirectly in many applications. However, signals pose a safety risk as they can occur practically any time without regard for synchronisation and require great care to process correctly (e. g. functions called in a signal handler must be re-entrant in order not to deadlock, unhandled signals may shut down the application ungracefully). Standards therefore discourage their use\(^5\) in a safety-critical software.

The abolishment of signals is especially problematic when an application should run in the background as a daemon process since most APIs (such as daemon() or start-stop-daemon) rely on signals for communication. The safety\_lib therefore provides a framework for creating and controlling daemons without signals, instead using POSIX message queues for relaying start/stop/restart commands to the background process. This allows for ignoring most signals, reducing the risk of harmful interruptions.

5 Technical Aspects

As mentioned before, the safety\_lib is designed to be portable to a majority of POSIX 2001 conformance systems. This was ensured by successfully building and testing the library on a variety of platforms which are listed in Table 2. It should be mentioned, that the networking API requires a dual-stack system to build (i. e. the network stack must support both IPv4 and IPv6). The library is optimised for 32-bit platforms but should work without problems on 64-bit systems, despite not actually using 64-bit variables internally.

<table>
<thead>
<tr>
<th>OS/Distro</th>
<th>HW Arch</th>
<th>gcc -v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian 6</td>
<td>x86</td>
<td>4.4.5</td>
</tr>
<tr>
<td>Debian 6</td>
<td>x86_64</td>
<td>4.4.5</td>
</tr>
<tr>
<td>Debian 6</td>
<td>mipsel</td>
<td>4.4.5</td>
</tr>
<tr>
<td>Debian 6</td>
<td>ppc</td>
<td>4.4.5</td>
</tr>
<tr>
<td>Ubuntu 10.04</td>
<td>x86</td>
<td>4.4.3</td>
</tr>
<tr>
<td>FreeBSD 8.2</td>
<td>x86</td>
<td>4.2.1</td>
</tr>
<tr>
<td>Debian GNU/Hurd 6</td>
<td>x86</td>
<td>4.4.6</td>
</tr>
<tr>
<td>OpenSolaris 2009.06</td>
<td>x86</td>
<td>3.4.3</td>
</tr>
<tr>
<td>OpenIndiana build 151a</td>
<td>x86</td>
<td>3.4.3</td>
</tr>
</tbody>
</table>

TABLE 2: SW/HW-Platforms on which the safety\_lib was tested.

To fulfil the requirement for modularity, the library code respects the following code metric limits:

- Maximum LOC per function: 200
- Maximum cyclomatic complexity per function: 20
- Maximum number of function parameters: 7 (with 1 exception)

Functions that work with persistent objects (e. g. lists, signal gates) are fully thread-safe by using pthread mutexes. All remaining functions are re-entrant.

6 Evidence provided by the safety\_lib

As important as the actual code is the evidence for fulfilling the safety requirements mentioned at the beginning. The subsequent list gives the specific proofs that are provided by the safety\_lib:

- Coding guidelines/static analysis – MISRA:2004 requires two pieces of evidence to proof the conformance of code to its rules: A ‘compliance matrix’ explaining how each rule was actually enforced and a set of ‘deviation requests’ which justify any deviation from a rule. Both of these objects were created for the safety\_lib. As for the actual checking, the library code was analysed with the commercial static analysis tool FlexeLint\(^{[14]}\) supported by manual code reviews targeting rules that can not be assessed by the tool. The reports of the analyser and the reviews are also available.
• **Coding style guide** – The code follows a self-defined style guide loosely based on the Linux kernel coding style[15]. No suitable checking tool was available, so manual reviews and their reports act as substitute.

• **Modularity** – As mentioned before, metric limits were enforced on the source code which were checked with cccc[16]. The metric reports generated by the tool provide the necessary evidence.

• **Defined interfaces** – All functions with external linkage are documented inline in a consistent manner detailing the function purpose, meaning of parameters and return values. Doxygen[9] markup was used to automatically generate the documentation from source.

• **Testing** – The library code was tested by automated unit tests using a modified version of the CUnit[18] framework to achieve a minimum statement/branch coverage of 90%/80% per unit and 93%/84% on average for the whole library. The code coverage was measured with gcov and graphical reports were generated with lcov[19].

7 Alternatives

Implementing safe versions of standard functions and programming idioms is an established practice. Although there already exist solutions in a similar vein to the safety_lib, they are insufficient for direct adoption into a safety-critical software. This section discusses some of the freely available alternatives and their deficiencies.

• **Safe C Library** – This library implements alternative versions of the standard library’s string handling and memory allocation functions defined by ISO/IEC TR 24731[20]. These extensions mainly add bounds checking and have the benefit of standardisation and completeness in regard to the standard library but nothing else. Furthermore, no additional safety evidence is available.

• **Safe C String Library** – A library providing safe string manipulation using a custom string type with length information. The drawbacks of this approach are the increased effort in porting legacy applications and the usage of dynamic allocation for storing strings. Evidence is missing as well.

• **Cyclone** – As a rather different approach, Cyclone[21] is a dialect of C which adds safety checks during compilation and runtime, preventing certain types of errors. Despite the great potential, it is unfortunately no longer maintained and would require justification for the compiler which is most likely not proven-in-use.

To summarise, the alternatives lack either the necessary evidence, do not adhere to the requirements of safety standards or have a too narrow scope.

8 Future Work

While the decision that the safety_lib should be released under a FLOSS license was a quick one, the details still have to be clarified. The following list includes some of the decisions to be made and the preconditions that have to be established before we can actually release our code – the intention of this paper is to get feedback from potential users that may help us to make those decisions which will greatly influence the future of the safety_lib.

**Deciding on a FLOSS License** – It can be assumed that most companies would not be interested in linking to a library that “forces” them to release their application under the same license. Because of this, we are currently tending towards LGPL [22] or a similar model, as this allows the usage of the library in conjunction with a proprietary software, while ensuring that improvements to the safety_lib itself are released as FLOSS. The intention is to prevent ‘grab and run’ in the interest of everyone who really wants to use and participate in the development of the safety_lib, while still allowing contributing parties to use the safety_lib without having to license their whole application under a FLOSS license.

**Define a Development Life Cycle** – A general requirement of the various safety standards not yet discussed is the need for a well defined development life cycle. To satisfy this demand in a way that embraces the needs of software development in the safety domain as well as the needs for a community driven open-source development we will need to define such a life cycle in detail prior to release.

**Evidence Management** – The basis for a successful management of evidence that can be used during the certification process is to have a
technical framework that supports the collection and management of evidence and semi-automatically produces the documents that can be provided to the certification authorities. In our current development, we are using Codestriker[23] to document code reviews and Bugzilla[24] to report bugs. Both of those tools store the collected data in SQL databases from which a review report and a report of known problems can be pulled with small effort.

**Legal Issues** – Just as important as the management of evidence is the question of legal problems with the collected data. This includes copyright and licensing issues as well an estimation of the impact on a possible certification process against a standard. Basically, this means that the evidence itself will need to be released under an open license as well. The first that comes in mind here is the FDL [25] but CC [26] might be an option as well.

### 8.1 Community Development

To allow efficient community driven development, the safety\_lib has already been pulled out of the subversion repository of the project it evolved from and has been moved into a separate git repository. Care has been taken to preserve traceability in form of the change history back to the very beginning. As soon as the safety\_lib is released under a FLOSS license this repository will be made accessible to the public.

As described above it is crucial to decide on a development life-cycle for the safety\_lib. This will also concern the way how patches find their way into the official repository and contributions to the evidence databases can be committed. As these kind of things are new to us, and – as far as we know – unique in the safety world we do not want to rush into things but to define a strategy - that not just pleases us, but makes the safety\_lib interesting for everyone developing software for safety critical systems on POSIX compliant systems - up front.

### 8.2 Joint Evidence Collection

The evidence collected so far, as described in previous sections, only builds the basis of the evidence that can be provided in a joint development. We also would like to emphasise again that we do not think that the savings in implementation costs are the foremost benefit of an open-source safety library. Rather, the real advantage will lie in a thorough set of evidence at a level which is very hard to achieve within one single organisation. In a first step, this includes more code reviews by developers from different organisations and from different industries towards a higher chance of discovering subtle bugs – especially by running the code on a variety of hardware and software platforms in various applications.

In the future, pure testing and code reviews will be less and less effective to master growth in code size and complexity and the usage of formal methods for code verification will become more and more important for this case – even for systems with a low safety integrity level. Unfortunately, most of these formal method techniques are time intensive tasks demanding expert knowledge. Here, the big advantage of joint development manifests itself by the chance for those developers new to the formal analysis to learn from those who have experience, and to get their first steps in formal verification checked by experts.

### 9 Conclusions

Safety standards impose a lot of constraints on the implementation of software deemed for certification. A set common to most important standards can be defined from these constraints which demands evidence that all requirements are fulfilled by the code. The acquisition of this evidence is time-consuming and often not even possible – especially when depending on closed-source software.

Inside this paper we proposed the usage of an open-source library developed under the normative constraints as a way to raise the safety of an application and satisfy the need for evidence. A proof-of-concept library, the safety\_lib, is presented as a foundation for a generic safety library jointly developed by the FLOSS community.

The next steps depend on the feedback of the community. If there is sufficient interest, a development life cycle needs to be defined which enables contributions both to the code and the evidence base without violating safety constraints.

### 10 Acknowledgements

The safety\_lib was developed by the 'Stadt Wien Kompetenztteam für Safety Network Engineering (SNET)' which is supported by MA27 – EU-Strategie und Wirtschaftsentwicklung in the course of the funding programme 'Stiftungsprofessuren und Kompetenztteams für die Wiener Fachhochschul-Ausbildungen'.
We dedicate this paper and the safety lib to the late DI Herbert Haas who lead the ‘Stadt Wien Kompetenzteam für Safety Network Engineering (SNET)’. He originally proposed the idea of a software library for safety-critical applications and his encouragement was invaluable to us during the development.

References


[17] www.doxygen.org


[26] Creative Commons License, http://creativecommons.org
"Open Proof" for Railway Safety Software

Klaus-Rüdiger Hase
Deutsche Bahn AG
Richelstrasse 3, 80634 München, Germany
Klaus-Ruediger.Hase{at}DeutscheBahn{dot}com

Abstract
"Open Proof (OP) is a new approach for safety and security critical systems and a further development of the Open Source Software (OSS) movement, not just applying OSS licensing concepts to the final software products itself, but also to the entire life cycle and all software components involved, including tools, documentation for specification, verification, implementation, maintenance and in particular including safety case documents. A potential field of applying OP could be the European Train Control System (ETCS) the new signaling and Automatic Train Protection (ATP) system to replace some 20 national legacy signaling systems in all over the European Union. The OP approach might help manufacturers, train operators, infrastructure managers as well as safety authorities alike to eventually reach the ambitious goal of an unified fully interoperable and still affordable European Train Control and Signaling System, facilitating fast and reliable cross-border rail traffic at state of the art safety and security levels.

Keywords: ATC, ATP, Critical Software, Embedded Control, ETCS, EUPL, FLOSS, Open Proof, openETCS, Train Control, Standardization.

1 Introduction

The European Train Control System (ETCS, [1]) is intended to replace several national legacy signaling and train control systems all across Europe. The system consists of facilities in infrastructure and on-board units (OBU). Especially for the ETCS on-board equipment the degree of functional complexity to be implemented is expected to be significantly higher than in conventional systems. In terms of technology, this is mostly done by software in so-called embedded control system implementations. While electronic hardware is getting continuously cheaper, the high complexity of the safety critical software has caused significant cost increases for development, homologation and maintenance of this technology. This has raised questions for many railway operators with respect to the economy of ETCS in general.

The key element for improving that situation seems to be a greater degree of standardization in particular for the ETCS onboard equipment on various levels: Hardware, software, methods and tools. Standardization by applying open source licensing concepts will be the focus of this paper.

1.1 From National Diversity to European Standard
Looking back into history of signaling and automatic train protection (ATP) for mainline railways systems, in the past 40 years a major change in technology has taken place. In the early days of ATP almost all functions were implemented in hardware, starting with pure mechanical systems, advancing to electromechanical components and later on using solid state electronics, like gates, amplifiers, and
other discrete components. Software was not an issue then. Beginning in the late 1970 years an increasing number of functions were shifted into software, executed by so called micro computers. Today the actual functions of such devices are almost entirely determined by software. The dramatic performance increase of microcomputers in the past 30 years on the one hand and rising demand for more functionality on the other hand, has caused a significant increase in complexity of those embedded control systems - how such devices are usually called. Furthermore, the development from purely monitoring safety protection systems, like the German INDUSI (later called PZB: Punktfrmige Zug-Beenieflussung) or similar systems in other European countries, which only monitor speed at certain critical points and eventually stop the train, if the driver has missed a halt signal or has exceeded a safe speed level, to a more or less (semi) Automatic Train Control (ATC) systems like the German continuous train control system, called LZB (Linien-Zug-Beenieflussung), which has increasingly shifted safety responsibility from the infrastructure into the vehicle control units. Displaying signal commands inside the vehicle on certain computer screens, so called cab signaling, has resulted in greater independence from adverse weather conditions.

**FIGURE 1:** Europes challenge is to substitute more than 20 signaling and ATP systems by just one single system, ETCS, in order to provide border crossing interstate rail transit in all over the European Union.

In all over Europe there are more than 20 different mostly not compatible signaling and train protection systems in use (figure 1). For internationally operating high speed passenger trains or cargo locomotives up to 7 different sets of equipment have been installed, just to operate in three or four countries. Since each of those systems have their own antennas to sense signals coming from the way-side and their own data processing units and display interfaces, space limitations are making it simply impossible to equip a locomotive for operation in all EU railway networks, not to mention prohibitive cost figures for such equipment. Furthermore, some of the systems are in use for more than 70 years and may not meet todays expected safety level. Some are reaching their useful end of life causing obsolescence problems.

For a unified European rail system it is very costly to maintain this diversity of signaling systems forever and therefore the European Commission has set new rules by so called Technical Specifications for Interoperability (TSI) with the goal to implement a unified European Train Control System, which is part of the European Rail Traffic Management System (ERTMS), consisting of ETCS, GSM-R, a cab radio system based on the GSM public standard enhanced by certain rail specific extensions and the European Traffic Management Layer (ETML). Legacy ATP or so called Class B systems are supposed to be phased out within the next decades.

1.2 ETCS: A new Challenge for Europes Railways

Before launching the ETCS program, national operational rules for the railway operation were very closely linked with the technical design of the signal and train protection systems. That is going to change radically with ETCS. One single technology has to serve several different sets of operational rules and even safety philosophies.

The experience of Deutsche Bahn AG after German reunification has made very clear that it will take several years or even decades to harmonize operational rules in all over Europe. Even under nearly ideal conditions (one language, one national safety board and even within one single organization) it was a slow and laborious process to convert different rules and regulations back into one set of unified operational rules. After 40-years of separation into two independent railway organizations (Deutsche Reichsbahn in the east and Deutsche Bundesbahn, west), it took almost 15 years for Deutsche Bahn AG to get back to one single unified signaling handbook for the entire infrastructure of what is today DB Netz AG.

Therefore, it seem unrealistic to assume that there will be one set of operational rules for all ETCS lines in all over Europe any time soon (Which does not mean that these efforts should not be started as soon as possible, but without raising far too high expectations about when this will be accomplished.). That means, in order to achieve interoperability by using a single technical solution: This new system has to cope with various operational regimes for the foreseeable future. Beside this, for more than a
decade there will be hundreds of transition points between track sections equipped with ETCS and sections with one of several different legacy systems. This will cause an additional increase of functional complexity for onboard devices.

1.3 Technology is not the Limiting Factor

With state of the art microcomputer technology, from a technological point of view, this degree of complexity will most likely not cause any performance problems since the enormous increase in performance of microcomputer technology in recent years can provide more than sufficient computing power and storage capacity at an acceptable cost level; to master complex algorithms and a huge amount of data.

The real limiting factor here is the human brain power. In the end it is the human mind, which has to specify these functions consistently and completely, then provide for correct designs, implement them and ultimately make sure that the system is fit for its purpose and can prove its safety and security. The tremendous increase in complexity, absorbing large numbers of engineering hours is one reason why we are observing cost figures for R&D, testing and homologation of the software components in various ETCS projects that have surpassed all other cost positions for hardware design, manufacturing and installation. This has caused a substantial cost increase for the new onboard equipment compared with legacy systems of similar functionality and performance.

Normally we would expect from any new technology a much better price to performance ratio than for the legacy technology to be replaced. Due to the fact, that this is obviously not the case for ETCS, makes it less attractive for those countries and infrastructure managers, who have already implemented a reliably performing and sufficiently safe signaling and train control system. In addition there is no improvement expected for ETCS with respect to performance and safety compared with service proven systems like LZB and PZB. In order to reach the goal of EU-wide interoperability soon, the EU Commission has implemented legal measures, regulating member states policies for infrastructure financing and vehicle homologation. While in the long run, ETCS can lower the cost for infrastructure operators, especially for Level 2 implementations making conventional line signals obsolete, the burden of cost increase stays with the vehicle owners and operators.

Therefore it became an important issue for vehicle operators to identify potential cost drivers and options for cost reduction measures, so as not to endanger the wellintentioned general goal of unrestricted interoperability.

2 Software in ETCS Vehicle Equipment

As discussed above, state of the art technology requires for almost all safety critical as well as non-safety related functions to be implemented in software. The end-user will normally receive this software not as a separate package, but integrated in his embedded control device. Therefore software is usually only provided in a format directly executable by the built-in microprocessor, a pattern of ones and zeros, but therefore not well suited for humans to understand the algorithm. The original source code, a comprehensible documentation format of this software, which is used to generate the executable code by a compiler and all needed software maintenance tools are usually not made available to the users. Manufacturers are doing this, because they believe that they can protect their high R&D investment this way.

2.1 Impact of Closed Source Software

However concealment of the software source code documentation has increasingly been considered as problematic, not only for economical reasons for the users, but more and more for safety and security reasons as well. Economically it is unsatisfactory for the operators to remain completely dependent from the original equipment manufacturer (OEM), no matter whether software defects have to be fixed or functions to be adapted due to changing operational requirements. For all services linked to these embedded control systems there is no competitive market, since bundling of non-standard electronic hardware together with closed source or proprietary software makes it practically and legally impossible for third parties to provide such service. This keeps prices at high levels. While malfunctions and vulnerability of software products, allowing malware (malicious software: as there are viruses, trojans, worms etc.) to harm the system, can be considered as quality deficiencies, which can practically not be discovered in proprietary software by users or independent third parties, whereas the question of the vendor lock-in due to contractual restrictions and limiting license agreements is generally foreseeable, but due to gen-
eral accepted practices in this particular market, hardly to be influenced by individual customers (e.g. railway operators). Especially security vulnerability of software must be considered as a specific characteristic of proprietary or closed source software. So-called End User License Agreements (EULA) do usually not allow end-users to analyze copy or redistribute the software freely and legally. Even analysis and improvement of the software for the users own purposes is almost generally prohibited in most EULAs. While on the one hand customers who are playing by the rules are barred from analyzing and finding potential security gaps or hazardous software parts and therefore not being able to contribute to software improvements, even not for obvious defects, however the same legal restrictions on the other hand do not prevent bad guys from disassembling (a method of reverse-engineering) and analyzing the code by using freely available tools, in order to search for security gaps and occasionally (or better: mostly) being successful in finding unauthorized access points or so-called backdoors. Intentionally implemented backdoors by irregularly working programmers or just due to lax quality assurance enforcement or simply by mistake are causing serious threats in all software projects. In most cases intentionally implemented backdoors are hard to find with conventional review methods and testing procedures. In a typical proprietary R&D environment only limited resources are allocated in commercial projects for this type of security checks and therefore stay most likely undiscovered. That backdoors cannot be considered as a minor issue, has been discussed in various papers [2, 3, 4, 5] and has already been identified as a serious threat by the EU Parliament, which has initiated resolution A5-0264/2001 in the aftermath of the Echelon interception scandal, resulting in following recommendations [6]:

... Measures to encourage self-protection by citizens and firms:

29. Urges the Commission and Member States to devise appropriate measures to promote ... and ... above all to support projects aimed at developing user-friendly open-source encryption software;

30. Calls on the Commission and Member States to promote software projects whose source text is made public (open-source software), as this is the only way of guaranteeing that no backdoors are built into programmes;

31. Calls on the Commission to lay down a standard for the level of security of e-mail software packages, placing those packages whose source code has not been made public in the least reliable category; ...

This resolution was mainly targeting electronic communication with private or business related content, which most likely will not hurt people or endanger their lives. However a recent attack by the so called STUXNET worm [7], a new type of highly sophisticated and extremely aggressive malware, which in particular was targeting industrial process control systems via its tools chain, even in safety critical applications (chemical and nuclear facilities). Systems, which are very similar in terms of architecture and software design standards with signaling and interlocking control systems. This incident has demonstrated that we have to consider such impact in railway control and command systems as well, commercially and technically.

2.2 Software Quality Issues in ETCS Projects

Despite a relatively short track record of ETCS in revenue service we had already received reports on accidents caused by software defects, like the well documented derailment of cargo train No. 43647 on 16 October 2007 at the Ltschberg base line in Switzerland [8]. German Railways has been spared so far from software errors with severe consequences, possibly due to a relatively conservative migration strategy. During the past 40 years, software was only introduced very slowly in small incremental steps into safety-critical signaling and train protection systems and carefully monitored over years of operation, before rolled out in larger quantities. Software was more or less replacing hard-wired circuits with relatively low complexity based on well serviceproven functional requirement specifications over a period of four decades. With ETCS however, a relatively large step will be taken: Virtually all new vehicles have to be equipped with ETCS from 2015 on, enforced by European legal requirements, despite the fact that no long-term experience has been made. The ongoing development of the functional ETCS specification as well as project specific adaptations to national or line-specific conditions has resulted in numerous different versions of ETCS implementations not fully interoperable. Up to now, there is still no single ETCS onboard equipment on the market that could be used on all lines in Europe, which are said to be equipped with ETCS. That means that the big goal of unrestricted interoperability would have been missed, at least until 2010. The next major release of the System Requirements Specification (SRS 3.0.0), also called "baseline 3", is expected to elim-
ination those shortcomings. Baseline 3 has another important feature: Other than all previous SRS versions, which have been published under the copyright of UNISIG, an association of major European signaling manufacturers, SRS 3.0.0 in the opposite has been published as an official document by the European Railway Agency (ERA) a governmental organization implemented by the European Commission. This gives the SRS a status of a public domain document. That means, everyone in Europe is legally entitled to use that information in order to build ETCS compliant equipment.

2.3 Quality Deficiencies in Software Products

Everyone who has ever used software products knows that almost all software has errors and no respectable software company claims, that their software is totally free of defects. There are various opportunities to make mistakes during the life cycle of software production and maintenance: Starting with System Analysis, System Requirement Specification, Functional Requirement Specification, etc., down to the software code generation, integration, commissioning, operation and maintenance phases. A NASA Study on Flight Software Complexity [12] shows contribution to bug counts, which can be expected in different steps of software production (figure 2).

![FIGURE 2: Propagation of residual defects (bugs) as a result of defect insertion and defect removal rates during several stages of the software production process, according to a NASA research on high assurance software for flight control units for each 1000 lines of code (TLOC) [12].](image)

Figure 3 characterizes the actual situation in the European signaling industry with several equipment manufacturers working in parallel, using the same specification document in natural language precision giving room for interpretation, combined with different ways and traditions of making mistakes, resulting in a low degree of standardization, even for those components, which cannot be used for product differentiation (core functionality according to UNISIG subset 026 [1]). Since all or at least most of the documents are created by humans, there is always the human factor involved, causing ambiguities and therefore divergent results. Herbert Klaeren refers to reports in his lecture [9], which have found an average of 25 errors per 1000 Lines Of programming Code (TLOC) for newly produced software. The book Code Complete by Steve McConnell has a brief section about errors to be expected. He basically says that there is a wide range [10]:

(a) Industry Average: "about 15 - 50 errors per 1000 lines of delivered code." He further says this is usually representative of code that has some level of structured programming behind it, but probably includes a mix of coding techniques.

(b) Microsoft Applications: "about 10 - 20 defects per 1000 lines of code during inhouse testing, and 0.5 defect per TLOC in released products [10]." He attributes this to a combination of code-reading techniques and independent testing.

(c) "Harlan Mills pioneered a so called 'clean room development', a technique that has been able to achieve rates as low as 3 defects per 1000 lines of code during in-house testing and 0.1 defect per 1000 lines of code in released product (Cobb and Mills 1990 [11]). A few projects - for example, the space-shuttle software - have achieved a level of 0 defects in 500,000 lines of code using a system of formal development methods, peer reviews, and statistical testing."

![FIGURE 3: Divergent interpretation of a common public domain ETCS System Requirement Specification (SRS) document, due to the human factor by all parties involved, causing different software solutions with deviant behavior of products from different manufacturers, which result in interoperability deficiencies and costly subsequent improvement activities.](image)
However the U.S. space shuttle software program came at a cost level of about U.S. $ 1,000 per line of code (3 million LOC – 3 billion U.S. $ [9], cost basis 1978), not typical for the railway sector, which is more in a range between 30 per LOC for non-safety applications and up to 100 for SIL 3-4 quality (SIL: Safety Integrity Level) levels.

2.4 Life Cycle of Complex Software

While on the one hand, electronic components are becoming increasingly powerful, yet lower in cost, on the other hand, cost levels of complex software products are increasingly rising not only because the amount of code lines need tremendous men power, but for those lines of code a proof of correctness, or also called safety case, has to be delivered in order to reach approval for operation in revenue service. Some manufacturers have already reported software volumes of over 250 TLOC for the ETCS core functionality defined by ETCS SRS subset 026 [1]. It is very difficult to receive reliable statistics about errors on safety related soft- ware products, because almost all software manufacturers hide their source code using proprietary license agreements. However we can assume that software in other mission critical systems, like communication servers, may have the same characteristics with respect to bug counts. One of those rare examples published, was taken from an AT&T communication software project, which from its size is in the same order of magnitude as todays ETCS onboard software packages (figure 4) [9],[13]. One particular characteristic in figure 4 is quite obvious: The size of the software is continuously growing from version to version, despite the fact that this software was always serving the same purpose. Starting with a relatively high bug count of almost 1000 bugs in less than 150 TLOC, the software matures after several release changes and is reaching a residual bug density, which is less than a tenth of the initial bug density. During its early life the absolute number of bugs is oscillating and stabilizes in its more mature life period. At a later phase of the life cycle the absolute number of bugs is slightly growing despite a decreasing bug density. The late life-cycle bug density is mainly determined by the effectiveness and quality measures taken on the one hand and the number of functional changes and adaptations built in since last release on the other hand. The actual number of bugs is often unknown and can only subsequently been determined.

Even though that extensive testing has been and still is proposed as an effective method for detecting errors, it has become evident, that by testing alone the correctness of software cannot be proven, because tests can only detect those errors for which the test engineer is looking for [14]. This means ultimately that there is no way to base a safety case on testing alone, because the goal is not to find errors, but to prove the absence of errors. One of the great pioneers of software engineering, Edsgar W. Dijkstra, has put it into the following words [15]:

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

We have even to admit that at the current state of software technology, there is no generally accepted single method to prove the correctness of software that means, there is no way to prove the absence of bugs, at least not for software in a range of 100 TLOC or more. The only promising strategy for minimizing errors is:

- The development of functional and design specifications (architecture) has to be given top priority and needs adequate resources,
- The safety part of the software has to be kept as small as possible, and
- The software life-cycle has to undergo a broad as possible, manifold, and continuous review process.

Successful software projects require for the first point, the specification, at least between 20% and 40% [12] of the total development cost, depending on the size of the final software package. Trying to
save at this point will almost certainly result in inflated costs during later project phases (see figure 5).

**FIGURE 5:** Fraction of the overall project budgets spent on specification (architecture) versus fraction of budget spend on rework + architecture, which defines a so-called sweet spot where it reaches its minimum [12]. However, this cost function does not take any potential damages into account, which might result from fatalities caused by software bugs.

Formal modeling methods and close communication with the end-user may be helpful in this stage, especially when operational scenarios can be modeled formally as well in order to verify and validate the design. Specification, modeling and reviews by closely involving the customer may even require several cycles in order to come to a satisfactory result.

**FIGURE 6:** Graph taken from a NASA Study on Flight Software Complexity [12] suggesting a reasonable limit for software size, determine a level, which results in certainty of failure beyond this size (NCSL: Non-Commentary Source Lines = LOC as used in this paper).

Large railway operators in the opposite may have several hundreds of trains, representing the same level of material value (but carrying hundreds of passengers) operating at the same time. Assuming that a mission critical failure in software of a particular size may show up once in 1000 years, would mean for a space mission duration of 1 year, a probability for a mission failure of about 0.1% due to software (equal distribution assumed). For the railway operator however, who operates 1000 trains at the same time, having the same size and quality of software on board may cause a mission failure event about once a year, making drastically clear that code size matters. While the third point is very difficult to get implemented within conventional closed source software projects, simply because highly qualified review resources are always very limited in commercial projects. Therefore errors are often diagnosed at a late stage in the process. Their removal is expensive and time consuming. That is why big software projects often fail due to schedule overruns and cost levels out of control and often even been abandoned altogether. Never the less, continuous further development and consistent use of quality assurance measures can result in a remarkable process of maturation of software products, which is demonstrated by the fact that in our example in figure 4 the bug density has been reduced by more than an order of magnitude (initially above 6 bugs/TLOC down to below 0.5 bugs/TLOC). On the other hand, in a later stage of the life cycle, due to the continuous growth of the number of code lines, which seem to go faster than the reduction of the bug density, a slight increase of the total number of bugs, can be observed. Given a certain methodology and set of quality assurance measures on the one hand and a number of change requests to be implemented per release, then this will result in a certain number of bugs that remains in the software. Many of those bugs stay unrecognized forever. However some are also known errors, but their elimination is either not possible for some reason or can only be repaired at an unreasonably high level of cost. The revelation of the unknown bugs can take several thousand unit operation years (number of units times number of years of operation) and must be considered as a random process. That means for the operator, that even after many years of flawless operation, unpleasant surprises have to be expected at any time. In Europe, in a not too distant future up to 50,000 trains will operate with ETCS, carrying millions of passengers daily, plus unnumbered trains with hazardous material. Then the idea is rather scary that in any of those European Vital Computers (EVC), the core element of the ETCS vehicle equipment, between 100 and 1000 undetected errors are most likely left over, even after successfully passing safety case assessments and after required authorization has been granted. Even if we would assume, that only one out of 100 bugs might eventually cause a hazard [12], that still means 1 to 10 mission critical...
defects per unit. Furthermore, there will be several different manufacturer-specific variants of fault patterns under way.

2.5 New Technologies Have to Have At Least Same Level of Safety

According to German law (and equally most other EU states) defined in the EBO (Eisenbahn Bau- und Betriebsordnung: German railway building and operation regulations) any new technology has to maintain at least the same safety level as provided by the preceding technology [16]. Assuming that the more complex ETCS technology requires about ten times more software code than legacy technology like LZB and PZB as an average and given the fact, that PZB and LZB have already reached a very mature stage of the software integrated after almost 3 decades of continuous development, then it seems very unlikely, that the at least same level of safety can be proven by using the same technical rules and design practices for a relatively immature ETCS technology. In addition, due to less service experience the criticality of deviations from expected reaction patterns are difficult to assess. This raises the question whether proprietary software combined with a business model that sells the software together with the hardware device, and then - as up to now - will be operated largely without any defined maintenance strategy, might be inadequate for such a gigantic European project. A project eventually replacing all legacy, but well service proven signaling and train protection systems with one single unified, but less service proven technology, especially when independent verification and system validation is only provided at rare occasions by a very limited number of experts .... or ... then again after a critical incident has taken place only? Instead, a broad and continues peer review scheme with full transparency in all stages of the life cycle of ETCS on-board software products would be highly recommended. In particular during the critical specification, verification and validation phases, following the so called Linus’s Law, according to Eric S. Raymond, which states that:

\textit{given enough eyeballs, all bugs are shallow.}

More formally:

\textit{Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone.}

The rule was formulated and named by Eric S. Raymond in his essay The Cathedral and the Bazaar [17]. Presenting the code to multiple developers with the purpose of reaching consensus about its acceptance is a simple form of the software reviewing process. Researchers and practitioners have repeatedly shown the effectiveness of the reviewing process in finding bugs and security issues [18].

2.6 Changing Business Model for Software: From Sales to Service

Such problems can be solved by changing the business model. Looking at software as a service rather than a commodity or product (in its traditional definition) is justified by the fact that software is growing continuously in size (but not necessarily increasing the value to the user) as shown in figure 4. Over a life-cycle of 17 releases, the total size of that software grew by more than 300%. Furthermore, about 40% of the modules of the first release had to be fixed or rewritten, due to one or more bugs per module. That means only 60% of the original modules were reusable for second or later releases. It is fair to assume that half of the original 60% virtually bug free code had to be adapted or otherwise modified to use it for functional enhancements. This results in not more than 30% of remaining code, which is about 50 TLOC of the original code having a chance to survive unchanged up to release No. 17. Biggerstaff [19] and Rix [20] suggest that these assumptions might even be too optimistic, as long as no specific measures have been taken in order to support reusability of code. It can be assumed that a potential sales price of all versions would be at the same level, since all versions serve in principle the same functions. That means during its life cycle only 10% (50 TLOC out of about 500 TLOC) of the final code was left unchanged from the first version in this particular example. In other words: 90% of the work (code lines) has been added by software maintenance and continuous further development efforts over the observed life cycle, which can be considered as servicing the software. In order to make this a viable business, users and software producers have to contract so called service agreements for a certain period of time. This kind of business model can be a win-win for both, users and manufacturers alike. Manufacturers are generating continuous cash flow, allowing them to maintain a team of experts over an extended period of time, dedicated to continuously improving the software. Users in exchange are having guaranteed access to a qualified technical support team ensuring fast response in a case of critical software failures. Proprietary software makes it mostly impossible for the user to switch software service providers later on, but leave users in a vendor lock-in situation with no competition on the software
service market. Competition however is the most effective driver for quality improvement and cost efficiency.

Considering commercial, technical, safety and security aspects, the risks associated with complex closed source software should be reason enough for the railway operators to consider alternatives, in particular when a large economic body, like the European Union, defines a new technological standard. Watts Humphrey, a fellow of the Software Engineering Institute and a Recipient of the 2003 National Medal of Technology (US), has put the general problem of growing software complexity in these words [21]:

While technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years. It is also why, unless you do something, the situation won't improve by itself. In fact, current trends suggest that your future products will use more software and be more complex than those of to-day. This means that more of your people will work on software and that their work will be harder to track and more difficult to manage. Unless you make some changes in the way your software work is done, your current problems will likely get much worse.

3 Proposal: Free / Libre Open Source Software for ETCS

A promising solution for the previously described difficulties could be given by providing an Open Source ETCS onboard system, making the embedded software source code and relevant documentation open to the entire railway sector. Open Source Software, Free Software or Libre Software more often called Free/Libre Open Source Software short: FLOSS [22], is software that:

- Can be used for any purpose,
- Can be studied by analyzing the source code,
- Can be improved and modified and
- Can be distributed with or without modifications.

This basic definition of FLOSS is identical to the Four Freedoms, with which the Free Software Foundation (FSF, USA, [23]) has defined free software and is in line with the open source definition formulated by the Open Source Initiative (OSI) [24].

3.1 Public License for an European Project

A potential candidate for a license agreement text could be the most widely used General Public License (GPL) or occasionally called GNU Public License, which has been published by the Free Software Foundation [23]. Because this license text (and several similar license texts as well) is based on the Anglo-American legal system. In Europe applicability and enforceability of certain provisions of the GPL are considered as critical by many legal experts. The European Union has recognized this problem some time ago and has issued the European Union Public License text [25], which not only is available in 22 official EU languages, but is adapted to the European legal systems, so that it meets essential requirements for copyright and legal liability issues. The EU Commission recommends and uses this particular License for its own European eGovernment Services project (iDABC [26]). A key feature of the aforementioned license types is the so-called strong ”Copy Left” [27]. The Copy-Left requires a user who modifies, extends or improves the software and distributes it for commercial or non-profit purposes, to make also the source code of the modified version available to the community under the same or at least equivalent license conditions, which has applied to the original software. That means everybody will get access to all improvements and further developments of the software in the future. The distribution in principle has to be done free of charge, however add-on services for a fee are permissible. That means for embedded control systems, that software-hardware integration work, vehicle integration, homologation and authorization costs can be charged to the customer as well as service level agreements for a fee are allowed within the EUPL. By applying such license concept to the core functionality of the ETCS vehicle function as defined and already published in UNISIG subset 026 of the SRS v3.0.0 [1] all equipment manufacturers as well as end-users would be free to use this ETCS software partly or as a whole in their own hardware products or own vehicles. Due to the fact that a software package of substantial value would be then available to all parties, there would be not much incentive any more for newcomers to start their own ETCS software development project from scratch, but would more likely participate in the OSS project and utilize the effect of cost sharing. Also established manufacturers, who already may have a product on their own, might consider sharing in for all further add-on functions by trying to provide an interface to the OSS software modules with their own existing software. This
will result in some kind of an informal or even formally set up consortium of co-developing firms and a so called open source eco-system around this core is most likely to evolve. This has been demonstrated by many similar FLOSS projects. The effect of cooperation of otherwise in competition operating firms, based on a common standard core product, is often called co-competition. In analogy with other similar open projects the name openETCS has been suggested for such a project. Occasionally expressed concern that such a model would squander costly acquired intellectual property of the manufacturers to competitors does not really hit the point, because on the one hand the essential functional knowledge, which is basically concentrated in the specification, has already been published by UNISIG and ERA within the SRS and cannot be used as unique selling point. On the other hand implementation know-how for specific hardware architecture and vehicle integration as well as service knowledge will not be affected and has the potential to become part of the core business for the industry. In addition, for the pioneering manufacturer open up his own software could not be better investment money, if this software becomes part of an industrial standard, which is very likely (if others are not quickly following this move) as demonstrated several times in the software industry. Not only that, but since safety related software products are closely related to the design process, tools and quality assurance measures, the pioneering OSS supplier would automatically make his way of designing software to an industrial standard as well (process standardization). Late followers had simply to accept those procedures and may end up with more or less higher switching costs, giving the pioneer a head start. Even in the case that one or two competitors would do the same thing quickly, those companies could form a consortium sharing their R&D cost and utilizing the effect of quality improving feedback from third parties and therefore improving their competitive position compared to those firms sticking with a proprietary product concept. The UNUMERIT study on FLOSS [22] has shown that integration as well as service proven products would be simply by selecting one of the already existing and (as far as possible) service proven products from the market and put it under an appropriate open source license. There are numerous examples from the IT sector, such as the software development tool Eclipse, the successor to IBMs Visual Age for Java 4.0, source code was released in 2001 [28], the Internet browser Mozilla FireFox (former: Netscape Navigator), and office communication software Open Office (former: StarOffice) and many more.

3.3 Tools and Documents Need to be Included

In the long term it will not be enough only to make the software in the on-board equipment open. Tools for specification, modeling and simulation as well as software development, testing and documentation are also essential for securing quality and lowering life cycle cost. To meet the request for more competition in the after sales software service business and avoiding vendor lock-in effects, requires third parties to be in a position to maintain software, prepare safety case documents, and get the modified software authorized again without depending on proprietary information. A request no one would seriously deny for other safety critical elements e.g. in the mechanical parts section of a railway vehicle, like loadbearing car body parts or wheel discs. The past has shown that software tools are becoming obsolete quite often due to new releases, changing of operating systems, or tool suppliers simply going out of business, leaving customers alone with little or no support for their proprietary products. Railway vehicles are often in revenue service for more than 40 years and electronic equipment is expected to be serviced for at least 20 years and tools need to be up to the required technical level for the whole period. The aircraft industry.

3.2 ETCS Vehicle On-Board Units with openETCS

Software that comes with a FLOSS license and a Copy-Left, represent some kind of a gift with a commitment, namely as such that the donor has almost a claim to receive any improvements made and further distributed by the recipient. That means all the technical improvements, which have been based on collective experiences of other users/developers and integrated into product improvements need to be distributed so that even the original investor gets the benefits. By recalling the fact that during the life cycle of a large software product, as shown in figure 4, more than 90% of the code and improvements were made after the first product launch, means that sharing the original software investment with a community (eco-system) becomes a smart investment for railway operators and manufacturers alike, by simply reducing their future upgrade and maintenance costs significantly. Rather than starting to develop a new open source software package from scratch, the easiest and fastest way for a user to reach that goal would be simply by selecting one of the already existing and (as far as possible) service proven products from the market and put it under an appropriate open source license. There are numerous examples from the IT sector, such as the software development tool Eclipse, the successor to IBMs Visual Age for Java 4.0, source code was released in 2001 [28], the Internet browser Mozilla FireFox (former: Netscape Navigator), and office communication software Open Office (former: StarOffice) and many more.
with similar or sometimes even longer product life-cycles has realizing this decades ago, starting with ADA compiler in the 1980th, specifically designed for developing high assurance software for embedded control design projects, originally initiated by the US Air Force and developed by the New York University (GNAT: GNU NYU Ada Translator), which is available in the public domain and further developed by AdaCore and the GNU Project [3], [23] and a somewhat more sophisticated tools chain, which is called TOPCASED, initiated by AIRBUS Industries [29]. TOPCASED represents a tools set, based on ECLIPSE (another OSS software development tools platform [28]) for safety critical flight control applications with the objective to cover the whole life cycle of such software products, including formal specification, modeling, software generation, verification and validation, based on FLOSS in order to guarantee long term availability. TOPCASED seems to be a reasonable candidate for a future openETCS reference tools platform, since it is a highly flexible open framework, adaptable in various ways for meeting a wide range of requirements. Today manufacturers in the rail segment are using a mix of proprietary and open source tools, since some software development tools like ADA and other Products from the GNU Compiler Collection (GCC) [24] have already been used in several railway projects. Even FLOSS tools, not specifically designed for safety applications, like BugZilla for bug tracing and record keeping, have already been found its way into SIL 4 R&D programs for railway signaling [30]. The importance of qualified and certified tools is rising, since it became obvious, that poor quality tools or even malware infected tools can have a devastating effect on the quality of the final software product. 6.6 of proposed prEN 50128:2009 norm [31], modification and change control, requires to take care of the software development tools chain and processes, which in the future formally have to comply with requirements for the respective SIL level of the final product. Recent news about the STUXNET attack, a type of malware (worm) specifically designed to target industrial process control computers via its tools chain (maintenance PCs with closed source operating system) has made pretty clear, that no one can be hilled into security even not with control and monitoring systems designed for safety critical embedded applications [7]. Ken Thompson, one of the pioneers of the B Language, a predecessor of C, and UNIX operating system design has demonstrated in his Reflections on Trusting Trust [2] that compilers can be infected with malicious software parts in a way that the resulting executable software (e.g. an operating system) generated by this compiler out of a given clean (means: free of malware) source code, can be infected with a backdoor, almost invisible for the programmer. It took several years of research until David A. Wheeler suggested in his dissertation thesis (2009) a method called Diverse Double-Compiling [32], based on open source tools for countering the so called Thomson Hack. Therefore Wheeler suggests on his personal website:

> Normal mathematicians publish their proofs, and then depend on worldwide peer review to find the errors and weaknesses in their proofs. And for good reason; it turns out that many formally published math articles (which went through expert peer review before publication) have had flaws discovered later, and had to be corrected later or withdrawn. Only through lengthy, public worldwide review have these problems surfaced. If those who dedicate their lives to mathematics often make mistakes, its only reasonable to suspect that software developers who hide their code and proofs from others are far more likely to get it wrong. ... At least for safety-critical work making FLOSS (or at least world-readable) code and proofs would make sense. Why should we accept safety software that cannot undergo worldwide review? Are mathematical proofs really more important than software that protects peoples lives? [3]

### 3.4 Open Proof the ultimate Objective for openETCS

Wheelers statement confirms the need for an open source tools chain to cover the software production and documentation process for verification and validation into the open source concept in total, providing an Open Proof (OP) methodology [33]. OP should be then the ultimate objective for an openETCS project, in order to make the system as robust as possible for reliability, safety as well as for security reasons. An essential precondition for any high quality product is an un-ambiguous specification. Until this day only a written more or less structured text in natural language is the basis for ETCS product development, leaving more room for divergent interpretation (figure 3) than desirable. A potential solution for avoiding ambiguities right in the beginning of the product development process could be the conversion into a formal that means mathematical description of the functional requirement specification. As recommended by Jan Peleska in his Habilitationsschrift (post doctoral thesis) [34]:

> ... how the software crisis should be tackled in the future:
The complexity of today’s applications can only be managed by applying a combination of methods; each of them specialised to support specific development steps in an optimised way during the system development process.

The application of formal methods should be supported by development standards, i.e., explanations or “recipes” showing how to apply the methods in the most efficient way to a specific type of development task....

The application of formal methods for the development of dependable systems will only become cost-effective if the degree of re-usability is increased by means of re-usable (generic) specifications, re-usable proofs, code and even re-usable development processes.

Despite the fact that several attempts have been made in the past, a comprehensive Formal Functional Requirement Specification (FFRS) has never been completed for ETCS due to lack of resources and/or funding. Based on proprietary software business concepts there is obviously not a positive business case for suppliers for a FFRS. Formal specification works does not have to be started from scratch, because there are already a number of partial results from a series of earlier work, although that different approaches, methods and tools have been used [35], [36], [37]. Evaluating those results and trying to apply a method successfully applied in several open source projects and known as a so called Stone Soup Development Methodology might be able to bring all those elements and all experts involved together in order to contribute to such project at relatively low cost [3], [38].

3.5 Formal Methods to validate Specification for openETCS

In the first step of formalization only a generic, purely functional and therefore not implementation related specification has to be developed. This can be mainly done in the academic sector and by R&D institutes. However railway operators have to feed in their operational experience, in order to make sure that man-machine-interactions and case studies for test definitions are covering real life operational scenarios and not only synthetic test cases of solely academic interest.

For verification purposes a test case data base need to be derived from the functional specification and supplemented by a response pattern data base, which defines the expected outcome of a certain test case. That database needs to be open for all parties and should collect even all real world cases of potentially critical situations and in particular those cases, which have already caused safety relevant incidents. That means this type of formalized database will keep growing and continuously being completed to make sure that all lessons learned are on record for future tests. State of the art formal specification tools do not only provide formatting support for unambiguous graphical and textual representation of a specification document, but provide also a modeling platform to execute the model in a more or less dynamical way. This modeling can be used to verify the correctness and integrity of the ETCS specification itself not only statically, but also dynamically. In addition transitions to and from class B systems need to be specified formally as well and that might depend on national rules, even in those cases where the same class B system is used (e.g. for PZB-STMs hot stand-by functions are handled differently in Germany and Austria). Based on a particular reference architecture the resulting formal functional specification can be transformed in a formal software specification and then converted into open Proof for Railway Safety Software
executable software code. Even without existing real target hardware, those elements can be used to simulate the ETCS behavior and modeling critical operational test cases in a so called Software-in-the-Loop modeling set-up. Once the specification of the functionality has been approved and validated, the code generation can be done for the EVC embedded control system. Standardization can be accomplished by providing an Application Programmer Interface (API) similar to the approach successfully applied in the automotive industry within the AUTOSAR project [39] or for industrial process control systems based on open Programmable Logic Control (PLC) within the PLCopen project [40] including safety critical systems. In addition to the software specification, generation, verification and validation tools chain also tools for maintenance (parameter setting, system configuration, software upload services) have to be included in the OSS concept, as shown in figure 7.

3.6 How FLOSS can meet Safety and Security Requirements

For many railway experts, not familiar with open source development methodology, open source is often associated with some kind of chaotic and arbitrary access to the software source code by amateur programmers (hackers), completely out of control and therefore not suited for any kind of quality software production. This may have been an issue of the past and still being in existence with some low level projects, adequate for their purpose. However since OSS license and R&D methodologies concepts have successfully been applied to unnumbered serious business projects, even for the highest safety and security levels for governmental administration, e.g. within the iDABC, European eGovernment Services Project [26] as well as commercial, avionics [29] and military use [24], a concept based on a group of qualified and so called Trusted Developers (figure 8) having exclusively access to a so called Trusted Repository, which on the other hand can be watched and closely monitored by a large community of developers, being able to post bug reports and other findings visible to the whole community, has made this so called bazaar process [17] to a much more robust methodology compared with any other proprietary development scheme. According to several research projects, OSS projects in general tend to find malicious code faster than closed source projects, which is indicated for example in the average life time of so called backdoors, a potential security threat, which might exist in closed source software for several month or even years, while having an average survival time of days or few weeks, at the most, in the case of well managed OSS projects [3], [4], [5], [32]. Figure 8 demonstrates the principle information and source code flow for a typical FLOSS development set-up.

![FIGURE 8: The classical Stone Soup Development Methodology often applied in Open Source Software projects according to [3], where the User in most cases is also active as Developer, which need to be adapted to the rail sector, where Users may be more in a reporting rather developing role. Only trusted developers are privileged to make changes to the source code in the trusted repository, all others have read only access.](image)

It is not in question that well acknowledged and mandatory rules and regulations according to state of the art R&D processes and procedures (e.g. EN 50128) have to be applied to any software part in order to get approval from safety authorities before going into revenue service. While open source eco-systems in the IT industry are generally driven by users, having the expertise and therefore being in a position to contribute to the software source code themselves, so it seems unlikely for the railway segment to find many end users of embedded control equipment for ETCS (here: railway operators or railway vehicle owners), who will have this level of expertise. Therefore the classical OSS development concept and organization has to be adapted to the railway sector. Figure 9 shows a proposal for an open source software development eco-system for openETCS utilizing a neutral organization to coordinate the so-called “co-competition” business model for cooperating several competing equipment integrators and distributors for ETCS onboard products and services based on a common FLOSS standard core module, adapted to the needs of the railway signaling sector providing high assurance products to be authorized by safety authorities (NSA, NoBo). The concept as shown in figure 9 assumes a license with Copy-Left, requiring in general distributing the source code free of charge, even if code has been added or modified and further distributed, so that
the community can re-use the improvements as well. That means that only certain added values can be sold for a fee. Typical added values can be service for software maintenance (bug-fixing), software adaptation for specific applications, integration into embedded control hardware and integration into the vehicle system, test and homologation services, training for personnel and so forth.

For further development of the software, especially for the development of new complex add-on functions, costly functional improvements, etc., it might be difficult to find funding, since a Copy-Left in the FLOSS license requires to publish that software free of charge, when distributed.

Therefore many OSS projects are using a so-called dual licensing policy, by offering the identical software under two (or more) different license agreements (figure 10). One might be the European EUPL, a Copy-Left type FLOSS license and the other one can be a For-Fee-License (without Copy-Left), which does not require publishing all modification. In exchange ac certain fee has to be paid, which may also provide for warranty and other services. Combined with a scheduled release scheme (e.g. defining a fixed release day per year or any other reasonable frequency), all new modules will be available only under the For-Fee-License first, until R&D costs have been paid off by those users, who want to make use of the new functionality, while all others can stick with the older, but free of charge software versions. Once the new features are paid off, those particular software modules can then be set under the FLOSS license (EUPL). That allows fair cost sharing for all early implementers and does not leave an undesired burden on those users, who can live without the additional functions for a while, but still being able to upgrade later on.

Since those upgrades will be provided by service level agreements through OEMs or software service providers, customers have the choice to either opt for low cost, but later upgrade service or higher priced early implementing services, whatever fits best to their business needs. The dual-licensing scheme has an additional advantage, allowing even those ETCS suppliers, who are not able, due to technical limitations or legal restrictions caused by their legacy system design or other reasons, to put their software under an OSS license, never the less being able to participate in the cost sharing effects for further
add-on functional development. In most cases it is technically much easier to implement a small API, interfacing just for the add-on functions, rather than providing a fully functional API for the whole kernel (figure 11).

If the non-OSS supplier wants to make use of those Add-on-SW-Modules from the library, he cannot use the OSS-licensed software, but can combine any proprietary software with alterative licensed software, not including a Copy-Left provision. Besides commercial matters also technical constrains have to be taken into account when combining software parts, developed for different architectural designs. A concept of hardware virtualization has already been discussed to overcome potential security issues [43].

3.7 How to Phase-in an OSS Approach into a Proprietary Environment?

Even though the original concept of the ETCS goes far back into the early 1990 years projecting an open white-box design of interchangeable building blocks, independent from certain manufacturers, based on a common specification and mainly driven by the railway operators organized in the UIC (Union International des Chemin de Fer = International Union of Railways), software was not a central issue and open source software concepts were in its infancy [41], [42]. Since then a lot of conceptual effort and detailed product development work has been done, but the white box approach has never been adapted by the manufacturing industry. Besides various difficulties and shortcomings, as mentioned earlier, the European signal manufacturers have developed several products, more or less fit for its purpose and it would be unwise to ignore this status of development and start a brand new development path from scratch. This would just lead to another product competing in an even more fragmented market rather than promoting an effective product standard. In addition, it needs at least one strong manufacturer with undoubted reputation and a sound financial basis in combination with a sufficient customer base to enforce a standard in a certain market. Therefore starting a new product line, by having the need to catch up with more than a decade of R&D efforts is not an option.

Based on this insight, a viable strategy has to act in two ways:

- 1. Ground work has to be started to provide an open source reference system, based on an unambiguous specification, which means using formal methods, in order to deliver a reference onboard system as soon as possible, which can be used to compare various products on the market in a simulated as well as real world infrastructure test environment. This device needs to be functionally correct, however does not to be a vital (or fail-safe) implementation.

- 2. At least one or better more manufacturers have to be convinced to share-in into an open source software based business approach by simply converting their existing and approved proprietary ETCS onboard product into an open source software product by just switching to a FLOSS license agreement, preferably by using the European Union Public License (EUPL), including interface definition and safety case documentation. No technical changes are required.

- 3. Once a formally specified reference FLOSS package has been provided, implemented on a non-vital reference hardware architecture, according to step 1, in a future step by step approach all add-on functions and enhancements and future major software releases should be based on formal specifications, allowing a migration of the original manufacturers software design solution into the formal method based approach, due to the openness of the product(s) from step 2.

Figure 12 demonstrates this two path approach with a conventional roll-out scheme, as planned by a

**FIGURE 12:** Interaction between openETCS project providing formally specified non-vital reference OBU for validating proprietary as well into OSS converted industrial products and for future migration to a fully formally specified openETCS software version to implemented in a market product.
supplier, based on proprietary designs (upper half) and major milestones for the openETCS project, providing a non-vital OBU based on formal specification and later migrating to a formally specified vendor specific implementation of the kernel software (lower half).

Trying to implement an independent formal open source software package without the backing of at least one strong manufacturer, will most likely fail if no approved and certified product can be used to start with. The only promising way to accomplish the crucial second step in this concept is by using a tender for a sufficiently attractive (large enough) ETCS retrofit project by adding a request for an OSS license for the software to be delivered. The EU commission has provided a guideline for such OSS driven tenders, the so called OSOR Procurement Guide (Guideline on public procurement of Open Source Software, issued March 2010, [26]). As an example, figure 12 shows the time line for an ETCS retrofit project for high speed passenger trains to be equipped by 2012 with an pre-baseline 3 proprietary software in 2012, to be added by an open source license as soon as the first baseline 3 software package is expected to be released.

3.8 Economical Aspects of openETCS for Europes Railway Sector

A free of charge, high-quality ETCS vehicle software product on the market, makes it less attractive, under economical aspects, to start a new software development or even further development of a different but functionally identical proprietary software product. This will lead sooner or later to some kind of cooperation of competing ETCS equipment suppliers, a co-competition with all those suppliers who can and will adapt their own products by providing an API to their particular system. Due to the fact that very different design and safety philosophies have been evolved in the past years, some of the manufacturers have to decide either to convert their systems or share into the co-competition grouping, or otherwise stick with costly proprietary software maintenance on their own. As figure 4 demonstrates clearly that the increase of the software volume over time may exceed the original volume by a factor of 3. It is unlikely to assume that the development of the ETCS vehicle software will run much differently. Then it will be very obvious that for a relatively limited market, of perhaps up to 50,000 rail cars to be equipped with ETCS in Europe, a larger number of parallel software product development lines will hardly be able to survive. A study funded by the EU Commission [22] has identified a potential average cost reduction of 36% for the corresponding R&D by the use of FLOSS. As a result, a significantly lower cost of ownership for vehicle operators would accelerate the ETCS migration on the vehicle side.

3.9 Benefits for the ETCS Manufacturers

The core of the ETCS software functionality defined by UNISIG subset 026, to be implemented in each EVC, is a published and binding standard requirement and therefore not suitable for defining an Unique Selling Proposition (USP). As a result it makes perfectly sense from the perspective of manufacturers, to share the development cost and the risk for all R&D of the ETCS core functionality even with their competitors, often practiced in other industrial sectors (e.g. automotive). The involvement of several manufacturers in the development of openETCS will help to enhance the quality in terms of security and reliability (stability and safety) of the software, because different design traditions and experiences can easily complement each other. As a FLOSS-based business model can no longer rely on the sales of the software as such, the business focus has to be shifted to services around the software and even other add-on features to the product. That means the business has to evolve into service contracts for product maintenance (further development, performance enhancements and bug fixes). It thereby helps the ETCS equipment manufacturers to generate a dependable long-term cash flow, funding software maintenance teams even long after the hardware product has been discontinued and to cover long term maintenance obligations for the product even by third parties, helping to reserves scarce software development resources for future product R&D. With respect to the scarcity of well educated software engineers from Universities, FLOSS has the side effect, that openETCS can and most likely will become subject to academic research, generating numerous master and dissertation thesiss and student research projects.

3.10 Benefits for Operators and Vehicle Owners

The use of openETCS is a better protection for the vehicle owners investment, because an obsolescence problem on the hardware side does not necessarily mean discontinued software service. Modification
of the ETCS kernel can also be developed by independent software producers. This enables competition on after-sales services and enhancements, because not only the software sources but also associated software development tools are accessible to all parties. As shown above, due to the complexity of the software, malfunctions of the system may show up many years, even decades after commissioning. Conventional procurement processes are therefore not suitable, since they provide only a few years of warranty coverage for those kinds of defects. These concepts imply that customers would be able to find all potential defects within this limited time frame, just by applying reasonable care and observation of the product by the user, which does not match experiences with complex software packages with more than 100,000 lines of code. This finding suggests that complex software will need care during the whole lifecycle. Since software matures during long term quality maintenance, means that during early usage, or after major changes, the software may need more intensive care whereas in its later period of use, service intensity may slow down. But as long as the software is in use, a stand-by team is needed to counter unforeseeable malfunctions, triggered by extremely rare operational conditions. As the ETCS onboard software can be considered as mission critical, operators are well advised to maintain a service level agreement to get the systems up and running again, even after worst case scenarios. Railway operators and vehicle owners are usually not able to provide that software support for themselves. They usually rely on services provided by the OEM. However due to slowing service intensity after several years of operation, this service model may not match the OEMs cost structure in particular after the hardware has been phased out. In those cases OEMs are likely to increase prices or even to discontinue this kind of service. A typical escrow agreement for proprietary software might help, but has its price too, because alternative service providers have first to learn how to deal with the software. Only a well established FLOSS-ecosystem can fill in the gap at reasonable cost for the end user, and that is only possible with FLOSS. DBs experience with FLOSS is very positive in general. For more than a decade, DB is using FLOSS in various ways: In office applications, for the intranet and DBs official internet presence and services on more than 2000 servers world-wide and even in business critical applications. The original decision in favor of FLOSS was mainly driven by expected savings on license cost. However looking back, quality became a more important issue over time, since FLOSS application have had never caused a service level breach, which cannot be said for proprietary software, selected by applying the same quality criteria. This supports the impression that FLOSS does tend to have a higher quality.

4 Conclusion

The major goal of unified European train control, signaling, and train protection system, ETCS, has led to highly complex functionality for the onboard units, which converts into a level of complexity for the safety critical software not seen on rail vehicles before. A lack of standardization on various levels, different national homologation procedures and a diversity of operational rules to be covered, combined with interfacing to several legacy systems during a lengthy transitional period has to be considered as a major cost driver. Therefore, even compared with some of the more sophisticated legacy ATP and ATC systems in Europe ETCS has turned out to be far more expensive without providing much if any additional performance or safety advantages. Due to ambiguities in the system requirement specification (SRS) various deviations have been revealed in several projects, so that even the ultimate goal of full interoperability has not yet been accomplished. Therefore the development of ETCS has to be considered as work in progress, resulting in many software upgrades to be expected in the near and distant future. Since almost all products on the market are based on proprietary software, this means a low degree of standardization for the most complex component as well as life-long dependency to the original equipment manufacturers with high cost of ownership for vehicle holders and operators. Therefore an open source approach has been suggested, not only covering the embedded control software of the ETCS onboard unit itself, but including all tools and documents in order to make the whole product life cycle as transparent as possible optimizing economy, reliability, safety and security alike. This concept is called open proof a new approach for the railway signaling sector. A dual licensing concept is suggested, based on the European Union Public License with a non-copy left for-fee-license on the other hand to provide a cost sharing effect for participating suppliers and service providers. By offering a trusted repository, a dedicated sources code access policy in combination with a release schedule policy, economical as well as safety and security considerations can be taken into account. A two step approach, providing a formally specified non-vital reference system and a procurement program, asking for converting existing commercial products from closed source into open
source, and later merging those two approaches, is expected to enhance quality and safety parameters in the long run. A neutral independent and mainly not-for-profit organization is suggested to manage the project involving all major stake holders to define the future product strategy. The whole openETCS project has to be considered as a business conversion project from a purely competitive sales oriented market into a co-competitive service market, enhancing cooperation on standards by enabling competition on implementation and services. It is well understood that such a change cannot be accomplished even by one of the largest railway operators alone. Therefore several EU railway organizations, as there are: ATOC (UK), DB (D), NS (NL), SNCF (F) and Trenitalia (I) have already signed a Memorandum of Understanding promoting the openETCS concept in the framework of an international project.

References


Open Proof for Railway Safety Software

[40] PLCopen; Molenstraat 34, 4201 CX Gorinchem, NL.: http://www.plcopen.org/

[41] UIC/ERRI A200: ETCS, European Train Control System, Overall Project Declaration including the contribution to be made by UIC, Utrecht, NL, Jan. 1992,


Migrating an OSEK run-time environment to the OVERSEE platform

Andreas Platschek
OpenTech EDV Research GmbH
Augasse21, A-2193 Bullendorf
andreas.platschek@opentech.at

Georg Schiesser
OpenTech EDV Research GmbH
Augasse21, A-2193 Bullendorf
georg.schiesser@opentech.at

Abstract

As virtualization techniques are being used in the automotive industry, in order to save hardware, reduce power consumption and allow the reuse of legacy applications, as well as allow the fast development and integration of new applications, the need for a run-time environment that is suitable and in wide use in the automotive industry emerges. The requirements for such a run-time environment are defined in the most widely used specification in this industry - OSEK/VDX.

One key feature the OVERSEE project is taking advantage of, is that co-locating a OSEK run-time environment and a full-featured GPOS GNU/Linux eliminates many limitations of OSEK/VDX by the extension through virtualization and notably allowing to mitigate some of the serious shortcomings in the security area by resolving these issues at the architectural level rather than trying to patch up the limited OSEK OS. This may well constitute a general trend to specialize operating systems and operate powerful hardware as an assortment of specialized FLOSS systems collaborating to provide different services, including full backwards compatibility to legacy operating systems.

Currently, several FLOSS implementation of this specification are available under different FLOSS license models and with a different degree of compliance. This paper gives an overview of the available implementations, a rational for the chosen implementation as well as a description of the efforts for the migration to XtratuM.

1 Introduction

In the effort to reduce costs by saving hardware and reuse of legacy code, the automotive industry is relying on well specified and standardized operating systems. The OSEK specification (Open Systems and the Corresponding Interfaces for Automotive Electronics) has been around since 1993 and after merging with the VDX (Vehicle Distributed Executive) it has grown to the most important operating system specification in the automotive industry.

OSEK’s main goals are to specify an operating system that is suitable for the automotive industry, and that allows to write highly portable applications which only depend on an OSEK compliant API. Furthermore the OSEK communication specification provides a well specified API for internal as well as external communication, turning OSEK compliant operating systems into highly portable, scalable operating systems that support re-usability of legacy OSEK compliant applications.

Although it’s successor [7] is going to be the future of the industry, OSEK/VDX will be around for quite some time, since it is the basis for AUTOSAR:

"The OS shall provide an API that is backward compatible to the API of OSEK OS. New require-
ments shall be integrated as an extension of the functionality provided by OSEK OS." [BSW097] Existing OS, AUTOSAR, Requirements on Operating System V2.1.0 R4.0 Rev 1

All this explains, why support for an OSEK compliant run-time environments is an indispensable requirement for a software platform - like the one developed in the OVERSEE [1] project - that targets the automotive industry. A high level view of this software platform can be seen in figure 1.

FIGURE 1: High Level Architecture

This paper will give an introduction to the OSEK/VDX operating system specification, and describe the efforts that were necessary to allow the execution of FreeOSEK [2] a FLOSS implementation of OSEK/VDX in a virtualized environment, namely the XtratuM hypervisor allowing to run several FreeOSEK run-time environments in parallel with other run-time environments like Linux partitions or LithOS [9] partitions, while guaranteeing the independence between those run-time environments.

2 OSEK/VDX

In the following, the open operating system specification OSEK/VDX [8] is summarized, looking at the highest conformance class ECC2 (extended conformance class). The lower conformance classes are subsets of ECC2, the relation between the conformance classes can be found in [4], Figure 3-3.

2.1 OSEK OS

The most important part of OSEK/VDX to understand the context of this paper is OSEK OS. It specifies a operating system, well suited for the needs of the automotive industry. The standardized API and well defined behavior of OSEK/VDX compliant operating systems, allow high portability of applications developed for such an operating system.

The following summarizes the essential points of OSEK/VDX, for more details, please refer to the homepage [4], where all parts can be downloaded free of charge, since it is an open standard.

Task Management OSEK/VDX distinguishes between two different types of tasks, basic tasks (BT) and extended tasks (ET). While a BT can only release the processor if it terminates, or if it is preempted by a higher priority task or an interrupt service routine (ISR), an ET can also go into a waiting state, allowing the scheduler to dispatch a lower priority task, without terminating the higher priority task. An example for this would be, if the ET is waiting for some kind of event to happen. Instead of just polling and wasting CPU time, it can go into the waiting state, in state it is not scheduled, before the event is signaled (more on signals below).

OSEK/VDX provides a Task state Model ([4], section 4.2) that describes the states a task can be in, and the transitions between those tasks. The task state model for extended tasks is shown in figure 2. For basic tasks the task state model is essentially the same, but without the waiting state.

The states a task can be in are the following:

- **running** - a task in the running state is currently active and executed. At all times only one task can be in the running state. (OSEK/VDX is specified for single core CPUs only, multi-core solutions are covered by newer versions of AUTOSAR)
- **ready** - all schedulable tasks are in the ready state, waiting for their turn to transition into the running state.
- **suspended** - tasks in the suspended task are currently inactive and wait for their activation to become ready.
- **waiting** - extended tasks that are waiting for some event to happen can decide to go into the waiting state instead of wasting CPU time. A task in the waiting state will be released from the waiting state as soon as the desired event has happened.
In the OSEK/VDX task management, the scheduling policy is assigned by the system integrator. A system's scheduling policy can be configured to be fully preemptive, non-preemptive or mixed (both preemptable as well as non-preemptable tasks are running at the same time). The scheduling decision itself is based on priority scheduling, with static priorities (where 0 is the lowest priority and bigger numbers denote higher priorities). Depending on the conformance class, one or more tasks of the same priority can exist at the same time.

If preemption is disabled, only voluntary preemption of tasks is possible, rescheduling happens only in the following cases:

- the running task terminates successfully
- explicit call of the scheduler by the running task
- the running task transitions into the waiting state

**Interrupts** - OSEK/VDX distinguishes between 2 types of interrupts:

- category 1 ISRs do not use operating system services, and after they are finished, execution continues exactly at the point where it was before the ISR has been called (no influence on task management).
- category 2 ISRs are allowed to use operating system services that are concerned with handling interrupts (enable, disable, etc.), these ISRs prepare the system for a RTE to run a dedicated user routine (comparable to Bottom Halves). After a category 2 ISR has been executed, the execution does not return to the last point before the interrupt, instead the scheduler is invoked, in order to check if a dedicated user routine (bottom half) has a higher priority than the current running task.

Depending on the scheduling policy, the point of rescheduling is either, when the event is set (fully preemptive) or at the next point of rescheduling in non-preemptive mode (listed above in Task Management).

**Events** are a means of synchronization. They are only available for extended tasks, since they are used to transition tasks into and out of the waiting state. Events are objects assigned to tasks, and uniquely identified through their name and the task they belong to. At task activation of an extended task, all the events are cleared automatically. Events can be set by any task (also basic tasks) as well as category 2 ISRs, to change the task state of the events owner from the waiting to the ready state, but only the owner of the event is allowed to clear the event afterwards.

Depending on the scheduling policy, the point of rescheduling is either, when the event is set (fully preemptive) or at the next point of rescheduling in non-preemptive mode (listed above in Task Management).

**Resource Management** In order to allow the concurrent task execution model described above, a resource management has to be provided, in order to assure

- mutually exclusive access to resources
- prevent priority inversion
- detect and prevent deadlocks
- and access to a resource must never lead to a transition into a waiting state

All these problems are high probable error sources, the goal of the OSEK resource management system is to do everything possible to
prevent them from the operating system side. To reach these goals, the following mechanisms are specified by OSEK/VDX:

OSEK Priority Ceiling Protocol [4], section 8.5, introduces the OSEK Priority Ceiling Protocol, used to avoid priority inversion and deadlocks between tasks. This protocol provides a ceiling priority for each resource (this ceiling priority is statically assigned at system generation), which shall be set to priority of the highest-prior task using the resource. If a task with a lower priority accesses the resource, it’s own priority is risen to the resource priority temporarily. After the task releases the resource, it’s priority is set back to it’s old priority. This way, it is not possible that the task is preempted by an higher prior task that competes for the same resource, while the lower prior task is holding the resource.

Section 8.6 of [4] introduces an optional extension of the OSEK Priority Ceiling Protocol, that includes ISRs. Restrictions when using Resources

OSEK/VDX defines restrictions on the system calls that may be used, while a task is holding a resource. The calls forbidden while holding a resource are TerminateTask, ChainTask, Schedule and WaitEvent. As can be inferred from the names, those calls that invoke the scheduler and might lead to the scheduling of another task are the ones prohibited while holding a resource.

This is a simple an effective way of assuring the mutual exclusivity of resources, furthermore it helps to prevent deadlocks between tasks.

Scheduler as a Resource

If a task wants to prevent itself from being preempted, it can lock the scheduler. If a task chooses to do so, the scheduler is still invoked, but not allowed to schedule any other tasks. Interrupts are received and processed independently of the state of the scheduler.

Alarms are special (time-dependent) events, offered by the OSEK OS, to activate tasks after a counter has experienced. A counter in OSEK is represented a counter value measured in ticks, if the counter reaches a predefined value, the alarm expires and the alarm-event is set off. The predefined value can be specified either relative to the actual counter value (relative alarm) or as an absolute value (absolute alarm).

The counter value can be incremented by all kinds of sources, of course this could be a real-time clock, but it could also be any other interrupt source that increments the counter.

While any number of alarms can be assigned to the same counter, each alarm has exactly one counter and exactly one alarm-callback routine assigned at generation time.

Error Handling

OSEK/VDX defines hook routines which can be used for a variety of tasks.

Hook Routines are part of the operating system, although implemented by the applications developer. They can be seen as a possibility for the application developer to extend the functionality of the operating system. The hook routines are called by the OS at pre-configured events, which events depends on the implementation of the operating system itself. Since hook routines are part of the OS, they have higher priority than all tasks, and they can not be interrupted by category2 ISRs. While the interface for hook routines are standardized, functionality is not and is up to the application developer.

Error Handling

OSEK/VDX distinguishes between two categories of errors - application errors and fatal errors. In case of a fatal error, the integrity of the operating systems internal data can no longer be guaranteed, and the operating systems shut down. If an application error occurs, a system call could not be serviced properly, but the internal data of the operating system is still assumed to be correct. If a system service routine returns an error code, an error hook routine is called. This hook routine has to be provided by the user, who has the responsibility to bring his application back on track.

System Startup/Shutdown

All low level (hardware) initialization is up to the application developer, the specifications of the OSEK/VDX concern only the platform independent parts and start with the call to StartOS. Shutdowns are a little more complicated, since each task has to be informed of the
shutdown, so it can bring potential actu-
ators into a safe state. Therefore before
the system can actually shutdown, a shut-
down hook is called.

**Debugging** is done via a *PreTaskHook* and a *PostTaskHook*, which are called on task
switches. These hooks can be used for de-
bugging and measurement purposes.

**Standardized API** in [4], sections 12 and 13, the
system services provided by the API of an
OSEK/VDX compliant operating system are
specified. This API must be the only way for
the application to use the above described op-
erating subsystems, like alarms, events, etc.

### 2.2 Other parts of OSEK/VDX

OSEK/VDX consists of multiple parts, OSEK OS
described previously is the most important one for
the porting efforts of FreeOSEK to XtratuM, while
the other parts do not really play a role in this con-
text (except for some sections of OSEK Com). But
for completeness, here is a short list of all parts:

**OSEK COM - Communication Layer** specifies
a message based communication for (inter pro-
cessor) communication - it shows a stunning
resemblance with ARINC653 interpartition
communication, but describes a communica-
tion system for internal and external commu-
nication.

**OSEK NM - Network Management** provides a
standardized way of configuring networks of
OSEK/VDX nodes, initialization of networking
peripherals, network start-up, network
monitoring and a lot more, everything that is
needed to start, maintain and diagnose a net-
work of nodes running OSEK/VDX compliant
nodes.

**OSEK OIL - OSEK Interpretation Language**
specifies a standardized configuration mecha-
nism for OSEK/VDX compliant nodes. The
configuration files as defined by OSEK OIL are
per node (single CPU nodes only), and do not
include network configuration.

**OSEK Time - Time-Triggered OS** specifies a
time-triggered variant of OSEK VDX, the dif-
ferences are e.g. time-triggered scheduling and
the like. It is also possible to run a mixed
variant, were a standard OSEK OS is run in
time slots of the time-triggered OS.

**OSEK FTCom - Fault-Tolerant Communication**
provides a standardized time-triggered net-
working variant that in order to achieve bet-
ter fault-tolerance than with the standard
OSEK/VDX networking layer.

### 2.3 AUTOSAR

While OSEK/VDX is currently the most used op-
erating system standard in the automotive industry,
its successor AUTOSAR [7] is on it’s way to take
over. The main reason for this is definitely the fact
that the newest release - AUTOSAR 4.0 - is the first
operating systems standard taking multi-core CPU’s
into account. Since the days of single-core CPU’s are
counted, this a real important topic that will shake
the safety-community over the next years.

For the OVERSEE project, AUTOSAR is inves-
tigated, but not strictly followed, the reason is sim-
ply it’s size and the fact that AUTOSAR is based
on OSEK/VDX, so every application written for an
OSEK/VDX compliant operating system can also be
executed on a AUTOSAR compliant operating sys-
tem. Nevertheless OVERSEE’s design decisions are
loosely based on the AUTOSAR architecture.

### 3 XtratuM

XtratuM is a type II (bare metal) hypervisor tar-
geting safety related composable systems. The main
guidelines for design come from one of the key IMA
standards, ARINC 653 [3]. XtratuM is an active
FLOSS project being developed at Instituto de In-
formatica Industrial, Universidad Politecnica de Va-
len西亚. While the OVERSEE project is focused on
security aspects the goal is to provide a platform
that in principle can also satisfy safety requirements.
There is a strong sharing of core demands on the low-
est OS layer with respect to safety and security, and
while safety and security have sometimes conflicting
demands at higher levels these differences are not
present at the lowest level of a hypervisor [10]. The
key to unify the requirements at the lowest level of
safety and security is to provide a sound:

- **Temporal isolation**
- **Spatial isolation**

allowing to build high-level services on top that
only allows explicitly permitted sharing of resources
as well as communication. XtratuM thus is inten-
tionally reduced close to the bare minimum that is
needed to allow high-level services to operate in their respective OS environments and still give strong guarantees with respect to independence.

3.1 XM Hypercall Interface

XtratuM offers a relatively narrow interface of Hypercalls to its partitions. This simplified things a lot for our porting efforts. In this section we will only briefly outline hypercalls that were used in this porting effort, for a full list of available hypercalls we refer you to the XtratuM Reference Manual [11]. The intention of this section is to show the interface size used in the XtratuM guest management for an actual example.

- **Time services**: XtratuM provides an independent virtual time to each domain on which the guest-OS then can implement high-level timing services. In this sense the low-level services can be seen as mimicking hardware timing services.
  - **XM_set_time**: Time entities in XtratuM are of microsecond granularity, and are maintained relative to the last system reset. There are two basic clocks in the system. Clocks in XtratuM are strictly monotonic. Clocks are maintained for the system (XM_HW_CLOCK) as well as for the partitions execution (XM_EXEC_CCLOCK).
  - **XM_set_timer**: Interval timer service (providing one-shot behavior by setting the interval to 0). The expire time is an absolute time with respect to either hardware clock or execution clock. To a partition the expired timer is signaled as a virtual timer interrupt (emulating a hardware timer).

- **Interrupt services**: Signaling to partitions is provided via virtual interrupts, it is up to the guest-OS to then assign suitable meaning and response to the events. Note the absence of a interrupt request hypercall - as all resources are allocated statically in XtratuM there is no need for a request_irq.
  - **XM_enable_irqs**: globally disable interrupt delivery to this partition
  - **XM_disable_irqs**: globally enable interrupt delivery
  - **XM_set_irqmask**: used for masking (blocking) and unmasking of interrupts

- **Basic partition management functions**: Much of the partition management is related to the initialization and shutdown phase of a partition. The essence of the interface is that it minimizes the state information that needs to be handled by the hypervisor - leaving more or less all state related work to the partition.
  - **XM_suspend_partition**: This is a basic function that is only used in supervisor mode to manage a partition. It is used to block a partition (waiting on a resource) or temporarily stop a partition if errors are detected.
  - **XM_resume_partition**: Simply the opposite to the above partition suspension.
  - **XM_shutdown_partition**: As the hypervisor does not have information about the internal state of a partition shutdown is provided as an asynchronous notification. Basically a partition is sent a request to shut down via a dedicated interrupt and after cleaning up any internal state will then terminate itself.
  - **XM_reset_partition**: Conversely to the **XM_shutdown_partition**, the **XM_reset_partition** is a forced shutdown of a partition whereby a warm and cold reset is differentiated, a warm reset preserves some of the partitions initialized resources (i.e. open ports and memory areas) while a cold reset clears this all and thus can have side-effects on other partitions via communication channels no longer being served.
  - **XM_halt_partition**: A halted partition is set into an inactive state but no reclamation of resources (spatial or temporal) are done (that is left to the partition reset) in this state the partition is simply no longer scheduled by the hypervisor. The **XM_halt_partition** called by non-supervisor partitions can only pass self as the target of the halt.
  - **XM_idle_self**: This allows a partition to suspend itself within its time slot. The partition will only be re-woken on its next time-slot or if a NMI is received within its current time slot. This can be used to implement donation schemes for system partitions.

- **Basic system management functions**: Note that these are not directly related to the guest-OS as these calls are related to privileged domains - they are listed here for completeness.
XM_halt_system: The halt partition call (also described above) is used by system partitions to manage the system as a whole as well as individual partitions. Only supervisor partitions can halt other partitions. This is used to prepare a partition reset as well as mode switching.

XM_reset_system: Brute force system halt of the entire board after this only a hardware reset can reboot the system. No precautions are taken to put any partition into a sane state thus this is only the last step in a system shutdown as well as in extreme emergency situations.

- Low level Communication related functions: In practical implementations one does not actually use the low level object class functions but uses the wrappers provided to the commonly used objects (sampling and queuing ports as specified in ARINC 653). These wrappers thus are the actual hypercalls that will be issued though they are rarely used in guest-OS code.
  - XM_read_object: read the object, verifying access permissions and other low-level properties. Usage in all reading functions like XM_receive_queueing_message, XM_read_sampling_message, etc.
  - XM_write_object: write the object. This is used i.e. in XM_write_sampling_queueing_message, XM_send_queueing_message, etc.
  - XM_ctrl_object: is used to create and manage objects with specific properties as well as query these objects (i.e. retrieve the id of the object). This hypercall is used in object management functions like XM_create_sampling_queueing_port, XM_get_sampling_queueing_port_status, etc.

While the overall hypercall set is a bit more elaborate than listed here, the essential calls used to implement the OSEK guest-OS are listed showing how small such a guest-OS code may be which the abstraction level is pulled down far enough. A full description of the interface is out of scope for this paper though.

4 FreeOSEK

FreeOSEK[2] is a OSEK implementation started by Mariano Cerdeiro. It originally ran on ARM and on POSIX compliant platforms (this is just a simulation environment, running FreeOSEK as a user-space process intended to allow everyone to test it on a normal Linux desktop).

FreeOSEK is licensed under the GPLv3 with link exception. This means, that you can link your code into FreeOSEK and can still license your code under whatever license you want (free or proprietary).

According to the FreeOSEK homepage, they currently run about 80% of the OSEK OS conformance tests, and of those about 95% pass. In addition, FreeOSEK is tested, using the static code checking tool splint.

Fortunately big parts of FreeOSEK are generic C-code (e.g. the task scheduler) and only the parts that directly deal with hardware had to be adapted (see section 5 for details).

While OSEK OS is almost complete, OSEK Com is more or less non existent in FreeOSEK, but this is no big problem for us, as we will see later in section 5.4, since most of the functionality needed for OSEK Com compliant communication is already provided by XtratuM.

5 Porting Efforts

The following section describes the efforts that have to be taken to run FreeOSEK as an run-time environment in a XtratuM partition.

This includes also a description of which steps already have been achieved successfully, and gives insight into the parts that will need more work. To anticipate the most important thing first: As of this writing, FreeOSEK can be used as an XtratuM run-time environment, but more work will be needed to make a full compliant version possible, most notably in the task management and communication subsytem some (re)work will be necessary.

5.1 Adaptation of the Build System

The first step to running FreeOSEK inside of an XtratuM partition, was to adapt FreeOSEK’s build system, so that the resulting binary would be accepted by XtratuM. The most important thing here is, that FreeOSEK must not be compiled as an executable binary, but instead it has to be compiled as a relocatable object, that can be linked into an XtratuM partition - if necessary even in multiple partitions - at a memory address that is specified at configuration time in the XtratuM configuration file.
After this stage it is already possible to boot into FreeOSEK, and to put some `xprintf`'s\(^1\) into the init code. Since most of the initialization code is generic (e.g. load the data of the application’s task) this is already done without any changes to the FreeOSEK code base. The next point that really needed attention, was the x86 specific code for the task switches.

### 5.2 Task Management

In order to assure a flawless scheduling of tasks, it has to be assured, that for each possible point of rescheduling, the transition from the old to the new task is done properly.

Which actions have to be performed during dispatching, depends on the event that led to the rescheduling - that is on the point of rescheduling itself.

OSEK OS lists the following 4 points of rescheduling for non-preemptive scheduling:

- Task Termination
- explicit activation of successor task
- explicit call of the scheduler
- a transition into a waiting state takes place

Let’s have a quick look at those four points of rescheduling. The first two can be handled really easily, for those two, the task context of the old task does not have to be saved, since it terminates, before the new task is scheduled. Therefore, all that was needed to get a basic version of FreeOSEK running on XtratuM, was to set the stack pointer to the stack of the new task, and jump into task itself. This way, simple examples that activate non-preemptive tasks, and chain non-preemptive tasks can already be run.

If preemptive scheduling is desired, the following extended list of points of rescheduling has to be considered:

- Task Termination
- explicit activation of successor task
- activation of a task at task level
- explicit call of the scheduler
- a transition into a waiting state takes place

In order to allow preemption of tasks (either voluntarily by going into waiting states or involuntarily by hitting one of the points of rescheduling from the above list, the context has to be saved before and restored after rescheduling, this part of the task management is not clean yet and will need some rework so it can be considered done. For a proof of concept as necessary by the OVERSEE project, other parts of OSEK are more important and will therefore need to be handled before finishing up task management.

### 5.3 Counters and Alarms

As described above, one way a task can be activated is if an alarm has expired. Each alarm is triggered by exactly one counter.

Counters can be incremented by all kinds of events but one of the most common ones are timers, in order to allow timed activation of tasks. All that was to do, to allow alarms that wake up tasks, was to add an IRQ handler which is triggered by the virtualized XM timer interrupts. Inside of this IRQ handler a counter is incremented, using the OSEK defined `IncrementCounter()` call. The virtualized timer is configured in the initialization code of FreeOSEK. Now one or more alarm(s) can be associated with the counter in the OIL configuration file of the application, to make those alarms go off as soon as the counter has reached a limit.

An example for such configuration could look like this (only the part that deals with counters and alarms):

```plaintext
COUNTER HardwareCounter {
  MAXALLOWEDVALUE = 100000;
  TICKSPERBASE = 1000;
  MINCYCLE = 1;
  TYPE = HARDWARE;
  COUNTER = HWCOUNTER0;
};

COUNTER SoftwareCounter {
  MAXALLOWEDVALUE = 100000;
  TICKSPERBASE = 100;
  MINCYCLE = 1;
};
```

\(^1\) `xprintf` is a library function of `libxm` wrapping a XM `write` console, giving the application programmer a way to use formatted printing.
5.4 Interpartition Communication

Communication in OSEK is defined in [5], which defines the main goals of this specification as follows:

"It is the aim of the OSEK COM specification to support the portability, re-usability and interoperability of application software. The API hides the differences between internal and external communication as well as different communication protocols, bus systems and networks." [OSEK Communication Specification 3.0.3, 1.1 Requirements]

From this paragraph we already can deduce, that connecting FreeOSEK to the message passing interpartition communication system that is provided by XtratuM is conforming to the specification. More importantly, the latter part stating that communication protocols as well as communication media should be transparent to the application implies, that it has to be possible, to run a legacy OSEK compliant application, that uses OSEK COM. Specifically it can be run in an FreeOSEK run-time environment communicating via the XtratuM interpartition communication system instead of let’s say a CAN bus, without even knowing it, and without the need of changing a single line of application code.

One further thing we can take into account, is the ARINC653 compliant interpartition communication system provided by XtratuM, and the resemblance of the OSEK Com system and the ARINC 653 interpartition communication system. This similarity in communication mechanisms leads to a huge simplification in the external communication which can be done me wrapper functions for the XtratuM hypercalls, which conffrue the XtratuM (ARINC 653) ports to behave the way expected from FreeOSEK and allow a OSEK Com compliant interface. Things like FIFO buffers for queueing messages do not have to be implemented, since they already are implemented in the XtratuM core.

6 Conclusion

Even if the FreeOSEK run-time environment is far from being perfect, the implementation shows the feasibility of running an OSEK compliant operating system as a XtratuM run-time environment, fulfilling one of OVERSEES main missions - reuse of existing automotive applications in a security enhanced environment with minimum effort.

Furthermore, the theoretical mapping between OSEK compliant communication and ARINC653 compliant communication could be proven valid and compatible to the point where a legacy OSEK compliant application can be moved into a XtratuM run-time environment with a virtualized communication system replacing a legacy physical communications system, without the need of adapting the application itself.

The next steps in the port of FreeOSEK to XtratuM will be the cleanup of the context switch, in order to allow fully preemptive task scheduling. This is also the pre-requisite for most of the MODISTARC [6] tests which are already implemented in FreeOSEK and help to show the compliance with the OSEK/VDX specifications and finally the integration of the FreeOSEK port into the overall OVERSEE architecture Proof-of-Concept framework.
7 Acknowledgments

This paper has been produced in the context of the OVERSEE project (FP7-ICT-2009-4, Project ID 248333).

References