

Eleventh Real-Time Linux Workshop



Dresden 2009

Eleventh Real-Time Linux Workshop

Organizing committee

Prof. Hermann Härtig (TU Dresden, Germany)
Dr. Carsten Emde (OSADL, Germany)
Dr. Peter Wurmsdobler (Cambridge Consulting, UK)
Prof. Nicholas Mc Guire, (Lanzhou University, China)
Dr. Zhou Qingguo, (Lanzhou University, China)

Program committee

Roberto Bucher, RTAI Maintainer, Switzerland
Don Carr, University of Guadalajara, Mexico
Shawn Choo, Weslab LLC, Singapore
Carsten Emde, OSADL, Germany
Philippe Gerum, Xenomai Maintainer, France
Thomas Gleixner, Linutronix, Germany
Nicholas Mc Guire, Lanzhou University, China
Hermann Härtig, TU Dresden, Germany
Paolo Mantegazza, RTAI Maintainer, Italy
Peter Okech, Strathmore University, Kenya
Zhou Qingguo, Lanzhou University, China
Klaus Reichl, Thales RSS, Austria
Peter Wurmsdobler, Cambridge Consulting, UK
Bernhard Zagar, Johannes Kepler University, Austria
Thomas Fischer, FH Campus Wien, Austria

Dresden 2009

Preface

After several Real-Time Linux Workshops in Europe (Vienna 1999, Milan 2001, Valencia 2003, Lille 2005, Linz 2007), in America (Orlando 2000, Boston 2002, Guadalajara 2008), and Asia (Singapore 2004, Lanzhou 2006), the Eleventh Real-Time Linux Workshop comes to Dresden, Germany. 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.

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 proceedings and all informations on the workshop can be found on

<http://www.realtimelinuxfoundation.org/>

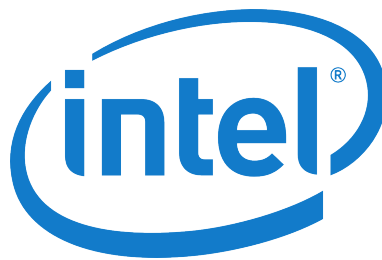
The organizing committee

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 11th 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 General	1
‘Real Time’ vs. ‘Real Fast’: How to Choose?	
<i>Paul E. McKenney</i>	1
Myths and Realities of Real-Time Linux Software Systems	
<i>Kushal Koolwal</i>	13
Joint Opportunities for Real-Time Linux and Real-Time Systems Research	
<i>Björn B. Brandenburg and James H. Anderson</i>	19
Problems regarding teaching Linux programming for Industrial realtime programmers	
<i>Fodrek Peter</i>	31
Analysis of inherent randomness of the Linux kernel	
<i>Nicholas Mc Guire, Peter Okech, Georg Schiesser</i>	35
Use of cookies in real-time system development	
<i>M. Gleixner, M. Mc Guire</i>	43
Real-Time Linux Applications	59
Linux based demonstrator of a failure tolerant numerical control system	
<i>Hans-Peter Bock and Alexander Verl</i>	59
An Open Implementation of Profibus DP	
<i>Duy Khanh Tran, Pavel Pisa and Petr Smolik</i>	65
RTAI based measuring and control over EtherCAT	
<i>Fodrek Peter and Kamensky Jaroslav</i>	73
RTAI in CAN bus based Network Control of CNC for plasma cutting	
<i>Fodrek Peter, Foltin Martin, Blaho Michal, Farkas Ludovit, Poliacik Michal, Murgas Tomas and Chren Vladimir</i>	79
On the RTAI-Lab implementation of an observer-based controller for a DC motor	
<i>Salas-Cabrera, J. C. Mayo-Maldonado, E. Y. Rendon-Fraga, E. N. Salas-Cabrera and H. Cisneros-Villegas</i>	89
Getting Linux into a train HVAC (Heating, Ventilating and Air Conditioning)	
<i>Markus Kreidl, Nicholas McGuire</i>	97
Real-Time Linux Infrastructure and Tools	105
A New Real-time Network protocol-Node Order Protocol	
<i>Li chanjuan, Nicholas Mc Guire and Zhou Qingguo</i>	105
SIL4Linux: An attempt to explore Linux satisfying SIL4 in some restrictive conditions	
<i>Lijuan Wang, Chuande Zhang, Zhangjin Wu, Nicolas Mc Guire, Qingguo Zhou and Li Su</i>	111
Finding Origins of Latencies Using Ftrace	
<i>Steven Rostedt</i>	117
Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT_RT) on AMD’s GEODE LX Platform	
<i>Kushal Koolwal</i>	131
Timing Analysis of Linux CAN Drivers	
<i>Michal Sojka, Pavel Píša</i>	147
Efficient embedded software development using QEMU	
<i>Pradyumna Sampath, Rachana Rao</i>	155
Real-Time Linux Ports	163

L4eRTL: Port of eRTL(PaRTiKle) to L4/Fiasco microkernel	
<i>Guanghai Cheng, Nicholas Mc Guire, Qingguo Zhou, Lian Li and Li Su</i>	163
Porting RT-preempt to Loongson2F	
<i>Wu Zhangjin and Nicholas Mc Guire</i>	169
Real-Time Linux Scheduling	175
Offline Scheduler	
<i>Raz Ben Yehuda</i>	175
Reservation-Based Scheduling for IRQ Threads	
<i>Luca Abeni, Nicola Manica and Luigi Palopoli</i>	179
A Flexible Scheduling Framework Supporting Multiple Programming Models with Arbitrary Semantics in Linux	
<i>Noah Watkins, Jared Straub and Douglas Niehaus</i>	187
An EDF scheduling class for the Linux kernel	
<i>Dario Faggioli, Fabio Checconi, Michael Trimarchi and Claudio Scordino</i>	197
Real-Time Linux Concepts	205
Towards Linux as a Real-Time Hypervisor	
<i>Jan Kiszka</i>	205
Requeue-PI: Making Glibc Condvars PI-Aware	
<i>Darren Hart and Dinakar Guniguntala</i>	215
Using the Realtime Preemption Patch on ARM CPUs	
<i>Jan Altenberg</i>	229
Faithful Virtualization on a Real-Time Operating System	
<i>Henning Schild, Adam Lackorzynski and Alexander Warg</i>	237
Deterministic Synchronization in Multicore Systems: the Role of RCU	
<i>Paul E. McKenney</i>	245
The ARM Fast Context Switch Extension for Linux	
<i>Gilles Chantepredrix and Richard Cochran</i>	255
XtratuM: a Hypervisor for Safety Critical Embedded Systems	
<i>M. Masmano, I. Ripoll, A. Crespo and J.J. Metge</i>	263

‘Real Time’ vs. ‘Real Fast’: How to Choose?

Paul E. McKenney

IBM Linux Technology Center

15350 SW Koll Parkway, Beaverton, OR, 97006, USA

paulmck@linux.vnet.ibm.com

Abstract

Although “real-time is not real-fast” makes a nice sound bite, it does not help developers much. This paper will provide the background needed to make a considered design choice between “real time” (getting started as quickly as possible) and “real fast” (getting done quickly once started). Of course, some developers and their users will have the natural human tendency to want both “real time” *and* “real fast”, so this paper concludes with some thoughts on how real-time Linux might continue to move in this direction.

1 Introduction

Linux[™] has made much progress in the real-time arena over the past decade, particularly given that a significant fraction of the -rt patchset [11] has now reached mainline. This naturally leads to the question of which workloads gain improved performance by running on real-time Linux. To help answer this question, we take a close look at the real-time vs. real-fast distinction in order to produce useful criteria for choosing between a real-time and non-real-time Linux, updating an earlier paper on this subject [9].

Section 2 looks at a pair of example applications in order to make a clear distinction between real-time and real-fast, Section 3 examines some factors governing the choice between real-time and real-fast, and Section 4 gives an overview of the underlying causes of real-time Linux’s additional overhead. Section 5 lays out some simple criteria to help choose between real fast and real time, Section 6 outlines some ways of approaching that nirvana of real-time *and* real-fast, and finally, Section 7 presents concluding remarks.

2 Example Applications

This section considers a pair of diverse workloads, an embedded fuel-injection application and a Linux kernel build.

2.1 Fuel Injection

This rather fanciful fuel-injection scenario evaluates real-time Linux for controlling fuel injection for a mid-sized industrial engine with a maximum rotation rate of 1500 RPM. This is slower than an automotive engine; when all else is equal, larger mechanical artifacts move more slowly than do smaller ones. We will be ignoring complicating factors such as computing how much fuel is to be injected.

If we are required to inject the fuel within one degree of top dead center (the point in the combustion cycle where the piston is at the very top of the cylinder), what jitter can be tolerated in the injection timing? 1500 RPM is 25 RPS, which in turn is 9000 degrees per second. Therefore, a tolerance of one degree turns into a tolerance of one nine-thousandth of a second, or about 111 microseconds.

Such an engine would likely have a rotational position sensor that might generate an interrupt to a device driver, which might in turn awaken a real-time control process. This process could then calcu-

```

1  for (i = 0; i < iter; i++) {
2      if (clock_gettime(CLOCK_MONOTONIC, &timestart) != 0) {
3          perror("clock_gettime 1");
4          exit(-1);
5      }
6      if (nanosleep(&timewait, NULL) != 0) {
7          perror("nanosleep");
8          exit(-1);
9      }
10     if (clock_gettime(CLOCK_MONOTONIC, &timeend) != 0) {
11         perror("clock_gettime 2");
12         exit(-1);
13     }
14     delta = (double)(timeend.tv_sec - timestart.tv_sec) * 1000000 +
15             (double)(timeend.tv_nsec - timestart.tv_nsec) / 1000.;
16     printf("iter %d delta %g\n", iter, delta - duration);
17 }

```

Table 1: Loop to Validate nanosleep()

late the time until top dead center for each cylinder, and then execute a sequence of `nanosleep()` system calls to control the timing. The code to actuate the fuel injector might be a short sequence of memory mapped I/O (MMIO) operations.

This is a classic real-time scenario. We need to do something before a deadline, and faster is most definitely *not* better. Injecting fuel too early is just as bad as injecting it too late. This situation calls for some benchmarking and validation of the `nanosleep()` system call, for example, with the code shown in Table 1. On each pass through the loop, lines 2-5 record the start time, lines 6-9 execute the `nanosleep()` system call with the specified sleep duration, lines 10-13 record the end time, and lines 14-16 compute the jitter in microseconds and print it out. This jitter is negative if the `nanosleep()` call did not sleep long enough, and positive if it slept too long.

It is important to use `clock_gettime()` with the `CLOCK_MONOTONIC` argument. The more-intuitive `CLOCK_REALTIME` argument to `clock_gettime()` means “real” as in real-world wall-clock time, *not* as in real-time. System administrators and NTP can adjust real-world wall-clock time. If you incorrectly use `gettimeofday()` or `CLOCK_REALTIME` and the systems administrator sets the time back one minute, your program might fail to actuate the fuel injectors for a full minute, which will cause the engine to stop. You have been warned!

Before executing this validation code, it is first necessary to set a real-time scheduling priority, as shown in Table 2. Line 2-5 invokes `sched_get_`

`priority_max()` to obtain the highest possible real-time (`SCHED_FIFO`) priority (or print an error) and lines 6-9 set the current process’s priority. You must have appropriate privileges to switch to a real-time priority; either super-user or `CAP_SYS_NICE`. There is also a `sched_get_priority_min()` that gives the lowest priority for a given scheduler policy, so that `sched_get_priority_min(SCHED_FIFO)` returns the lowest real-time priority, allowing applications to allocate multiple priority levels in an implementation-independent manner, if desired.

However, real-time priority is not sufficient to obtain real-time behavior, because the program might still take page faults. The fix is to lock all of the pages into memory, as shown in Table 3. The `mlockall()` system call will lock all of the process’s current memory down (`MCL_CURRENT`), and all future mappings as well (`MCL_FUTURE`).

Hardware irq handlers will preempt this code. However, the -rt Linux kernel has threaded irq handlers, which appear in the `ps` listing with names resembling `IRQ-16`. You can check their priority using the `sched_getscheduler()` system call, or by looking at the second-to-last field in `/proc/<PID>/stat`, where `<PID>` is replaced by the actual process ID of the irq thread of interest. It is possible to run your real-time application at a higher priority than that of the threaded irq handlers, but be warned that an infinite loop in such an application can lock out your irqs, which can cause your system to hang.

If you are running on a multi-core system, another way to get rid of hardware-irq latencies is to direct them to a specific CPU (also known as “hard-

```

1  sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
2  if (sp.sched_priority == -1) {
3      perror("sched_get_priority_max");
4      exit(-1);
5  }
6  if (sched_setscheduler(0, SCHED_FIFO, &sp) != 0) {
7      perror("sched_setscheduler");
8      exit(-1);
9  }

```

Table 2: Setting Real-Time Priority

```

1  if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0) {
2      perror("mlockall");
3      exit(-1);
4  }

```

Table 3: Preventing Page Faults

ware thread"). You can do this using `/proc/irq/<IRQ>/smp_affinity`, where `<IRQ>` is replaced by the irq number. You can then affinity your real-time program to some other CPU, thereby insulating your program from interrupt latency. It may be necessary to pin various kernel daemons to subsets of the CPUs as well, and the `schedutils taskset` command may be used for this purpose (though care is required, as some of the per-CPU kernel daemons really do need to run on the corresponding CPU). This has the downside of prohibiting your real-time program from using all of the CPUs, thereby limiting its performance. This technique is nonetheless useful in some cases.

It is also necessary to eliminate all firmware and hardware sources of non-real-time behavior, for example, by shutting off system-management interrupts (SMIs). Some care is needed, as some systems use SMIs to handle thermal, power, and memory-error events.

Once we have shut down these sources of non-real-time behavior, we can run the program on both a real-time and a non-real-time Linux system. In both cases, we run on a four-CPU 2.2GHz x86 system running with low-latency firmware.

Even after taking all of these precautions, the non-real-time Linux fails miserably, missing the mark by up to 3 *milliseconds*. Non-real-time Linux systems are therefore completely inappropriate for this fuel-injection application.

As one might hope, real-time Linux does much better. Nanosleep always gets within 20 *microseconds* of the requested value, and 99.999%

of the time within 13 microseconds in a run of 10,000,000 trials. Please note that the results in this paper are from a lightly tuned system. More careful configuration (for example, using dedicated CPUs) might well produce better results.

If real-time Linux can so easily meet such an aggressive real-time response goal, it should do extremely well for more typical workloads, right? This question is taken up in the next section.

2.2 Kernel Build

Since the canonical kernel-hacking workload is a kernel build, this section runs a kernel build on both a real-time and a non-real-time Linux. The script used for this purpose is shown in Table 4, featuring an 8-way parallel build of the 2.6.24 Linux kernel given an `allyesconfig` kernel configuration. The results (in decimal seconds) are shown on Table 5, and as you can see, real-time Linux is not helping this workload. The non-real-time Linux not only completed the build on average more than 15% faster than did the real-time Linux, but did so using less than half of the kernel-mode CPU time. Although there is much work in progress to narrow this gap, some of which will likely be complete before this paper is published, there is no getting around the fact that this is a large gap.

Clearly, there are jobs for which real-time Linux is not the right tool!

```

1 tar -xjf linux-2.6.24.tar.bz2
2 cd linux-2.6.24
3 make allyesconfig > /dev/null
4 time make -j8 > Make.out 2>&1
5 cd ..
6 rm -rf linux-2.6.24

```

Table 4: Kernel Build Script

		Real Fast (s)	Real Time (s)	Real-Fast Throughput Advantage
real	Raw Data	1350.4	1524.6	16.8%
		1332.7	1574.2	
		1314.5	1569.8	
	Average	1332.6	1556.2	
	Std. Dev.	14.6	22.4	
user	Raw Data	3027.2	2940.9	-1.6%
		3013.1	2982.2	
		2996.1	2971.2	
	Average	3012.2	2964.7	
	Std. Dev.	12.7	17.5	
sys	Raw Data	314.7	644.3	107.5%
		317.3	660.9	
		317.9	665.9	
	Average	316.6	657.0	
	Std. Dev.	1.4	9.2	

Table 5: Kernel Build Timings

2.3 Discussion

A key difference between these two applications is the duration of the computation. Fuel injection takes place in microseconds, while kernel builds take many seconds or minutes. In the fuel-injection scenario, we are therefore willing to sacrifice considerable performance in order to meet microsecond-scale deadlines. In contrast, even on a very fast and heavily tuned machine, handfults of milliseconds are simply irrelevant on the kernel-build timescale.

The next section takes a closer look.

3 Factors Governing Real Time and Real Fast

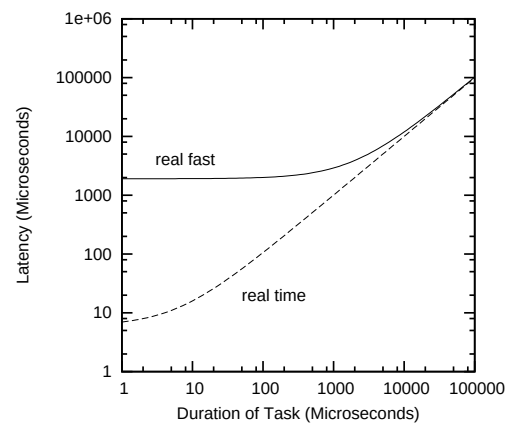


FIGURE 1: *Real Time vs. Real Fast Against Work-Unit Duration for User-Mode Computation*

In the previous section, we saw that the duration of the work is a critical factor. Although there are

a few exceptions, real-time response is usually only useful when performing very short units of work in response to a given real-time event. If the work unit is going to take three weeks to complete, then starting the work a few milliseconds late is unlikely to matter much. This relationship is displayed in Figure 1 for work-unit durations varying from one microsecond on the far left to 100 milliseconds on the far right, where smaller latencies are better. The y-axis shows the total delay, including the scheduling latency and the time required to perform the unit of work. If the unit of work to be done is quite small, a real-time system will out-perform a non-real-time system by orders of magnitude. However, when the duration of the unit of work exceeds a few tens of milliseconds, there is no discernable difference between the two.

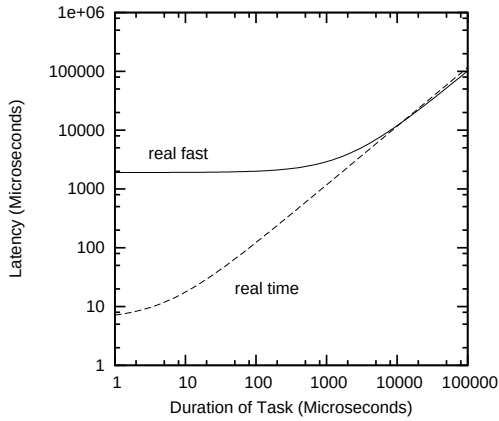


FIGURE 2: *Real Time vs. Real Fast Against Work-Unit Duration for Kernel Build*

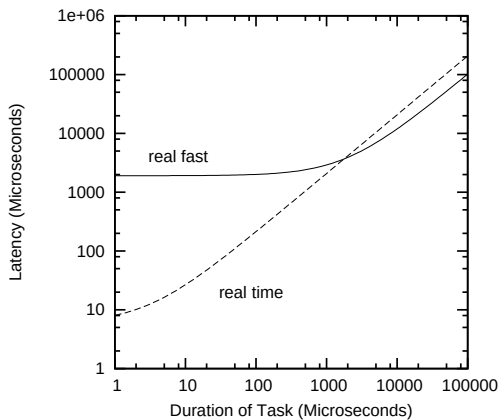


FIGURE 3: *Real Time vs. Real Fast Against Work-Unit Duration for Heavy I/O*

Furthermore, Figure 1 favors the real-time system because it assumes that the real-time system

processes the unit of work at the same rate as does the non-real-time system. However, in the kernel-build scenario discussed in Section 2.2, the non-real-time Linux built the kernel 16.78% faster than did the real-time Linux. If we factor in this real-time slowdown, the non-real-time kernel offers slightly *better* overall latency than does the real-time kernel for units of work requiring more than about ten milliseconds of processing, as shown in Figure 2. This breakeven would vary depending on the type of work. For example, floating-point processing speed would be largely independent of the type of kernel (and hence represented accurately by Figure 1), while heavy I/O workloads would likely be profoundly affected by the kernel type, as shown in Figure 3, which uses the 2-to-1 increase in kernel-build system time as an estimate of the slowdown. In this case, the crossover occurs at about one millisecond.

In addition, a concern with worst-case behavior should steer one towards real time, while a concern with throughput or efficiency should steer one towards real fast. In short, use real-time systems when the work to be done is both time-critical and of short duration. There are exceptions to this rule, but they are rare.

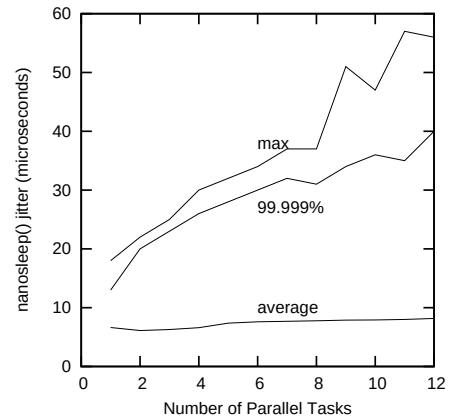


FIGURE 4: *Nanosleep Jitter With Increased Load*

CPU utilization is another critical factor. To show this, we run a number of the `nanosleep()` test programs in parallel, with each program running 100,000 calls to `nanosleep` in a loop (code shown in Table 1). Figure 4 shows the resulting average, 99.999 percentile delay, and maximum delay. The average jitter changes very little as we add tasks, which indicates that we are getting good scalability from a real-fast viewpoint. The 99.999 percentile and maximum delays tell a different story, as both increase by more than a factor of three as we go from a single task to 12 parallel tasks.

This is a key point: obtaining the best possible real-time response usually requires that the real-time system be run at low utilization. This is in direct conflict with the desire to conserve energy and reduce system footprint. In some cases, it is possible to get around this conflict by putting both real-time and non-realtime workload on the same system, but some care is still required. To illustrate this, run four parallel downloads of a kernel source tree onto the system, then unpack one of them and do a kernel build. When the `nanosleep` test program runs at maximum priority concurrently with this kernel-build workload, we see the 99.999% jitter at 59 microseconds with the worst case at 146 microseconds, which is worse than the parallel runs—but still much better than the multi-millisecond jitters from the non-real-time kernel.

Advancing technology can be expected to improve real-time Linux's ability to maintain real-time latencies in face of increasing CPU utilization, and careful choice of drivers and hardware might further improve the situation. Also, more-aggressive tuning might well produce better results. For example, this workload does not control the periodicity of the `nanosleep()` test programs, so that all 12 instances might well try to run simultaneously on a system that has but four CPUs. In real-world systems, mechanical constraints often limit the number of events that can occur simultaneously, in particular, engines are configured so that it is impossible for all cylinders to fire simultaneously. That said, sites requiring the best possible utilization will often need to sacrifice some real-time response.

Similarly, if you need to use virtualization to run multiple operating-system instances on a single server, you most likely need real fast as opposed to real time. Again, technology is advancing quite quickly in this area, especially in the embedded space, so we may soon see production-quality virtualization environments that can simultaneously support both real-time and real-fast operating systems. This is especially likely to work well if either: (1) CPUs and memory can be dedicated to a given operating instance or (2) the hypervisor (e.g., Linux with KVM) gives real-time response, but the guest operating systems need not do so. Longer term, it is quite possible that both the hypervisor and the guest OSes will offer real-time response.

4 Sources of Real-Time Overhead

The `nanosleep()` test program used the `mlockall()` system call to pin down memory in order to avoid page-fault latencies. This is great for this test program's latency, but has the side-effect of removing a chunk of memory from the VM system's control, which limits the system's ability to optimize memory usage. This can degrade throughput for some workloads.

Real-time Linux's more-aggressive preemption increases the overhead of locking and interrupts [2]. The reason for the increased locking overhead is that the corresponding critical sections may be preempted. Suppose that a given lock's critical section is preempted, and that each CPU subsequently attempts to acquire the lock. Non-real-time spinlocks would deadlock at this point. The CPUs would each spin until they acquired the lock, but the lock could not be released until the lock holder got a chance to run. Therefore, spinlock-acquisition primitives must block if they cannot immediately acquire the lock, resulting in increased overhead. The need to avoid priority inversion further increases locking overhead. This overhead results in particularly severe performance degradation for some disk-I/O benchmarks, however, real-time adaptive spinlocks may provide substantial improvements [4]. In addition, the performance of the user-level `pthread_mutex_lock()` primitives may be helped by private futexes [5].

Threaded interrupts permit long-running interrupt handlers to be preempted by high-priority real-time processes, greatly improving these processes' real-time latency. However, this adds a pair of context switches to each interrupt even in absence of preemption, one to awaken the handler thread and another when it goes back to sleep, and furthermore increases interrupt latency. Devices with very short interrupt handlers can specify `IRQF_NODELAY` in the `flags` field of their `struct irqaction` to retain the old `hardirq` behavior, but this is not acceptable for handlers that run for more than a small handful of microseconds.

Linux's $O(1)$ scheduler is extremely efficient on SMP systems, as a given CPU need only look at its own queue. This locality reduces cache thrashing, yielding extremely good performance and scalability, aside from infrequent load-balancing operations. However, real-time systems often impose the constraint that the N highest-priority runnable tasks be running at any given point in time, where N is the number of online CPUs. This constraint cannot be

met without global scheduling, which re-introduces cache thrashing and lock contention, degrading performance, especially on workloads with large numbers of runnable real-time tasks. In the future, real-time Linux is likely to partition large SMP systems, so that this expensive global scheduling constraint will apply only within each partition rather than across the entire system. Gregory Haskins's root-domain patch, which is included in the `-rt` patchset, in fact takes this approach.

Real-time Linux requires high-resolution timers with tens-of-microseconds accuracy and precision, resulting in higher-overhead timer management [3, 6]. However, these high-resolution timers are implemented on a per-CPU basis, so that it is unlikely that this overhead will be visible at the system level for most workloads. In addition, real-time Linux distinguishes between real-time “timers” and non-real-time “timeouts,” and only the real-time timers use new and more-expensive high-resolution-timer infrastructure. Timeouts, for example, TCP/IP retransmission timeouts, continue to use the original high-efficiency timer-wheel implementation, further reducing the likelihood of problematic timer over-heads.

Real-time Linux uses preemptible RCU, which has slightly higher read-side overhead than does Classic RCU [8]. However, the read-side difference is unlikely to be visible at the system level for most workloads. In contrast, preemptible RCU's update-side “grace-period” latency is significantly higher than that of RCU classic [7]. If this becomes a problem, it should be possible to expedite RCU grace period, albeit incurring additional overhead. It may then be possible to retire the Classic RCU implementation [10], but given that Classic RCU's read-side overhead is exactly zero, careful analysis will be required before such retirement can be appropriate.

In summary, the major contributors to the higher overhead of real-time Linux include increased overhead of locking, threaded interrupts, real-time task scheduling, and increased RCU grace-period latency. The next section gives some simple rules that help choose between the real fast non-real-time Linux kernel and the real-time Linux kernel.

5 How to Choose

The choice of real time vs. real fast is eased by considering the following principles:

1. Consider whether the goal is to get a lot of work done (real fast throughput), or to get a

little bit of work done in a predictable and deterministic timeframe (real-time latency).

2. Consider whether the hardware and software can accommodate the heaviest possible peak load without missing deadlines (real time), or whether occasional peak loads will degrade response times (real fast). It is common real-time practice to reserve some fraction of resources, for example, to limit CPU utilization to 50%.
3. Consider memory utilization. If your workload oversubscribes memory, so that page faults will occur, you cannot expect real-time response.
4. If you use virtualization, you are unlikely to get real-time response—though this may be changing.
5. Consider the workload. A process that executes normal instructions in user mode will incur a smaller real-time average-overhead penalty than will a process that makes heavy use of kernel services.
6. Focus on work-item completion time instead of on start time. The longer the work item's execution time, the less helpful real-time Linux will be.

The need to focus on deterministic work-item completion cannot be stressed enough. Common practice in the real-time arena is to focus on when the work-item starts, in other words, on scheduling latency. This is understandable, given the historic separation of the real-time community into RTOS and real-time application developers, both working on proprietary products. It is hoped that the advent of open-source real-time operating systems will make it easier for developers to take the more global viewpoint, focusing on the time required for the application to both start *and* finish its work. Please note that it is important to focus on the proper level of detail, for example, event-driven systems should analyze deadlines on a per-event basis.

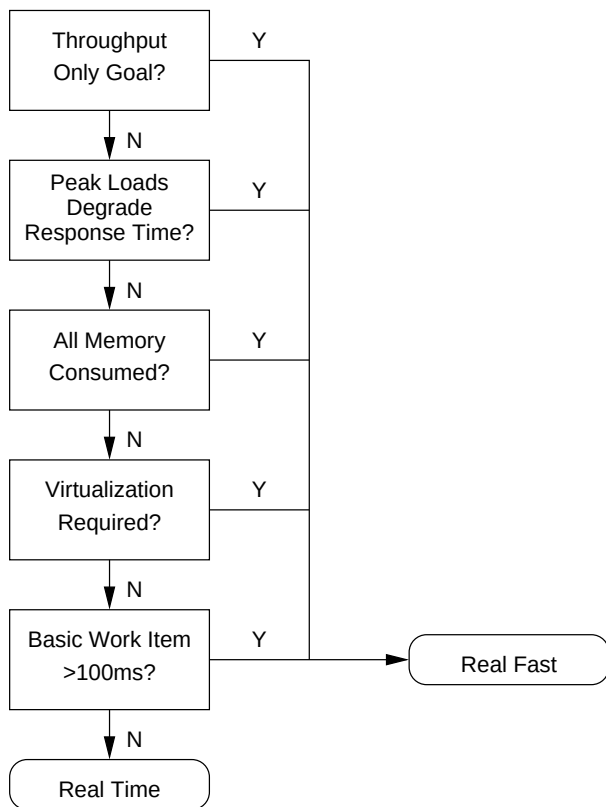


FIGURE 5: *Real Time vs. Real Fast Decision Flow*

A rough rule-of-thumb decision flow is shown in Figure 5. If you only care about throughput—the amount of work completed per unit time—then you want real fast. If cost, efficiency, or environmental concerns force you to run at high CPU utilization so that peak loads degrade response times, then you again want real fast—and as a rough rule of thumb, the more aggressive your real-time workload, the lower your CPU utilization must be. One exception to this occurs in some scientific barrier-based computations, where real-time Linux can reduce OS jitter, allowing the barrier computations to complete more quickly—and in this case, because floating point runs at full speed on real-time Linux, this is one of those rare cases where you get *both* real fast *and* real time simultaneously. If your workload will fill all of memory, then the `mlockall()` system call becomes infeasible, forcing you to either purchase more memory or allow the resulting page faults force you to go with real fast. Given the current state of the art, if you need virtualization, you are most likely in real-fast territory—though this may soon be changing, especially for carefully configured systems. Finally, if each basic item of work takes hundreds of milliseconds, any scheduling-latency benefit from real-time Linux is likely to be lost in the noise.

If you reach the real-time bubble in Figure 5, you may need some benchmarking to see which of real time or real fast works best for your workload. No benchmarking is needed to see that a workload requiring (say) 100 microseconds of processing with a 250-microsecond deadline will require real-time Linux, and there appears to be no shortage of applications of this type. In fact, it appears that real-time processing is becoming more mainstream. This is due to the fact that the availability of real-time Linux has made it easier to integrate real-time systems into enterprise workloads [1], which are starting to require increasing amounts of real-time behavior. Where traditional real-time systems were stand-alone systems, modern workloads increasingly require that the real-time systems be wired into the larger enterprise.

6 Avoiding the Need to Choose

It is only human to want everything all at once, and so it should be no surprise to encounter developers and users who want (and perhaps even need) both real-time and real-fast on the same system at the same time. It might or might not be possible to actually reach this nirvana, but incremental progress is nevertheless valuable, both by improving the -rt Linux kernel's throughput and by improving the standard kernel's real-time response.

Fortunately, there is quite a bit of work going on in this area:

1. Reduce locking overhead (ticket locks, adaptive spinlocks).
2. Optimize threading of irq handlers.
3. Eliminate reader-writer lock bottlenecks, for example, reducing the real-time performance penalty for multiple communications streams (numerous patches).
4. Reduce the real-time performance penalty for mass-storage I/O. (This becomes more urgent with the advent of solid-state disks.)
5. Reduce the preemptable RCU grace-period latency penalty. As of August 2009, a prototype preemptable RCU implementation nearly eliminates both the read-side and update-side performance penalties, but removes the ability to do CPU hotplug operations (this bug will be fixed).

		Real Fast (s)	Real Time (s)	Real-Fast Throughput Advantage
real	Raw Data	821.9	894.2	9.1%
		830.1	909.7	
		834.0	908.4	
	Average	828.7	904.1	
	Std. Dev.	5.0	7.0	
user	Raw Data	2331.7	2509.0	7.4%
		2343.1	2512.3	
		2338.9	2510.2	
	Average	2337.9	2510.5	
	Std. Dev.	4.7	1.4	
sys	Raw Data	316.5	415.0	33.2%
		326.7	438.3	
		327.0	438.9	
	Average	323.4	430.7	
	Std. Dev.	4.9	11.1	

Table 6: Kernel Build Timings, One Year On

	Old	New
real	16.8%	9.1%
user	-1.6%	7.4%
sys	107.5%	33.2%

Table 7: Real-Fast Throughput Advantage, Then and Now

- Where eliminating throughput penalties is infeasible, adjust implementation so that performance penalties are incurred only when there are actually real-time tasks in the system.

So we are indeed moving towards the real-time *and* real-fast nirvana, as shown in Table 6, which shows the kernel-build benchmark running on a more recent pair of mainline and -rt kernels. The real-fast kernel is now less than 10% faster (compared to 15%), and consumes only about one-third more kernel-mode CPU time (compared to more than twice as much), with the real-fast throughput advantage summarized in Table 7. Please note that the absolute timings are not directly comparable to the timings in Table 5 on page 4 due to compiler and kernel-configuration differences, even though the hardware and the kernel being built are believed to be identical.

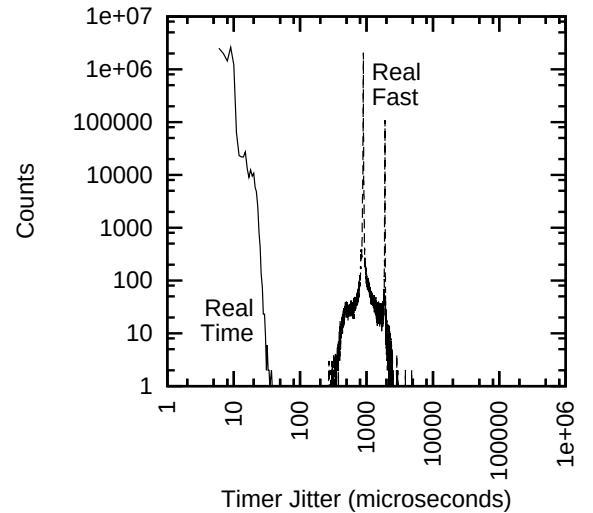


FIGURE 6: Real Time vs. Real Fast Timer Jitter

Of course continued attention to latencies will

also be required, as that which is not measured tends to degrade. A histogram of the timer jitter of more-recent kernels is shown in Figure 6, with the real-fast kernel still missing the mark by up to 100 *milliseconds*, while the real-time kernel's worst-case measured jitter remains in the tens of microseconds.

7 Concluding Remarks

Given proper hardware and software configuration, sufficiently fast hardware, and proper use of -rt Linux's real-time facilities by the real-time application, response times in the tens of microseconds are possible. However, there still are some throughput penalties incurred when attaining this level of real-time response, so if you remember only one thing from this paper, let it be this: "use the right tool for the job!!"

Ongoing work to reduce the overhead of real-time Linux will hopefully reduce the performance penalty imposed by the real-time kernel, which will in turn make real-time Linux the right tool for a greater variety of workloads. It will also likely be possible to further optimize some of the real-time implementations. In any case, real-time Linux and real-time applications both promise to remain an exciting and challenging areas for some time to come.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Dietrich, K.R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. We all owe Will Schmidt and Phil Estes a debt of gratitude for helping to render this paper human-readable. I am grateful to Paul Clarke for the use of his machines, and the ABAT team for providing easy access to these machines. Finally, I owe thanks to Kathy Bennett for her support of this effort.

Legal Statement

This work represents the views of the authors and does not necessarily represent the view of IBM.

Linux is a copyright of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

References

- [1] BERRY, R. F., MCKENNEY, P. E., AND PARR, F. N. Responsive systems: An introduction. *IBM Systems Journal* 47, 2 (April 2008), 197–206.
- [2] CORBET, J. Approaches to realtime Linux. Available: <http://lwn.net/Articles/106010/> [Viewed March 25, 2008], October 2004.
- [3] CORBET, J. A new approach to kernel timers. Available: <http://lwn.net/Articles/152436/> [Viewed April 14, 2008], September 2005.
- [4] CORBET, J. Realtime adaptive locks. Available: <http://lwn.net/Articles/271817/> [Viewed April 14, 2008], March 2008.
- [5] DUMAZET, E. [PATCH] FUTEX : new PRIVATE futexes. Available: <http://lkml.org/lkml/2007/4/5/236> [Viewed April 18, 2008], April 2007.
- [6] GLEIXNER, T., AND MOLNAR, I. [announce] ktimers subsystem. Available: <http://lwn.net/Articles/152363/> [Viewed April 14, 2008], September 2005.
- [7] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (May 2008), 221–236. Available: <http://www.research.ibm.com/journal/sj/472/guniguntala.pdf> [Viewed April 24, 2008].
- [8] MCKENNEY, P. E. The design of preemptible read-copy-update. Available: <http://lwn.net/Articles/253651/> [Viewed October 25, 2007], October 2007.
- [9] MCKENNEY, P. E. Real time vs. real fast: how to choose? In *Ottawa Linux Symposium* (July 2008), pp. v2 57–65. Available: <http://ols.fedoraproject.org/OLS/Reprints-2008/mckenney-reprint.pdf> [Viewed August 22, 2008].
- [10] MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184

- <http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf> [Viewed January 1, 2007].
- [11] MOLNAR, I. Index of /mingo/realtime-preempt. Available: <http://www.kernel.org/pub/linux/kernel/projects/rt/> [Viewed February 15, 2005], February 2005.

Myths and Realities of Real-Time Linux Software Systems

Kushal Koolwal

VersaLogic Corporation

3888 Stewart Road, Eugene, OR 97402 USA

kushalk@versalogic.com

Abstract

This document addresses some of the differences between real-time and general purpose operating systems. This includes an analysis of several common misconceptions including performance issues, latency, hard vs. soft real-time systems, programming APIs and the software kernel. In addition, there are suggestions for implementation options to convert Linux to a real-time operating system.

1 What is real-time system?

According to the Real-time Computing FAQ[1], a real-time operating system (RTOS) is defined as follows:

“A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints are not met, system failure is said to have occurred.”

In other words, a system should be able to perform computations deterministically - an important concept in real-time systems. A system is said to be deterministic if it's possible to predict the timings of its computation precisely, and computations are logically correct. For example, a real-time system should give the result of “2+2” not only as “4”, but it should also produce the correct result within a specified time range.

1.1 Analogy for real-time systems

To understand the concept of a real-time system, consider the example of an automobile airbag system, one of the most critical systems in a modern car. An airbag should be deployed within a particular time interval (for example, within 1 second¹)

after the sensors in the car detect a collision.

A car equipped with a real-time computer system will always guarantee that the airbag will be deployed within 1 second, no matter what happens (see worst-case scenario below). The real-time system will put all the non-critical tasks, such as powering the stereo, switching on the air-conditioning, activating cruise control, etc. on hold and will immediately deploy the airbag as soon as it receives the signal from the sensors.

If a car is equipped with a non real-time system, however, there is no guarantee that the airbag will be deployed exactly within 1 second after the sensor detects a collision. Such a non real-time system might deploy the airbag after finishing the request to activate the cruise control, which happened to come before the request to deploy the airbag. An airbag system that deploys even 0.1 second later than the expected time is as bad as not deploying at all. Why? Because it might already be too late to save the life of the passenger. In fact, in real-time systems, a difference of even 0.000001 second may be important.

2 Metrics of real-time

A number of parameters are required to quantify real-time systems. To understand these parameters, it is important to define the term “latency.”

¹Usually the time interval is much less than 1 sec., but we use 1 sec. for simplicity.

Latency: In computing, latency is “the time that elapses between a stimulus and the response to it[2].”

Using this definition and our airbag example, latency in real-time systems is defined as the time elapsed between the triggering of an event (signal sent by sensors), requesting a particular task (deploying the airbag), and the actual task being executed (airbag deployed). In other words, it is the delay between an action and a response to that particular action.

Following are some important metrics that help us to quantify a real-time system:

a) Interrupt Latency: The time elapsed between the generation of an interrupt and the start of the execution of the corresponding interrupt handler.

Example: When a hardware device performs a task, it generates an interrupt (an IRQ). This interrupt has the information about the task to be performed and about the interrupt handler (ISR) to be executed. The interrupt handler then performs the particular task.

b) Scheduling Latency: According to Clark Williams of RedHat[3], “it is the time between a wakeup (the stimulus) signaling that an event has occurred and the kernel scheduler getting an opportunity to schedule the thread that is waiting for the wakeup to occur (the response). Wakeups can be caused by hardware interrupts, or by other threads.” Scheduling latency is also known as **task-response latency** or **dispatch latency**.

An extension of this concept is the **priority inversion**. Here is one example: Sometimes while scheduling different threads (or tasks) a situation might arise in which a higher priority thread might be waiting to obtain a resource that is held by a lower priority thread. Here a kernel has to reach a safe point, named a **rescheduling point**[4], before it can give the resource to the higher priority task. In fact, one of the failures of the Mars Pathfinder was caused by an unbounded priority inversion[5].

Example: Suppose your car (a low priority task) enters a busy traffic intersection (CPU of the system) and you guess that you might be able to cross the intersection (use the CPU to process), but then you realize that all of the traffic in front of you has abruptly come to a stop (an unexpected event occurs in the system), and you are stuck right in the middle of the intersection. Suddenly you hear the siren of a fire truck (a high priority task) heading toward

you to cross the intersection (use CPU to process). Unfortunately, you cannot move, as you are stuck, and the fire truck also has to stop until you are out of its way (a high priority task has to wait for a low priority task). As a result, a building might continue burning and lives may be lost.

In the real-life scenario described above, the fire truck has to wait, but in a real-time system a real-time kernel can create paths (roads) on-the-fly which allows a high priority task (fire truck) to bypass a low priority task (traffic jam). These are named rescheduling points.

The figure below illustrates Interrupt Latency and Scheduling Latency[4].

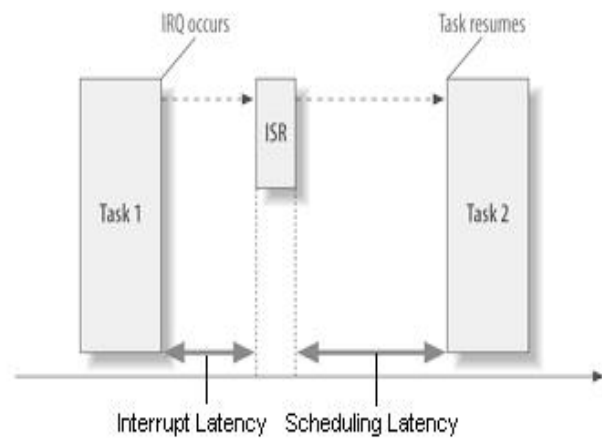


FIGURE 1: *Interrupt and Scheduling Latencies*

c) Worst-case Latency: This is defined as the maximum time that can lapse before the desired event occurs.

Worst-case refers to the condition when the system is under heavy load - more CPU load and I/O operations occurring than are typical for the particular system. When we talk about real-time it is very important that we define our results not only on an average basis, but in a worst-case scenario as well.

Example: The 1 second (maximum) limit discussed in the airbag example is the worst-case scenario. It means that, on average, the airbag will be deployed in, say, 0.3 seconds, but in a worst-case scenario (system under heavy load), the airbag will be deployed within 1 second.

Latency requirements: In general, a real-time system should be able to give an average latency in the range of $10\text{-}20^2 \mu\text{s}$. Under a worst-case scenario,

²1 second = 1000 milliseconds = 1000000 μs .

³These latency numbers are not set in stone. It really depends upon your needs, the underlying application and the system. We found these numbers based on our research [10] and [14]. Also, these numbers are with respect to Linux OS. Your mileage

the latency should be within a few hundred microseconds³.

3 Hard vs. Soft real-time

Most people use the terms hard real-time and soft real-time loosely. However, it is essential to distinguish between the two terms.

Hard real-time: A system that can meet the desired deadlines at all times (100 percent), even under a worst-case system load. In hard real-time systems, missing a deadline, even a single time, can have fatal consequences. Some examples are airbag systems, ABS (anti-lock braking systems), missile systems, aircraft controls, etc. Hard real-time systems are used in cases where a particular task, usually involving life safety issues, needs to be performed within a particular time frame, otherwise a catastrophic event will occur.

Soft real-time: A system that can meet the desired deadlines on average. Some examples are online audio/video broadcasts, stock market software, cell phone display, etc. In other words, a soft real-time system will give reduced average latency but not a guaranteed maximum response time. Due to the lack of deadline support, a soft RTOS is risky to use for industrial control and robotics[6].

A soft RTOS can miss a few deadlines (such as dropping a few frames in a video application) without failing the overall system. A hard RTOS cannot afford to miss a single deadline at any point because of its mission critical application nature (such as deploying an airbag).

4 GPOS vs. RTOS

Since the basics of an RTOS have been discussed, now is a good time to compare some of the important characteristics of an RTOS to those of a general-purpose operating system (GPOS). A GPOS is used by average computer users for day-to-day activities like checking e-mail, typing documents, listening to music, watching videos, etc. The table below compares GPOS and RTOS systems:

Parameters	GPOS	RTOS
Primary function	Performance	Responsiveness
Time bounded	No	Yes
Guaranteed Calculation (Worst-case)	No	Yes
Calculation Example (Max = Worst-case)	Avg: 5 secs (99.9%) Max: 22 secs (0.1%) ⁰ Deadline: 20 secs	Avg: 7 secs Max: 15 secs Deadline: 20 secs
Examples	WinXP, WinXPe, Linux, DOS, MacOS	WinCE, Vx-Works, QNX, Linux (RT patch)

5 Myths of real-time: Real-time = Performance?

It is not surprising how often people confuse real-time with performance. A real-time system does not mean faster performance. In fact, a real-time system may or may not lead to deterioration in performance. Application developers who think they need a real-time system for better performance of their application can actually get the performance they require by simply upgrading their hardware with faster processors, RAM, high speed buses, etc.

According to Yaghmour[4], “Real-time deals with guarantees, not with raw speed.” However, a quality RTOS will still deliver decent overall throughput (performance) but can sacrifice throughput for being deterministic (or predictable). In fact, many benchmarking studies on the performance of a real-time kernel against a standard kernel have found that overall performance does not decrease significantly[7].

6 Who needs real-time?

As mentioned above, an RTOS does not necessarily mean a faster machine. Not everyone needs a real-time system. In fact, almost 90 percent of embedded applications do not require a hard real-time system. It depends upon the specific needs and the underlying application. The key to determining whether

will vary if you use a different OS or a different real-time approach (see below).

⁰22 secs is not acceptable in this case and hence it does not qualify as a hard RTOS.

you really need a real-time system is to ask the right questions:

a) What type of latencies are required? What is the maximum latency that your application can tolerate without failing the system?

b) Does the application involve any life-safety issues? Or, in other words, does it need to perform a certain task within a particular deadline for the task to be of any value?

Example: A car manufacturer is designing a new car and requires the airbag system to deploy within 50 μ s, at any given cost, after collision has occurred, or else the application (deploying of airbag) won't be of any value.

c) Is it absolutely necessary to process all of the data in the system, or is it acceptable for a small amount of data to remain unprocessed?

d) Is the application dependent upon an external device that needs a response within a particular time frame or else the entire system will fail?

The following table lists examples of some hard and soft real-time applications by industry:

Industry	Hard Real-time	Soft Real-time
Aerospace	Fault detection, Aircraft control	Display Screen
Finance	ATMs	Stock Market Websites
Industrial	Robotics, DSP	Temperature monitoring
Medical	CT Scan, MRI, fMRI	Blood extraction, Surgical Assist.
Communications	QoS	Audio/Video streaming, Networking, Camcorders

7 Real-time in Linux

Traditionally, Linux was designed to be a GPOS. However, many projects have been started to convert the Linux operating system into an RTOS: RTLinux (Wind River), Xenomai, RTAI, RT Preemption Patch, etc. A complete list of implementations is available on the Real Time Linux Foundation Web site[8].

There are two fundamental approaches to making the Linux kernel real-time:

a) Improve the kernel preemption itself; that is, make the parts of the code as preemptible as possible.

b) Introduce a new software layer (a sub-kernel) beneath (or along with) the actual Linux kernel, called the Co-kernel approach.

It is beyond the scope of this paper to talk about all the different types of Linux RTOS implementations. The approaches mentioned above, except the RT Preemption (PREEMPT_RT) approach, are fundamentally based on the sub-kernel concept, whereas the (PREEMPT_RT) approach is purely based on improving kernel preemption. The RT Preemption approach is described below.

7.1 Advantages of the PREEMPT_RT approach

One of the primary advantages of this approach is the availability of the same APIs that exist in the non real-time kernel. Application programmers can use the existing POSIX compliant APIs (available under Linux) that they have already used in writing their applications[9]. Developers don't need to learn additional APIs that might be introduced if they were to go with a Co-kernel approach.

7.2 Is the PREEMPT_RT patch a hard real-time or soft real-time system?

There has been a lot of debate in the form of technical discussions over this issue. This document will try to describe the current scenario. When a generic Linux kernel is downloaded from www.kernel.org, or when a Linux distribution like Debian, Fedora, Ubuntu, etc. that comes with a pre-packaged kernel is used, it is not a fully real-time kernel. However, the kernel can be configured to a soft real-time kernel by choosing the "Preemptible Kernel" (PREEMPT_DESKTOP) option under the kernel configuration menu.

Moreover, if the soft real-time kernel is not desirable, then the Linux kernel can be made into a generic real-time kernel[10] by applying Ingo Molnar's PREEMPT_RT patch[11]. This patch makes a regular Linux OS (kernel) into a fully Preemptible OS - an OS that has the capability to stop/hold a low priority task in favor of a high priority task. This patch is not part of the mainline kernel as of yet. However, the team behind this project is constantly making efforts to make it a part of the mainline kernel. In fact, some parts of the PREEMPT_RT patch

have already made it into the mainline kernel. For example, the “Voluntary Kernel Preemption” (PREEMPT_VOLUNTARY) option in the mainline kernel is the result of one such effort.

As mentioned above, it is still not clear whether the Linux kernel with the PREEMPT_RT patch applied can be called a hard real-time system or not. According to Building Embedded Linux Systems[4], “For most applications that need real-time determinism, the RT-patched Linux kernel provides adequate service. But for those real-time applications that need more than low latencies and actually have a system that can be vigorously audited against bugs, the Linux kernel, with or without the RT patch, is not sufficient.”

7.3 Overview of PREEMPT_RT patch

What are the changes that are made to the mainline kernel with the PREEMPT_RT patch?

Giving a detailed technical explanation of every change made to the kernel code is beyond the scope of this document. However, here is a high-level overview of the changes made by this patch:

- a) Converts all Interrupt Service Routines (ISRs) to kernel threads, known as “Threaded Interrupt Service Routines.”
- b) Replaces kernel “spinlocks” with “mutexes” that support priority inheritance and are preemptive.
- c) Adds High Resolution Timer (HRT) support, which allows the timers to operate at a resolution of 1s.
- d) Disables unbounded priority inversion.

(For more technical details about the above issues, please refer to [12] and [13].)

7.4 Device drivers and the PREEMPT_RT patch

A common question from users is the following: “Is there a real-time driver for my ethernet controller that enables me to use the real-time Linux kernel (PREEMPT_RT) patch?” At present, most of the drivers available under the Linux kernel tree are quite robust and ready to be used with the real-time kernel (PREEMPT_RT patch). They are tightly written and highly scrutinized, so usually there is no room for further improvement. If any change has to be

made in order to further reduce the latency, then the change has to be made to the sub-system.

For example, to reduce network operation related latency, the change has to be made in the networking stack (TCP/IP protocol, etc.) and not necessarily in the network card driver. It would still be good practice to have your device driver inspected by a real-time expert to see if there is further possibility to tighten the code to make it more preemptible. The real improvement will still come from making changes to the core code (networking stack), as explained above. However, keep in mind that if you decide to do this, the reduced latency will come at the cost of performance.

8 Conclusion

While this document presents an overview of the issues and concerns commonly experienced by deterministic computer systems, it is not intended to be a substitute for a detailed analysis of your specific system requirements or operational priorities.

In summary, a thorough examination of the capabilities and limitations of using Linux as a real-time operating system requires a detailed analysis of your specific system performance and application requirements. These include, but are not limited to, an assessment of the mission critical nature of your system, acceptable risks and the level and quality of performance that is expected.

References

- [1] <http://www.faqs.org/faqs/realtime-computing/faq/>
- [2] <http://dictionary.reference.com/browse/latency>
- [3] <http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf>
- [4] *Building Embedded Linux Systems*, 2nd Edition, Karim Yaghmour, Jonathan Masters and Gilad Ben-Yossef, Aug. 2008, O'REILLY MEDIA
- [5] <http://catless.ncl.ac.uk/Risks/19.54.html#subj6>
- [6] *Operating System Concepts*, 6th Edition, Silberschatz, A., Galvin, P. B., and Gagne, G., 2001, JOHN WILEY AND SONS
- [7] <http://dslab.lzu.edu.cn:8080/docs/publications/Siro.pdf>
- [8] <http://www.reallinuxfoundation.org/>

- [9] http://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions#List_of_realtime_APIs.3F
- [10] <http://www.ibm.com/developerworks/linux/library/l-real-time-linux/>
- [11] <http://www.kernel.org/pub/linux/kernel/projects/rt/>
- [12] <http://lwn.net/Articles/146861/>
- [13] <http://www.linuxinsight.com/files/ols2007/rostedt-reprint.pdf>
- [14] <http://www.mvista.com/download/videos/playvideo.php?video=16>

Joint Opportunities for Real-Time Linux and Real-Time Systems Research

Björn B. Brandenburg

Department of Computer Science
The University of North Carolina at Chapel Hill
Campus Box #3175, Sitterson Hall, Chapel Hill, NC 27599-3175
bbb@cs.unc.edu

James H. Anderson

Department of Computer Science
The University of North Carolina at Chapel Hill
Campus Box #3175, Sitterson Hall, Chapel Hill, NC 27599-3175
anderson@cs.unc.edu

Abstract

Based on lessons learned in developing and maintaining LITMUS^{RT}, arguments in favor of an intensified collaboration between the academic and the open-source real-time communities are presented, and several ways in which ongoing efforts in these two communities may benefit each other are outlined. Some (unfortunately) commonly-encountered sources of friction and mutual misconceptions, which result from differing backgrounds and objectives in these two communities, are exposed, with the goal of finding common ground. Further, a “wish list” is presented of possible additions and changes to PREEMPT-RT that would enhance Linux’s viability as the “platform of choice” for real-time-systems research. These improvements are substantiated by examining EDF-HSB, a candidate algorithm for earliest-deadline-first (EDF) scheduling support in Linux that integrates hard and soft real-time guarantees as well as best-effort scheduling with bandwidth reservations.

1 Introduction

Due to its openness and its applicability across a wide range of hardware platforms, Linux has been popular with researchers for many years [47]. In research on real-time systems in particular, Linux has served as a starting point for numerous projects [19, 21, 24, 33, 37, 44, 46, 48]. Given these strong ties to real-time-systems research, one could reasonably expect Linux to provide many real-time-related features that are based on scheduling and synchronization algorithms that are provably among the best approaches (from an analytical standpoint). Sadly, this is not the case.

Linux’s central real-time features focus on low interrupt latency, static priority scheduling, and basic priority inheritance [30, 41]; while undoubt-

edly important, these features reflect the state of the art of over 20 years ago. Moreover, recently-added, more advanced features, such as hierarchical scheduling and bandwidth limiting, are *not* rooted in analytically-sound foundations [6]. In essence, there is a severe disconnect between academia and Linux technologists—few (if any) of the advances made in projects based on Linux have found their way back to Linux itself.

This lack of communication and exchange of ideas is very regrettable when in fact academics and Linux developers increasingly need each other. With ever-increasing hardware complexity, academic researchers critically depend on a strong foundation—building a fully-featured OS from scratch is prohibitively expensive. Given Linux’s extensive multi-processor support (since version 2.6) and the widely-

acknowledged, yet still growing importance of multi-core architectures, Linux will likely remain the primary platform for applied systems research in the foreseeable future. At the same time, multicore architectures pose complex multiprocessor resource-allocation and optimization challenges for which “trial-and-error” design yields only diminishing returns. While slow-moving at times, academia can offer insights and advice on fundamental algorithmic trade-offs that are only rarely exposed on mailing lists.

Hence, we strongly believe that improved collaboration between the Linux and academic real-time communities would be mutually beneficial. As a first step towards that goal, this paper comments on some of the common barriers and misconceptions that have hindered cooperation in the past, and examines real-time concepts and interfaces that would benefit both researchers and practitioners if implemented in Linux.

Defining “Real-Time Linux.” Many designs—often quite different from one another—have been labeled “real-time Linux” in the past.

Real-time Linux variants can generally be categorized into two groups. In a *native* design, the Linux kernel is the only kernel present and responsible for meeting real-time requirements, and real-time tasks are regular Linux processes. In contrast, in a *para-virtualized* design, a specialized (hard) real-time-capable microkernel (or hypervisor) is conceptually inserted between Linux and the actual hardware. Such implementations follow a classical microkernel design [28] in which Linux takes over the role of an OS server and is scheduled as a background, non-real-time thread by the microkernel (e.g., [24, 48]). Real-time tasks are specialized threads (i.e., not Linux processes) that are directly dispatched by the microkernel.

Because of Linux’s beginnings as a traditional monolithic kernel with (in the context of real-time-systems design) excessively-long non-preemptive sections, early Linux-based real-time systems were commonly based on para-virtualization. There are two key advantages to such a design. First, low interrupt latencies can be guaranteed to real-time tasks regardless of any deficiencies in the Linux kernel. Second, only (relatively) small changes to the Linux kernel are required, which means that integrating improvements made in newer Linux versions is (relatively) easy. A good example for these benefits is the L4Linux/Fiasco system [24], in which Linux is para-virtualized on top of Fiasco, TU Dresden’s L4-based microkernel: initially released in 1996 for

Linux 1.3.94 [25], L4Linux is still reliably tracking the latest Linux kernel versions in 2009.

Unfortunately, para-virtualized real-time Linux variants suffer from two significant drawbacks: **(i)** as mentioned above, real-time tasks are not Linux processes and **(ii)** there is only little benefit to mainline Linux. Regarding (i), to ensure predictably low interrupt latencies, real-time tasks usually execute directly on top of the microkernel and cannot make use of Linux services (such as device drivers, POSIX IPC, synchronization primitives, filesystems, etc.). This limitation is fundamental since para-virtualization does not improve Linux’s real-time capabilities; rather, it enables real-time tasks to safely co-exist with the Linux kernel. (One common way to enable communication between real-time threads and Linux processes in such systems is the use of non-blocking queues and buffers [3, 4].) Compared to ordinary Linux-based development, para-virtualized designs pose a greater engineering challenge due to the unusual and more complex system architecture and the restricted runtime environment for real-time tasks.

Regarding (ii), para-virtualization fundamentally takes the existing Linux kernel as a given; it is focused on engineering a solution that circumvents Linux’s deficiencies rather than solving them. As such, the capabilities of the Linux kernel are not improved and only few (if any) patches are conveyed back to mainline Linux. As an example, consider RTLinux [48]: despite both its use in industrial embedded systems and having been supported commercially for more than ten years, it has had very little influence on mainline Linux. In some sense, para-virtualization generates a “two-class society:” first-class applications, i.e., those that are important enough to warrant the effort of developing a custom para-virtualized system, can rely on reliable real-time properties; however, second-class applications (everything else) cannot. Note that the latter includes desktop applications such as media playback.

To summarize, while para-virtualization may be the only feasible (Linux-based) design for applications with very stringent latency requirements (e.g., engine control software), a native design is generally preferable for the vast majority of applications *if timing constraints can be met*: from the point of view of commercial users of real-time Linux, such a design can result in significant time-to-market and cost savings (due to a familiar development environment, greater talent pool, code re-use, etc.), and from the point of view of the Linux community, there is an increased chance that improvements are contributed

¹We acknowledge that microkernel designs can also be attractive for non-technical reasons, e.g., they allow circumvention of

back and that they can be integrated into mainline Linux.¹

Hence, we believe that the goal of Linux-based real-time research should be to improve native real-time Linux so that it becomes a viable solution for the widest range of applications possible. Consequently, we do not consider para-virtualized designs in the rest of this paper, and our proposals concerning “real-time Linux” should mostly be understood in the context of the PREEMPT-RT patch set [39].

2 Sources of Friction

With their foundations firmly grounded in openness and the free flow of ideas, the open source community and academia have many more commonalities than differences. Nonetheless, certain issues have repeatedly caused some degree of confusion and mutual misunderstanding. The following list of topics is neither exhaustive nor can it be “representative” of the points of view encountered in such large and diverse communities. However, we hope that future cooperation can be improved by pointing out how differences in background and objectives have led to misconceptions in previous interactions between these two communities.²

Low latency vs. predictability. A recurring debate is the topic of “real-time vs. real-fast” [34]: in practice, real-time constraints are often phrased in terms of latency requirements, whereas the academic definition of a real-time system does not even mention latency.

Formally, real-time systems are distinguished from other computing systems by having a *dual* notion of correctness: proper behavior depends not only on *logical* correctness (“it does the right thing”), but also on *temporal* correctness (“at the right time”). Note that, as with all formal properties, both correctness criteria require rigorous proof. Hence, real-time research is the study of systems with *provable* temporal properties: systems for which we can predict runtime behavior with mathematical certainty (if all assumptions are met).

In practice, it seems that “real-time” is often equated with an OS’s ability to transfer control to real-time tasks as quickly as possible, maybe together with a bound on worst-case latency. This is an oversimplification of the notion of “real-time computing.” Obviously, latency crucially impacts what kind of

guarantees can be made, but even a system with negligible observed latency is not truly a real-time system if it is constructed from algorithms that do not lend themselves to formal analysis; such a system is merely “fast.”

Another frequently-encountered difference is that, in practice, system design is often focused on supporting *one* dedicated real-time task, whereas a proper real-time system should support any number of such tasks (as long as meeting their collective timing constraints is feasible³).

If operating system design is predominantly focused on providing low latency to *the* highest-priority task (as opposed to meeting a set of specified timing constraints and enforcing budgets—see Sec. 3), then composing a real-time system from multiple tasks becomes unnecessarily complicated, if not impossible (e.g., given possible starvation, should a real-time web service’s priority be higher or lower than the network stack that it depends on?). As a result, design patterns such as “select/poll-based master loops” and “one real-time task per core” are often favored even though designs that employ multiple co-operating real-time tasks would be more desirable in terms of reliability and achievable resource utilization (i.e., more could be done with fewer processors).

Of course, we do not argue that real-time Linux should not provide low worst-case latencies. However, we do argue in favor of phrasing the discussion in terms of deadline and jitter requirements of supported workloads instead of raw (and often, it seems, somewhat arbitrary) latency numbers. Further, we believe that the kernel should support a more expressive task model and make scheduling decisions at runtime based on task constraints (as opposed to relying on a single static priority value—see Sec. 3).

Proofs and verification. As mentioned above, claims of real-time properties require an accompanying proof of correctness. Of course, in practice, even proving logical correctness is done rarely at best. Ignoring the question of whether it is beneficial to verify every program, it is probably safe to assume that many embedded development efforts are under sufficiently-pressing time-to-market and cost constraints that formal verification would likely be skipped even if tools for computer-aided correctness proofs were readily available. Worse, in reality, we are, in all likelihood, decades away from being able to formally verify software systems of Linux’s com-

Linux’s licensing requirements by placing proprietary code in a separate address space. Such concerns are beyond the scope of this paper.

²The authors acknowledge that their point of view is naturally biased towards academia.

³Establishing the feasibility of timing constraints lies at the core of real-time analysis. A thorough discussion of feasibility is beyond the scope of this paper; however, the interested reader is strongly encouraged to explore this important topic [31].

plexity.

Does this make real-time research “impractical”? Does it mean that real-time-system design practices in general, and Linux in particular, have little to gain from current research? Certainly not!

Even though Linux itself is not formally verified,⁴ it is built around concepts and algorithms that have been thoroughly explored in research papers. Examples are numerous: from data structures such as red-black trees and heaps to TCP’s congestion-avoidance protocol, from pattern-matching algorithms to concepts such as fairness, from encryption to mandatory access control, Linux’s internals rely on building blocks for which desirable properties have been proven in the research literature. Why should Linux’s real-time features be any different?

We argue not that every system has to be proven to be correct (which, though desirable, is clearly impractical at this point), but that every system should be *composed* from parts that *can be* and *have been* proven to be correct. The implemented algorithms should never be the weakest link.

Overheads. There has also been some controversy regarding the treatment of overheads in the academic literature. Specifically, concerns have been voiced that published scheduling analysis is not applicable to real systems because overheads are routinely assumed to be negligible.

Assuming overheads to be negligible is indeed standard practice in real-time analysis and may appear to be an oversimplification on first sight. However, this is not the case. In work on real-time resource allocation, overheads are often ignored in the primary analysis, as this reduces notational clutter that can obscure the results. Once such analysis has been obtained, overheads can be accounted for by inflating task execution costs. Thus, what we see here is actually a classic separation of concerns: establish an algorithm’s essential properties first, and then factor in overheads later.

Mode of operation. Last, but not least, there is a fundamental difference in how academia and the Linux community function.

The modus operandi of the Linux community is captured well by the widely-known mantra “release early, release often.” Developers are encouraged to provide early versions of their work to the *Linux kernel mailing list* (LKML) to solicit feedback and advice on what is often still a rough and unfinished prototype, and participants may engage in lively discussions before much developer effort has been

wasted on potentially futile approaches—but also, sometimes, before the problem and possible solutions are sufficiently well understood. Many projects go through frequent and fast iterations, and the resulting traffic on LKML can easily become overwhelming to observers.

In contrast, the academic culture is mostly focused on work that is, to some degree, finished and polished. The bar to publication can be high and the academic peer reviewing process is an anonymous, mostly one-way conversation, and its progress is usually measured in weeks and months, not hours or days. Traditionally, conferences provide the main venue for discussion, and as such, convergence on key research directions tends to happen through in-person exchanges. In discussions, academics tend to shy away from quick and categorical answers.

With such diverging cultural norms, mutual misunderstanding is all but guaranteed to occur. The intensity of the Linux community’s interactions can appear daunting and the brief messages exchanged during quick back-and-forth discussions may come across as being ill-prepared. Conversely, academics can appear to be slow moving, doubtful, and hesitant, characteristics that, when combined, are easily mistaken as disinterest. In the interest of improved collaboration between these communities, it is best to be aware of these differences.

3 Extending Real-Time Linux

Having touched on some of the issues that can prevent effective collaboration in the previous section, we next discuss some design choices, algorithmic concepts, and research directions that we believe could serve as starting point for enhancing real-time Linux.

3.1 Design Space

First, we discuss how real-time Linux could be structurally improved to both better support applied research and adapt to specific use cases.

Support scheduler plugins. The topic of scheduler plugins is a contentious issue and has provoked intense discussions in the Linux community in the past. While preferred by most scheduler developers (perhaps unsurprisingly), the concept has been rejected by other influential members of the Linux community. One frequently-voiced criticism of scheduler plugins is that they would likely lead to a proliferation of special-purpose plugins, and that

⁴Interestingly, growing interest in tools such as the *Clang Static Analyzer* [32], the *Coverity Integrity Checker* [20], and *Sparse* [42] seems to indicate that there is significant demand for static program analysis techniques.

Linux as a whole would be better served if all interested parties came together to build a “universal scheduler” that meets all requirements [35].

Unfortunately, such a scheduler is unlikely to exist. Evidence for this viewpoint can be found in recent implementation studies [17, 19] involving several widely-studied multiprocessor real-time scheduling algorithms. In these studies, none of the tested algorithms proved to be the “best;” rather, each algorithm performed well for some of the considered workloads, but not so well for others.

As an analogy, consider filesystems: there exist filesystems specialized for high-performance computing (HPC) and filesystems for embedded flash storage. It is widely acknowledged that there exist good reasons for the development of specialized filesystem implementations, and it is considered acceptable both to divide developer effort among multiple filesystems and for administrators to choose an appropriate filesystem for their workload. In light of this, why should one expect the same scheduler to be appropriate for both HPC and embedded real-time systems?

Hence, we argue strongly in favor of introducing a scheduler plugin interface in Linux. Not only would this greatly simplify the prototyping and experimental evaluation of new scheduling algorithms, it would also allow embedded Linux variants to be tailored to their workloads.

Note that, ideally, the default CFS/static-priority scheduler should also be a plugin (so that it can be replaced), and that appropriate run queues should be provided and *encapsulated* by each plugin (as opposed to passing pointers to a statically-allocated run queue to plugin-provided callbacks). The rationale for encapsulating the implementation of run queues is that there are many different priority-queue implementations that are worth exploring [15], and the current run-queue implementation is inherently limiting, as is discussed next.

Enable global scheduling. Research has shown that there are fundamental differences between partitioned (per-processor run queues) and global (shared run queue) scheduling. In particular, *global earliest-deadline first* (G-EDF) has been shown to have properties that are desirable for certain classes of soft real-time systems [22]. Unfortunately, implementing global schedulers in Linux, while feasible (see for example UNC’s LITMUS^{RT} [45]), is unnecessarily complicated because Linux’s run queues are inherently partitioned. Since the run-queue locks also serve to serialize updates to process state, the standard

run queues cannot simply be circumvented. As mentioned above, the run-queue implementation should be considered part of a scheduling plugin’s internals and not statically mandated for all schedulers.

Regarding the frequently voiced objections to G-EDF’s viability in a “real” system, it should be noted that *xnu*, the kernel underlying Apple’s multimedia-friendly OS X, has been relying on G-EDF to support real-time applications on multiprocessors for several years [5].

Support dynamic-priority scheduling. In recent years, interesting variations of G-EDF that reduce average deadline *tardiness*, i.e., the amount of time by which a deadline is missed, have been proposed [7, 26]. These variations fall into the class of *dynamic-priority* algorithms because a job’s (see Sec. 3.2 below) priority may change at arbitrary times. Similarly, the only optimal⁵ multiprocessor scheduling algorithm for which (published) implementations exist [17, 19, 43] also belongs to the class of dynamic-priority algorithms (as all optimal multiprocessor scheduling algorithms must).

Yet another class of algorithms of considerable recent interest, so-called *semi-partitioned* schedulers [2], includes several dynamic-priority variants. Under semi-partitioned scheduling, each task is categorized as either fixed (executing on one processor only) or migrating (executing on several processors). Such algorithms allow for better system utilization than pure partitioning approaches, without the run-time overhead of allowing all tasks to migrate.

So that the desirable properties of these algorithms can be exploited, Linux’s future scheduler plugin interface should be sufficiently flexible to support the implementation of algorithms in which processes may change priority and migrate frequently (and at arbitrary points in time).

Throw out POSIX. The real-time POSIX standard is mostly outdated and does not apply particularly well to multicore systems. Research plugins and scheduler plugins targeting embedded systems with pre-determined workloads should not be constrained by such legacy requirements.

For example, consider one of the most-widely used soft real-time applications: media playback and interactive graphical user interfaces (such as in games). Currently, real-time priorities require superuser privileges, hence it is not possible (nor, given static-priority scheduling without budget enforcement, particularly advisable) for non-privileged users to make such applications proper real-time processes. Ideally, future desktop and windowing sys-

⁵See [22, 31] for a proper definition of “optimal.”

tems should be able to provide real-time capabilities in a controlled fashion—this requires interfaces and abstractions that were not envisioned by the POSIX standard; hence, scheduler plugins should not be artificially restricted in such ways.

3.2 Task Model and Predictability

Instead of solely relying on static priorities, real-time Linux should offer a more expressive API that explicitly allows the specification of timing constraints. While there are many ways of specifying timing constraints, we advocate that initial support should focus on the sporadic task model since it is the most widely-studied such model (in work on multiprocessor real-time scheduling).

In the sporadic task model, a task $T_i = (p_i, e_i, d_i)$ is characterized by three parameters: its *minimum inter-arrival* time p_i , also known as its *period*, its *worst-case execution cost* e_i , and its *relative deadline* d_i . The recurring activation of T_i is modeled as a sequence of job releases $T_i^1, T_i^2, \dots$, subject to the following constraints: job releases are separated by at least p_i time units, each job requires at most e_i time units to complete execution, and each job should complete within d_i time units after its release. Note that a sporadic task is just a logical entity that is not scheduled; instead, jobs are scheduled by the OS.

It is not immediately clear how to best apply the sporadic task model in the context of a UNIX-like process model.⁶ If jobs, and not sporadic tasks, are scheduled, then how should sporadic tasks be mapped onto Linux processes? Further, discovering the *actual* worst-case execution time of non-trivial code paths in Linux is effectively impossible on contemporary processors given current worst-case execution time analysis techniques. Is the sporadic task model therefore “impractical”?

The sporadic task model should be seen as reservation mechanism in the context of a UNIX-like OS such as Linux.⁷ In this interpretation, a sporadic task (associated with a process) is simply an execution-time budget with a minimum separation between replenishments and a guarantee that the budget can be consumed within a specified time window (as defined by the task’s relative deadline) after each replenishment. The real-time scheduler should hence ensure that all budgets can be consumed before their corresponding deadline, and that individual processes do not consume more time than their specified budget. The real-time guarantee provided

by such a system is that, if a real-time application stays within its allocated budget, then it will meet its application deadlines. Because budgets are enforced, this is even true if some applications misbehave (e.g., enter an infinite loop), a concept known as *temporal isolation*.

In the real-time literature, the mechanism described above is sometimes referred to as a *polling server*, and the concept of controlling the execution of one or more processes by means of a budget accounting policy is generally called a *server abstraction* (this corresponds to Linux’s “cgroup” abstraction).

Server-based approaches hold the promise of controlling interference caused by (threaded) interrupt handlers [8, 27]. Similarly, several server schemes have been proposed in the literature to reserve bandwidth for non-real-time processes (e.g., see [1, 9, 12, 37]). Since scheduling analysis exists for most server schemes, server-based interference control offers the ability to compute budgets such that interrupts or background workloads are processed quickly, but without unduly delaying real-time applications.

To summarize, server schemes offer analytically sound ways of compartmentalizing a system at run-time; we advocate that Linux’s cgroup support should be extended to enable the implementation of various server schemes in scheduler plugins.

When realizing the sporadic task model as outlined above (and some of the server schemes), it must be ensured that the implemented budget accounting meets the assumptions of published scheduling analysis. This is especially true when dealing with suspensions, e.g., when processes block on I/O operations or due to semaphore contention. Accounting for suspension times accurately is a notoriously hard and largely unsolved problem (see [40] for details). Hence, most analysis currently requires job suspension times to be included in the worst-case execution time (static-priority scheduling is a notable exception to this). Consequently, real-time processes should consume budget even while they are suspended, otherwise suspensions can cause scheduling anomalies.

A more expressive real-time API should also allow policies to be specified for handling budget overruns and underruns. For example, LinuxRK [37] supports a task model in which real-time processes that overrun their budget can compete for execution at non-real-time priorities; the real-time budget is interpreted as a minimum supply in this case. Similarly, the BACKSLASH algorithm [29] allows tasks

⁶To avoid confusion, in the subsequent paragraphs, we use “task” to denote a sporadic task, and “process” to denote Linux’s notion of an executing program instance.

⁷Resource reservations have been studied extensively in the context of LinuxRK [37].

to “borrow” against future allocations. Due to the pessimism typically involved in deriving budget requirements for real-time tasks, most real-time applications, most of the time, will only partially use their allocated budgets. Such budget underruns create *dynamic slack*, which itself can be scheduled and used to reduce the impact of overruns in other applications or to improve the response time of non-real-time workloads (for examples, see M-CASH [36] and Sec. 4 below).

3.3 A Prototype

To facilitate applied real-time scheduling research and to evaluate the aforementioned concepts (and others) on real hardware platforms, our group has developed LITMUS^{RT}, the **L**inux **T**estbed for **M**ultiprocessor **S**cheduling in **R**eal-**T**ime **S**ystems [16, 17, 19, 45]. In this section, we briefly summarize some of the lessons learned while building and maintaining LITMUS^{RT}.

LITMUS^{RT} predates CFS’s scheduler-class hierarchy and thus was originally designed to hook manually into the Linux scheduler. We were delighted when the CFS patches introduced the concept of proper scheduling classes, as this promised to provide a much cleaner method for LITMUS^{RT} to interface with the core scheduler. However, the scheduler-class interface turned out to be too restrictive to serve as the sole LITMUS^{RT} interface (in no small part due to Linux’s reliance on partitioned run queues). Hence, LITMUS^{RT} still hooks into several scheduler routines (but to a lesser extent than before).

Migrations. To avoid races with code paths trying to resume processes from a suspension (`try_to_wake_up()`), every runnable process in Linux needs to be on one of the per-processor run queues at all times. Since one of the primary features of LITMUS^{RT} is the support of global scheduling algorithms, the scheduling path is complicated by the need to migrate processes in `schedule()`. To ensure consistency, process migrations require a processor to acquire both the local and the remote run-queue locks. Because `schedule()` already holds the local run-queue lock, and since run-queue locks must be acquired in order of increasing lock address (to avoid deadlock), such migrations may require first dropping and then re-acquiring the local run-queue lock. Since process state may change in the brief window in which the local run queue is unlocked, care must be taken to check for conflicting updates after the migration completes. Migrations could be greatly simplified if separate locks were used to protect pro-

cess state and run-queue integrity.

Task rejection. LITMUS^{RT} allows scheduler plugins to reject processes from becoming real-time tasks (e.g., if the resulting task system would be infeasible), which requires hooks in `sched_setscheduler()`. Once Linux has a proper scheduler plugin API, changing scheduling classes/plugins should also be modularized, and the acceptance logic should be delegated to scheduler plugins.

Polling servers. Budget enforcement requires the scheduler to split the run queue in two. The *ready queue* contains processes that are runnable and eligible to execute (i.e., the corresponding job is released and has a non-zero budget), and the *release queue* contains processes that are runnable but are awaiting the replenishment of their budgets. In LITMUS^{RT}, there are two implementations of this. In event-driven schedulers (such as G-EDF), job releases are triggered with high-resolution timers, hence the ready queue only conceptually contains tasks. In quantum-driven schedulers (such as PD² [43]), the release queue is realized as a circular array of binomial heaps, so that released processes can be efficiently merged into the ready queue at each quantum boundary.

Global scheduler state. LITMUS^{RT}’s G-EDF implementation is based on the concept of *task linking*,⁸ under which the assignment of *tasks* that should be scheduled (the “links”) is maintained separately from the set of actually-scheduled, backing *processes*. Note that the set of scheduled processes can trail the links.

This has several advantages. First, the implementation of the scheduling policy becomes independent from hardware constraints: while a processor can only physically reschedule its local process (i.e., perform the actual context switch), any processor can re-link any task to any processor. In our experience, this avoids several potential races in the implementation of global policies such as G-EDF. Second, the actual preemption/context-switching logic simply becomes a matter of tracking what process *should* be executing—this code can be policy-independent and should be re-used between plugins. Third, by investigating the links of other processors, it is relatively easy to avoid superfluous migrations and context switches. Fourth, link-based scheduling lends itself to real-time analysis—the system can be understood as consisting of a scheduler that can reschedule any processor at any time, and a scheduling latency that accounts for any delay in actually enacting task-

⁸First described in the context of synchronization, see [10].

link updates [10]. This nicely abstracts interprocessor interrupt latency, non-preemptive sections, and all other OS delays into a single latency value, which can be accounted for by inflating execution costs (as discussed in Sec. 2).

Synchronization. Based on task-linking, LITMUS^{RT} includes support for userspace non-preemptable sections, in which interrupts are enabled but task-link changes are not enacted until the currently-scheduled process signals that it has left its critical section. This mechanism can be protected against abuse with a simple timeout mechanism. LITMUS^{RT} also supports FMLP-based real-time synchronization [10, 13, 14, 18].⁹

4 Case Study: EDF-HSB

In this section, we summarize EDF-HSB [12] as a case study to highlight what we believe to be desirable properties for an integrated scheduling approach for mainline Linux, and how those properties can be achieved.

Design goals. EDF-HSB was designed to support a wide range of workloads. Real systems are likely to consist of mixes of applications with differing timing constraints, e.g., a few applications that are highly sensitive to jitter, several applications with moderate real-time requirements, and some background processes (such as logging, system maintenance, etc.).

Hence, EDF-HSB explicitly distinguishes between three task categories: *hard real-time* (HRT), *soft real-time* (SRT),¹⁰ and *best-effort* (BE) tasks. Note that this task type information is used at runtime, and that one or more of these categories may be absent in any given system.

The design of EDF-HSB hinges on the assumption that the HRT component requires only a small fraction of the total system capacity, an assumption that we believe to be true for the vast majority of real systems (and especially those that are being deployed using Linux).¹¹

Conversely, the SRT component is presumed to (potentially) require a large fraction of the system’s capacity. Hence, EDF-HSB is carefully designed to avoid both unnecessarily wasting utilization to accommodate HRT tasks and placing a limitation on the maximum total utilization that can be reserved for SRT tasks (within hardware limits). Note that this cannot be achieved in general with any pure par-

tioning approach (such as Linux’s current scheduler).

Finally, since the HRT and SRT reservations are likely pessimistic due to “engineering margins” and variations in execution time requirements, EDF-HSB has been designed to reclaim unused capacity and redistribute it to “needy” tasks at runtime, i.e., EDF-HSB incorporates dynamic slack scheduling.

Slack scheduling is best explained with an example. Suppose there are two SRT tasks, $T_1^S = (30, 7, 30)$, $T_2^S = (50, 25, 50)$, and one BE task, T_3^B , in the ready queue. By EDF, T_1^S is scheduled first. Now suppose that T_1^S completes its execution after only three time units, i.e., it leaves four time units of its per-job reserve unused. Because real-time analysis of the system showed that all timing constraints are met if T_1^S uses its full allocation, the system can schedule *any* lower-priority work using the unused part of T_1^S ’s allocation prior to T_1^S ’s deadline *without the risk of violating temporal correctness*. In this case, if T_3^B requires no more than four time units to complete, then its response time has been lowered by 25 time units, as is shown in Fig. 1.

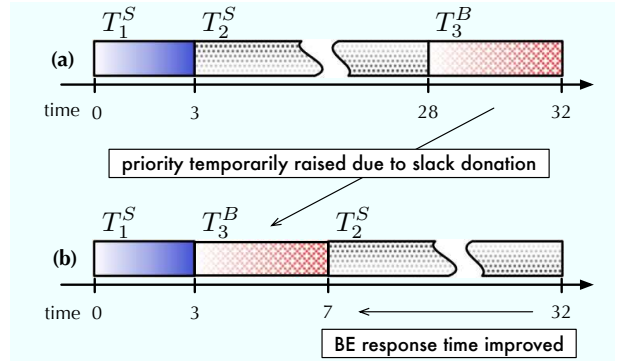


FIGURE 1: Benefits of slack scheduling. (a) Without slack scheduling, T_3^B is delayed until T_2^S completes. (b) With slack scheduling, T_3^B can execute for up to four time units at T_1^S ’s priority and hence complete before T_2^S is scheduled. This reduces T_3^B ’s response time by 25 time units.

In general, slack scheduling is a powerful technique that can significantly reduce the impact of budget overruns and drastically lower response times of background processes.

Algorithm structure. The purpose of this description is to serve as an overview; please see [12] for a full definition of the algorithm. The following

⁹A description of the synchronization support is beyond the scope of this paper—please refer to [14] for a discussion of the implementation.

¹⁰Using the bounded-tardiness definition of “soft real-time,” e.g., see [22].

¹¹Our assumption has been confirmed repeatedly in discussions with industry colleagues.

paragraphs correspond to Fig. 2, which illustrates EDF-HSB's structure.

Under EDF-HSB, similar to Linux's hierarchy of scheduling classes, the components are statically prioritized: eligible HRT tasks have higher priority than SRT tasks, and SRT tasks have higher priority than BE tasks. However, unlike Linux, special HRT servers are employed to rate-limit the HRT component, polling servers are used to isolate both HRT and SRT tasks, and BE servers are used to reserve bandwidth for BE work at SRT priority.

The HRT component is partitioned across processors to enable the re-use of uniprocessor HRT schedulability analysis. The SRT component is scheduled globally to ensure that the system can be fully utilized. As presented in [12], EDF-HSB also globally schedules the BE component; however, this could be easily changed if so desired.

EDF-HSB uses partitioned EDF together with server-based rate-limiting rules (see [12] for details) to schedule the HRT component. The SRT component is scheduled using G-EDF, and the BE component is scheduled using some non-real-time scheduler (global FIFO in [12], but integration with Linux's CFS is also possible).

Slack scheduling is carried out by collecting records of unused time in a global capacity queue, which is sorted by the donating jobs' deadlines. Capacities are scheduled with SRT priority. When a capacity is selected to "run," it is consumed by a SRT or BE job, which is selected by a heuristic and then scheduled. Such a job continues to consume the capacity until either the capacity is exhausted or higher-priority jobs arrive. Note that unused capacities must be properly "expired," as explained in [12].

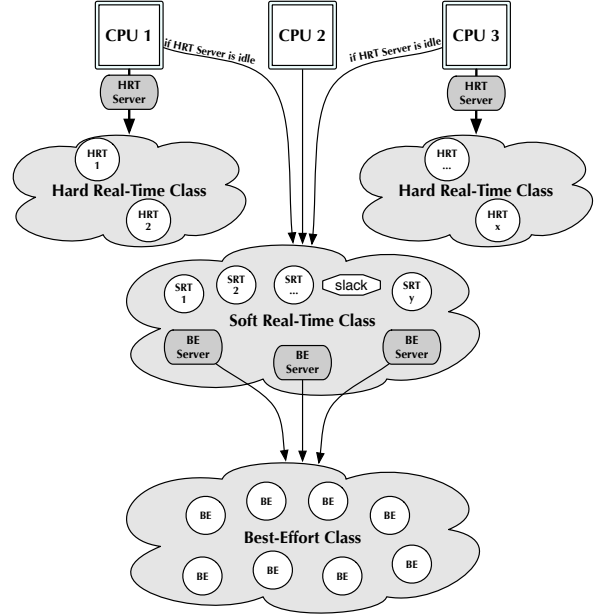


FIGURE 2: Illustration of EDF-HSB.

Properties. EDF-HSB has a number of desirable properties. The frequency of HRT interference can be freely adjusted (the period of the rate-limiting HRT server can be chosen independently of HRT deadlines; this allows "performance tuning"), and no over-provisioning of the HRT component is required (e.g., if the HRT tasks on one processor cumulatively require 20% of the processor's capacity, then the HRT server has a reserved utilization of exactly 20%, and no more).

EDF-HSB also ensures that BE processes are not starved because the SRT-level capacity reserves for BE workloads (if non-zero) ensure progress even if all SRT tasks are backlogged.

In [12], experiments are presented that demonstrate the effectiveness of slack scheduling in improving system responsiveness. In these experiments, slack scheduling resulted in a reduction of measured worst-case response times of BE tasks from over 300ms under G-EDF to less than 75ms under EDF-HSB [12]. To put these numbers into context, typing shell commands in a G-EDF-scheduled system (under a test load) felt notably sluggish, whereas (under the same load) the EDF-HSB-scheduled system exhibited average response times that, for humans, are indistinguishable from an idle system [12]. This can make a major difference in real-world scenarios. Consider, for example, an administrator who wants to re-configure a host that is suffering from overload of its real-time web services: with slack scheduling, the system remains responsive to background com-

mands as long as at least one HRT or SRT task generates slack.

The design of EDF-HSB makes it a viable choice for mainline Linux even if scheduler plugins remain excluded because it is inherently able to support a wide range of real-world workloads.

Open questions. Some extensions of EDF-HSB are still the subject of ongoing work. For one, lock-based synchronization across component boundaries remains a largely unsolved problem. Further, the dynamic nature of Linux workloads require server parameters to change at runtime, a process known as *re-weighting*. A relatively simple solution is to postpone parameter changes (and, consequently, enacting task arrivals and departures) until the beginning of the next period, but better results are attainable [11]. In the context of EDF-HSB, re-weighting schemes suitable for its hierarchical, server-based structure have yet to be developed. Note that task suspensions do not necessarily trigger re-weighting; re-weighting is only necessary on HRT/SRT task creation/destruction.

5 Conclusion

In this paper, we presented arguments in favor of greater cooperation between the real-time Linux community and academia, supported by concrete suggestions for improved real-time support in Linux. Further, we pointed out some common misconceptions that have lead to misunderstandings in the past.

Improved cooperation hinges on the willingness of both communities to approach each other. It is surely unreasonable for the Linux community to expect large, production-quality code contributions from academics. Likewise, it is unreasonable for academics to expect the Linux community to discover all relevant concepts and papers by itself. Instead, as an achievable short-term goal, participants should aim for more discussions and a lively flow of problem descriptions and possible solutions back and forth.

To that end, we suggest that (more) academics should consider attending next year's Real-Time Linux Workshop (RTLWS) [38], and (more) Linux developers should consider attending next year's International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) (to be held July 2010 in Brussels, in conjunction with the EuroMicro Conference on Real-Time Systems [23]).

References

- [1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 3–13. IEEE, December 1998.
- [2] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208. IEEE, July 2005.
- [3] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. IEEE, December 1996.
- [4] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37. IEEE, December 1995.
- [5] Apple Inc. xnu source code. <http://opensource.apple.com/source/xnu/>.
- [6] T. Baker. Re: RFC for a new scheduling policy/class in the Linux kernel. *Linux Kernel Mailing List*, July 2009.
- [7] T. Baker, M. Cirine, and M. Bertogna. EDZL scheduling analysis. *Real-Time Systems*, 40(3):264–289, 2008.
- [8] T. Baker, A. Wang, and M. Stanovich. Fitting Linux device drivers into an analyzable scheduling framework. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 1–9. NICTA, July 2007.
- [9] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, April 2004.
- [10] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80. IEEE, August 2007.
- [11] Aaron D. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina, 2008.

- [12] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70. IEEE, July 2007.
- [13] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 105–124. IEEE, December 2008.
- [14] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, M-PCP, D-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194. IEEE, August 2008.
- [15] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. Manuscript. Available at <http://www.cs.unc.edu/~anderson/papers.html>, May 2009.
- [16] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS^{RT}: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123. Real-Time Linux Foundation, November 2007.
- [17] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169. IEEE, December 2008.
- [18] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE, April 2008.
- [19] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123. IEEE, December 2006.
- [20] Coverity Inc. Coverity Integrity Checker. <http://www.coverity.com/>.
- [21] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a Linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238–245. IEEE, May 2004.
- [22] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.
- [23] J. Goossens, J. Anderson, and G. Fohler. Call for papers for the 22nd Euromicro Conference on Real-Time Systems. <http://ecrts10.ecrts.org/>, July 2010.
- [24] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*. Springer-Verlag, September 1998.
- [25] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master’s thesis, TU Dresden, 1996.
- [26] S. Kato and N. Yamasaki. Global EDF-based scheduling with efficient priority promotion. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 197–206. IEEE, August 2008.
- [27] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 57–68. IEEE, April 2007.
- [28] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM, 1995.
- [29] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421. IEEE, 2005.
- [30] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.
- [31] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.
- [32] LLVM Project. Clang Static Analyzer. <http://clang-analyzer.llvm.org/>.

- [33] P. Mantegazza, E. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, April 2000.
- [34] P. McKenney. Real time vs. real fast: How to choose? In *Proceedings of the 10th Linux Symposium*, pages 57–66, July 2008.
- [35] I. Molnar. Re: [PATCH][pluginsched 0/28] Plugable cpu scheduler framework. *Linux Kernel Mailing List*, November 2004.
- [36] R. Pellizzoni and M. Caccamo. M-CASH: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40(1):117–147, 2008.
- [37] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 476–490. ACM, January 2001.
- [38] Real-Time Linux Foundation.
<http://www.realtimelinuxfoundation.org/>.
- [39] Real-Time Linux Wiki.
<http://rt.wiki.kernel.org/>.
- [40] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 47–56. IEEE, December 2004.
- [41] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [42] Sparse Project. A semantic parser for C.
<http://www.kernel.org/pub/software/devel/sparse/>.
- [43] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198. ACM, May 2002.
- [44] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pages 112–119. IEEE, June 1998.
- [45] UNC Real-Time Group. LITMUS^{RT} project.
<http://www.cs.unc.edu/~anderson/litmus-rt/>.
- [46] Y.-C. Wang and K.-J. Lin. Enhancing the real-time capability of the linux kernel. In *Proceedings of the 5th International IEEE Conference on Real-Time Computing Systems and Applications*, pages 11–20. IEEE, October 1998.
- [47] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 1–12. USENIX Association, June 1993.
- [48] V. Yodaiken and M. Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. USENIX Association, January 1997.

Problems regarding teaching Linux programming for Industrial realtime programmers

Fodrek Peter

Department of Information and Communication systems,
Institute of Automatic Control and Industrial Informatics,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology

Ilkovicova 3, 812 19 Bratislava, Slovak republic
peter.fodrek@stuba.sk

Abstract

Paper describes problems regarding teaching of Linux programming of our students. First and only course of this based on parallel system programming at level of utilities. This way was chosen because anti-viruses are utilities and there were all of technological TOP3 antivirus software houses in 2007 based in Czech republic and Slovak republic, which were established by divorcing Czechoslovakia (Czech and Slovak federal republic) in 1993. First of them is based in Brno that is 130 km from Bratislava and second of them is based directly in Bratislava. Level of teaching and other discussed factors caused that Linux programming does not take sufficient results and does not support Linux knowledge of students as well.

1 Introduction

We and our PhD candidates use RTAI kernel modules to control Plasma based Cutting machine for metallic materials. JAVA/JNI based Eclipse application is used for an GUI. Therefore we planned to educate bachelor grade students for such type of programming. Before our course students have studied only Microsoft Visual Studio based C programming with basics of Algorithm design with basics of computer architecture. First of all we teach them JAVA with command line interface at the beginning and then GUI.

The first part is as bad for students as they are not able to repeat compile command or edit source code without JBuilder IDE. The second part is much more popular. Because of need for Web API of controlled processes in the industry as last part they rewrite JAVA application to JAVA applet. Last part is not popular. Second course is based on Linux programming. First thing that was eliminated was extremely unpopular need to program only in text ter-

minals/console without X server. It is same problems as in JAVA. students are not able to use command line interface. Why are students so afraid of command line(shell) and Linux as well?

2 Pre-university Education

Many students that studies and universities of Technology are not best student but worst ones. It is because on lower grade education are information technologies presented as main knowledge of anyone. They never heard about computer programming. This is because they never seen command line interface of computers. Although There was study developed between 2004 and 2005 which listed extreme savings per seat when open source e.g. Linux solutions will be used as shown by figure 1. Because Slovakia is member Eurozone(officially the euro area) from January the 1st, 2009, we can use conversion exchange rate 30,1260 SKK per 1 Euro. Then saving per 2500 seats is 5,127,630.62 EUR per inno-

vation cycle of 6-8 years. This amount of money equals to 0,9532 EUR per inhabitant per innovation cycle. There are 225,766 university students in Slovakia what is 90.3064 times more than number in study. Then there will be 86,08 savings per inhabitant per 8 years for only University students using open source solution.

Listed study shows that when there will be 50% migration to Open Source in non-priority areas like commercial sector and 100% migration at primary and secondary schools there will be savings at that time prices at least 18 billions SKK (597,490,540EUR) during 6-8 years. Educational sector can save 5.01 billion SKK (166,301,534 EUR) including saving in primary and secondary Education 4.1 billion SKK (136,095,067 EUR) and universities to save 0.75 billion SKK (24,895,439.20 EUR). This equals 12,150 computer seats when compared to study.

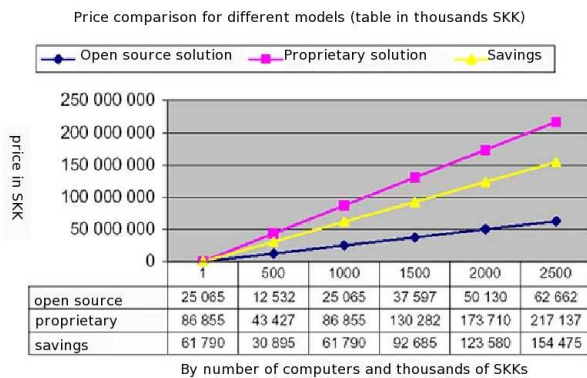


FIGURE 1: *Savings with use of open source solution[1]*

Therefor State secretary for Education join program to educate Primary and secondary schools teachers how to teach basic operation in open source software as shown of Figure 2. There were much anticipation of such project for us to prepare students to code for Linux, because it may stop FUD from Microsoft to act as main barrier from usage of Linux. This may cause not to aware use command line because student will know that there is also GUI for Linux. This act says that schools from Germany, Poland, Portugal, Spain and Slovakia will use open source software especially Openoffice.org and Mathexer with support of European Union's Comenius program But..



FIGURE 2: *Linux tag announcement[2]*

But This project in Slovakia failed. One and half of the month after this project started there were elections at Slovakia. And it completely changed situation. Project was stopped.

3 Political influence

Political representation as been changed and State secretary form 9,5% Hungarian minority was changed by Slovak nationalist. Therefor everything have to be reversed. And there was not exception in pro-OpenSource orientation. official reason to cancel tender for this project realization was "incorrect set of tender criterions to chose best offer... which too much preferred teacher's education instead of price"[3]. Education of teachers is most important not to stay at Microsoft's dominance. Our generation of 70's was leaded at computer courses by programmers. Current students are supervised by pedagogues which do not program computer codes. Pedagogues attend 3 week course of computing sponsored by Microsoft are know only its Windows and Office form faculty and lower education grades. The worse thing is that in december 2006 was tender repeated not publicly but with 5 chosen companies[4]. And the worst thing is that winner[5] is company that is present in current (august 2009) clientelistic

cause of Education state secretary[6]. It is little bit funny that author of Microsoft Office is Charles Simonyi(Simonyi Karolyi) which is hungarian.

4 Principle of university education

There are also problems in principle of university education. Especially in capability of university lectures to check-up students during study. There is and tradition for finished students to write their nicknames on the street near campus with same diagram or message. This year there was written sum block with several inputs. one of the inputs is described as "horse" meaning cheat sheet. In the programming there is not possible to use cheat sheet and therefor 40% of the students recognized Unix programing as a worst course of the semester.

Question is why student are able to use "horse" on each course except programming?

1st grade of primary school	53,000
9th (final) grade of primary school	65,000
1st grade of secondary school(offered seats)	93,000
4th (final)grade of secondary school	64,000
1st grade of Bachelor level (offered seats)	69,000

TABLE 1: *Count of students and offered student seats for next academic year in grades as of 2006[7][8][9]*

Table 1. answers this question for slovakia especially when we add information that university and secondary school endowments depends only on count of its students since 2002. In addition Technology is not preferred subject of study for vast majority of students. They hate mathematics and physics at most. Therefor study of technology is near exclusively for students thats application for study of humanities was rejected by university capacity. Only for about 1/12 of students of technology was technology their first choice of university study. And about half of them is unable to learn computer programming as found by Dehnadi[10]. As a result only 1/24 of students is capable to learn to produce computer programs. When we have 72-75 student seats in same grade(we are capable to teach 150) there are 3-5 students capable to learn to code in one grade. As Intel found there is only 1/100 of programmers who are capable to do parallel programming[11]. Therefor only one student in 20-30 years is allowed to be successful in course I deal with as only Linux programming course.

4.1 Unix/Linux

As additional problem there is fact that in 1995 then was only Internet access for students at our university via Digital Ultrix machine with vdt52s terminals or emulated vt100. We were needed to teach Unix shell commands to access internet before studying Unix programming. After years there was made a change: students can access Internet by their own laptops or Windows machines. We also have DOS and 8-bit personal computer usage experience and therefor we had no awareness of command line. Our course strategy was made long before we had been studied it. Generation before us was another then current generation. Technology was only studied by top students with interest in computer programming. Course strategy is based on assumption that nothing changed from that time. As described before assumption is false.

4.2 Time to study

Another reason of problems is miss-prediction of time to study. In 1995 then there were two courses with 6 credits. Number of credits per course is at our university counted as sum of 50 minutes lectures , 50 minutes applied practical laboratory lessons and 60 minutes preparation at home per week. Then there were 2 hours of lectures, 3 hours of lessons and predicted 1 one hour at home. when semester has 12 teaching weeks students have been studied two time 72 hours which equals 144 hours of Unix study. Now students have only one course with 5 credits (2 hours of lectures, 2 hours of lessons and 1 hour at home). Then they have 60 hours of Unix study at most. But as seen in introduction industry needs equal range of knowledge of graduates.

4.3 Old school methods

In addition lecturers are from different ages (over 55 or under 35) and there is hard to find and consensus in education methods or policy (friendship or authoritative) of access to students. In the methods we can agree across ages, but not in policy. Older generation prefer authoritative policy and younger prefer to used mixed authoritative and friendly policy.

4.4 Technology

As university have endowment only 1,300 EUR/year/student and students have 11 courses per year or 60 credits/year therefor we can use only

108,4 EUR/student/year which is extremely low to get Massive parallel processing based supercomputer (for at least about 1 million USD) or specialized parallel computing lab (for about 10 millions UDR to build a 2 millions USD per year of usage) needed to effective teaching of parallel programming.

5 Conclusions

We described problems in teaching Linux programming at Slovakia. As result we would only resign to such type of education which is unacceptable for slovak software industry or make course based on new assumptions which are true. Teaching kernel mode programming must stay to be part of thesis consultations as it is done just now because there is no time to teach it at normal courses and thesis about it is chosen normally only by students who are interested in the problem of hard realtime programming at work. As shown some problems are solvable some are not especially financial problem and popularity of study of technology. In area that we can solve we must do it because of industry but it will be helpful for us to get financial support from listed Bratislava's antivirus labs ESET(with NOD-32 product) a Brno's AVG technologies (Grisoft in the past, Brno is 130 km from Bratislava) and Prague's Allwill Software(with AVAST product, Prague is 331 km from Bratislava) [12] to solve another problems. But it will be harder because on June, the 11st 2009 signed state secretary new three year agreement with Microsoft for 13,127 Windows and Office university Campus licenses for year cost of 1,001,983.91 EUR. old agreement finished on June the 30th 2009.

References

- [1] http://www.infoware.sk/buxus_dev/generate_page.php?page_id=2616
- [2] <http://www.heise.de/newsticker/LinuxTag-Besuch-aus-dem-Justizministerium-/meldung/72720>
- [3] <http://www.sme.sk/c/2840662/mikolaj-zrusil-verejne-obstaravanie-vyhlasene-byvalym-vednim-ms.html>
- [4] <http://www.zive.sk/Bleskovky/Skoly-dostanu-400-multimedialnych-suprav-za-80-milionov/sc-4-a-270588/default.aspx>
- [5] <http://www.sme.sk/c/3391336/mikolaj-obdaruje-vybrane-skoly-multimediami.html>
- [6] http://spravy.pravda.sk/pocitace-doskol-doda-mikolajov-vyskumnik-fie/sk_domace.asp?c=A090730_113613_sk_domace_p12
- [7] <http://aktualne.centrum.sk/domov/zdravie-skolstvo-spolocnost/clanek.phtml?id=1173031>
- [8] <http://www.modernaskola.sk/un,c474/>
- [9] http://hnonline.sk/1-10042830-18692950-k00000_detail-63
- [10] <http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf>
- [11] <http://www.theinquirer.net/inquirer/news/1026585/parallel-programmers-prepared-glorious-revolution>
- [12] <http://news.softpedia.com/newsImage/NOD32-Among-The-Most-Powerful-Antiviruses-2.jpg/>

This paper is supported by grant from KEGA agency

Analysis of inherent randomness of the Linux kernel

Nicholas Mc Guire, Peter Okech, Georg Schiesser

Distributed & Embedded System Lab, SISE, Lanzhou University, China

Tianshui South Road 222, Lanzhou, P.R. China

mcguire@lzu.edu.cn, POkech@strathmore.edu, georg@opentech.at

Abstract

While analyzing latency data from real-time Linux variants we found that there are distinct parts to the system jitter - those that can be attributed to software constructs and those that are inherent in complex software systems running on non-deterministic hardware. Initial investigations focussed on explaining these results by considering various caches (L1, L2, BTB, TLB, etc) [12]. While this seemed to allow some level of explanation it did not satisfactorily explain the distributions found during analysis.

Essentially hunting for the maxima of latency, which was the common initial approach, only can detect grave latency issues, like excessively long holding of locks - but it is not able to detect low-level latency causes like miss-alignments, or short term locks that are in a hot path and thus contribute significantly to the systems overall latency and jitter. Further the maxima - if the assumption of inherent randomness hold - are not associated with a specific code-path but rather with the code-path being executed in a specific, system level, context - thus we believe that a statistic approach to tracing latency is needed.

Basic analysis of real time behavior in complex software systems can be split roughly into the following parts:

- Timestamp precision - how precisely can an event be associated with a timestamp from a specific clock-source
- inherent randomness - how non-predictable is the execution of functionally deterministic code.

The first can be quite nicely measured (or rather estimated based on measurements) - the second is a bit more complicated as there currently is not even a well accepted definition, nor a practically meaningful metric.

These two factors, we believe, can form a useful constraint for the lower-bounds of timing behavior that can be achieved in complex software systems - be that scheduling jitter, interrupt latency, or bandwidth variance.

In this paper we will present the current state of our assessment along with an argument why we believe that inherent randomness is present and of what quality this randomness actually is based on preliminary evaluation of a random number generated (RNG) derived from our timestamp measurement code.

KeyWords: inherent randomness, non-determinism, randomness, complex systems

1 Introduction

Real Time Operating Systems (RTOS) are typically identified with determinism and predictability. The prime way to tackle the problem of RT seems to be to postulate a deterministic hardware system and put on top of it a deterministic software blob - albeit neglecting the limitations of the abstractions behind both the hardware and software. Based on this one

then starts hunting for the cause of excessive latency or jitter in a RTOS and tries to identify it with particular code-paths or event sequences. While this is to a certain extent successful - notably as long as there are specific code-paths that need improvement, there is a certain point at which this "hunting-for-maxima" seems to yield no significant improvement. In this paper we will outline our findings with respect to inherent randomness of complex hardware/soft-

ware systems and argue that there is a certain level of randomness that is associated with the complexity rather than the specific code path and for complex hardware/software systems - GNU/Linux on mainstream super-scalar COTS CPUs - we need to develop metrics and methods to live with this inherent randomness rather than try to fight it.

1.1 Related Work

J. Viega in [3] focussed on security related random number needs, and outlines what methods seem suitable to provide reliable random numbers, though these are based on a seed which is from a true random source (i.e. OS entropy pool fed by interrupt related entropy [4]). The prime conclusion from this paper, from the perspective of our work, is that there are sound algorithms available, notably universal hash functions, that would allow to attain good randomness in a security sense in general purpose OS. A further note in this paper that our contribution also suffers from is that there "...there is not a consistent set of requirements or terminology between different solutions".

Sameer Nipadkar and Matt Davis in [2] describe their approach of using concurrent threads and the indeterminism in the scheduling of a complex OS to produce random numbers through thread synchronization timing variance. They describe this approach being neither a TRNG nor a PRNG - "Threads on the contrary lie somewhere in the middle between the true and pseudo random number generators." In their publication [1] they brand this approach as a "Pseudo Random Number Generator" while we believe there is sound evidence for such an approach being a True Random Number Generator albeit with some deficits that may need fixing in the specific implementation.

With respect to metrics, the most relevant work we have been building on is the work from Pierre L'Ecuyer [6] which provides a extensive set of statistical test utilities to evaluate the output of random number generators. We believe that a subset of these tests along with a well specified procedure could constitute a good basis for a generally acceptable metric for system level inherent randomness.

2 Sources of non-determinism

Any system has global variables in the one or other form, in complex systems there are many such global variables (think of free shadow registers, TLBs, available cache lines, memory ... timeouts in communication, etc.) many of these global variables are locally references directly or indirectly.

- direct reference - a send operation on a queue that can block
- indirect reference - a system call that internally allocates memory dynamically and thus can take largely diverging amounts of time depending on the systems state
- an instruction that needs to evict a L1 I line before being performed
- a my-op that gets stalled due to a (functionally) unrelated pipeline condition
- a branch in the my-ops that is once taken and once not due to the BTB being exhausted by functionally and temporally unrelated code execution paths.

Note that we are not referring to any external or asynchronous events yet in this list - so we refer to this as internal sources of non-determinism.

This for it self still is not non-deterministic from the perspective of the individual application. At the moment where we have a concurrent preemptible system the picture dramatically changes. The local state (application) actually can be revisited, but its dependency on global state makes the outcome indeterminable. For the temporal domain this is a well accepted fact - after all race conditions in software are a much feared fault for decades now and anybody hunting down such a bug knows how undetermined the temporal precision of full featured preemptive operating systems are. Notably the introduction of the lock-dependency validator in mainline 2.6 had revealed a number of race conditions that had not yet been discovered at runtime by anybody, even though they had been present for a substantial time in the kernels code-base.

2.1 Intentional non-determinism

In some cases we know that data used for decisions is random - some may be:

- random numbers (i.e. from a TRNG)
- asynchronous event timestamps
- global conditions (i.e. `if(in_interrupt())` derived from asynchronous events
- error conditions (i.e. checking return values) - or is anybody expecting a deterministic failure rate of `printf` ?

These and other sources of non-determinism exist at even simply applications levels - paired with concurrency this de-facto means that individual application code - while exhibiting a well defined local

state - has no deterministic global state and we are unable to predict the actual behavior of even simple applications.

Example: timing of a 5 integer instruction (C-level)

This code run with interrupts disabled, executes 5 integer instructions in a warmup loop - this ensures they are cache hot - and then times the final execution - the plot over the warmup loops indicate that it takes quite a few warmup loops to get the system into a more or less well-defined state - never the less it never reaches a constant execution time.

```
__asm__ __volatile__("cli>:::memory");
for(j=0; j< w; j++){
    x1=1;
    x2=x1*1;
    x3=x1*1;
    x3--;
    dummy+=x3/4;
}
__asm__ __volatile__("cpuid\n\t" \
                    "rdtsc\n\t":\
                    "=A" (start));

x1=1;
x2=x1*1;
x3=x1*1;
x3--;
dummy+=x3/4;
__asm__ __volatile__("rdtsc\n\t":\
                    "=A" (stop));

__asm__ __volatile__("sti>:::memory");

timestamps[n++]=((long)(stop-start));
```

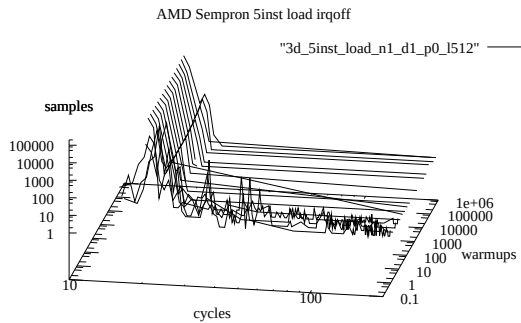


FIGURE 1: 5 instruction execution time with disabled interrupts (AMD Sempron)

In a modern operating system even a trivial hello world:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MSG "Hello World\n"

int main(int argc, char **arg){
    int ret;

    ret=printf(MSG);
    if(ret == strlen(MSG)) {
```

```
        return EXIT_SUCCESS;
    } else {
        return EXIT_FAILURE;
    }
}
```

exhibits non-deterministic properties. At the application level printf could fail, at the system level the entire process of spawning the application could fail and this program might actually never be executed at all. If one looks at the possible points of failure of this trivial application on a modern UNIX operating system there are actually a few hundred points in the code where failures are possible (coded fault handling) each of which has only a very small probability of actually ever being seen - never the less we only can guarantee this hello world to actually output the intended "Hello World" string with a certain probability and even if we were to analyze the entire OS level code (kernel, libc, shell, etc) involved we would not be able to guarantee its execution. The main sources of non-determinism of this trivial application are not in the scope of the application code - in fact one rarely sees anybody bothering to catch the return value of a printf as it simply is assumed to behave well - so most readers will feel this hello world is a bit artificial code. What this does exhibit though is that the local state transition of this trivial example (with its terminal state being success or failure) is dependant entirely on the OS level component and we are unable to actually predict there behavior other than statistically.

2.2 The issues with defining real-time

We are not going to go into the multitude of real-time definitions available, there are many and new ones being added (i.e. cooperate real-time recently hit the stage) - the only contribution to this discussion that we would like to add is that the question of determinism (hard real-time) vs probabilistic definitions (soft real-time, firm real-time) is maybe simply missing the point as they are concerned with two different system descriptions. The hard real-time definitions are focussed on functional determinism - neglecting problems of system failures (i.e. random hardware faults, or meteorites striking the test-system), while the soft real-time definitions are focusing on observations in systems where not all components are known (the question if they could be known or not is not so essential).

The claim we are making here is that at system level (a computing system in a real-world setup) there are always potential failure scenarios that invalidate any deterministic hard real-time definition - although one may be able to reach very high levels of confidence (lets say less than 10E-9 failures per

hour). So we are considering real-time a probabilistic quality of a system and are going to present recent tests to argue why this approach makes sense to us. I would like to emphasize that it is not relevant if the system might be deterministic, the question is only if we actually are able to formulate this property and the initial state of the system in a way that allows us to predict the behavior - which would amount to a deterministic hard real-time system - or if the inability to define the initial state and/or formulate the systems behavior prohibit us from determining the "next state" of the system with absolute certainty. This inability need not be based on physical constraints (like in quantum mechanics) they might be simply due to the limited resources available to us in the real-world.

3 Real Time Metrics

While there have been discussions on Real Time vs Real Fast at a qualitative level, we believe that a quantitative approach is mandatory for system level comparison - the long history of failed RT-metrics might suggest otherwise though.

We are not claiming that we solved the problem of RT-metrics, but we will give it a shot to introduce two fundamental metrics that allow system comparison.

- Inherent System Randomness
- Timestamp precision

These two metrics are the lower bound constraints for any quantitative statements of higher level metrics (i.e. interrupt response, scheduling jitter or WCET) in complex computing systems.

3.0.1 attempt at a definition

Inherent System Randomness: Non-determinism of a complex HW/SW system due to the limitations of putting any task (SW) into a well defined initial state.

The issue here simply is that, while it is not disputed that HW and SW in principle are deterministic (functionally and temporally) - the actual problem is to determine the initial state of both the hardware (i.e. CPU internal shadow registers, state of execution units, pipelines, caches, external controllers...) and software.

Missusing the concept of the pilot-wave theory [7] the claim is that the systems global state and the next state transition are well defined at all times, but not known by the observer; the initial conditions of the system (HW and SW) are not known accurately,

so that from the point of view of the observer, there are uncertainties which can be modeled as random characteristics.

While one can argue that it might be possible to actually determine the initial state of a modern CPU or even the entire system, this ability is not relevant for any practical system and our experiments at de-randomization suggest that achieving a well-defined state at application level is at least not practicable for any real-life application (see below)

3.0.2 Measuring randomness

The first problem to resolve is to demonstrate that a modern CPU actually exhibits inherent randomness - that is randomness that is not triggered by external events like interrupts.

The actual metric proposed for this is the quality values for random bitstreams produced by execution time randomness of a trivial code construct - something like:

```
__inline__ unsigned long long hwttime(int shift)
{
    unsigned long long int x,res;
    int i;
    int bit=1;
    res=0;
    bit<=<=shift;
    for(i=0;i<32;i++){
        __asm__ __volatile__("rdtsc\n\t"
                               : "=A" (x));
        res|=((x&bit)>>shift)<<i);
        usleep(delay);
    }
    return res;
}
```

If this were run as a tight-loop then patterns in sampling the TSC do emerge (though they still exhibit a high-level of entropy) - if the loop is allowed to run "unknown" code in between by the call to usleep, the randomness of the sampled TSC reaches very high levels, comparable to TRNGs based on background radiation of thermal noise - thus clearly indicating that the OS is "randomizing" access patterns. It should be noted that even when run with disabled interrupts this works just fine.

If the underlying system is deterministic this code will yield a non-random sequence - but as measurements show the output is comparable (if not better) than random generators using background radiation [9], thermal noise [8] or interrupts events [4] to "generate" entropy.

Readers will note that it is technically not feasible that the results of a software RNG - even if sampling the inherent randomness of the CPU - be of better quality than a geiger counter sampling background radiation - this surprising result can be

explained by reviewing the driver used at hotbits.org [10], which uses a serializing instruction before the call to rdtsc and further uses a slow peripheral (serial port) as input device - these two factors reduce the randomness of the sampling process and explain why such a lower randomness can be observed.

The second metric (and this still needs some work) is to look at the execution time variance of simple code sequences on the respective hardware. The interest in the behavior of simple code sequences is simply that if these exhibit a non-deterministic behavior timing wise then in a preemptive system like GNU/Linux this will be amplified at the higher levels of the system as small variances in execution times in individual code-paths induce unpredictability of instruction sequence at the CPU level. Though these variations might seem small the impact on modern CPUs is quite large due to the internal complexity of these CPUs (multiple pipelines, prefetch-units, pipeline inter-dependencies, etc.).

To measure this we again utilize a randomness metric but instead of calling on the OS to "randomize" the state of the CPU (the call to usleep(delay)) we put a well defined code sequence before the actually measured code and look at the timing variance of the fixed sequence:

```
warmup_loop{
    sequence
}
rdtsc
sequence
rdtsc
```

The actual code then looks like this:

```
for(i=0;i<32;i++){
    /* this code is a deterministic warmup */
    for(j=0; j< warmup; j++){
        x1=i;
        x2=x1*i;
        x3=x1*i;
        x3--;
        dummy+=x3*x2-(x1/4);
    }
    __asm__ __volatile__ ("rdtsc\n\t":"=A" (x))
    res |= (((x&bit)>>shift)<<i);
}
```

The code sequence only references local variables, is using a simple subset of instructions (arithmetic instruction available on all systems) and the results are stored in an array that is much smaller than the L1 D cache (we chose 1/8) to minimize caching effects. Further this is run with interrupts disabled. By doing so we can observe the randomness of the CPU itself, and by iterating over the number of warmup_loops used we can seek to "de-randomize" the CPU (which currently seems to be doable only

in very specific tuned assembler code cases). Thus what we get from this is the inherent variance of the CPUs execution time for a given simple sequence of instructions - any attempt to rely on a system providing a higher level of determinism seems unrealistic as impact of interrupts and caches will make things even worse.

As actual metrics we propose the use of the evaluation code presented at random.org [11], which tests for

- Entropy
- Arithmetic Mean
- Chi Square
- Monte Carlo Pi calculation
- Serial correlation

A typical run of this on a AMD Hammer (UP) will yield:

```
Entropy = 7.625602/byte.
compres = 4 %.
chi sqr = 71.09.
Arit mean = 119.2637
Mont Car Pi = 3.105882353, err. 1.14 %
Ser. Corel. = 0.041795
```

(Note that only the output format of **ent** was reformatted to allow better script processing - the tests themselves are unaltered).

A verification with the full test-suit from TestU01 [6] convinced us that the results are actually statistically random in nature reliably. Further these tests have been done on AMD Sempron, AMD Duron, Intel Core Duo 2, Intel Celeron

Note on other architectures: on MIPS (Loongson 2F) we were simply not able to access the respective register from user-space directly - so this simple measurement method did not work, on PowerPC (405/440) we had similar problems with direct access to the Timebase - so this is X86 only at present - for other architectures kernel level implementations are in the works.

timestamp precision:

Why timestamp precision? Any decisions made based on time mandate that an event of interest can actually be timestamped precisely - thus the timestamp precision is a lower bound for any time related decision in a system. Specifically in an RTOS no decision can be more precise than the timestamp capability of the system.

Timestamp precision - dependant on:

- time source resolution
- inherent randomness of the CPU (hardware)

- inherent randomness of the system (software)
- isolation of the time-sampling code

there is no point to claim any bounded jitter on scheduling that is below the inherent randomness of the system it self - this is a hard lower bounds for timestamp precision.

The time source resolution has been increased dramatically over the past decade allowing at least microsecond resolution on most current systems, generally allowing resolutions of one to 10 cycle at CPU frequency (though this depends on specific settings on some systems).

At the same time the complexity of CPUs has increased to a point where the inherent randomness of the CPU and its interdependency on other hardware units (i.e. FSB, north-bridge) makes it useless to increase the time source resolution any further.

At the software level the impact of functionally unrelated code has been improved (notably in RT-preempt) in GNU/Linux providing relatively good code isolation under appropriate configuration.

So timestamp precision in CPU cycles would be one of the proposed base metrics for a RTOS. The timestamp precision of system we measured can be as bad as a few hundred cycles due to the impact of serializing instructions. Claiming scheduling jitter below the timestamp precision technically makes no sense in our opinion.

3.1 Measuring timestamp precision

The actual metric proposed for the timestamp precision is quite trivially, run two calls to consecutive calls to "rdtsc", calculate the difference and search for min/max values.

```
while(n < loops){
    unsigned long long index=0;
    usleep(1);
    hwttime2 = rdtsc();
    hwttime1 = rdtsc();
    jitt=hwttime1-hwttime2;
    index=jitt/scale;
    if(index > GRAPH_SIZE){
        out_of_bounds=1;
    } else {
        graph[policy][index] += 1;
    }
    n++;
}
```

If this is now run at different priorities and scheduling policies one gets an overview of what the lower bounds of any timing decisions at code level will be for the given settings. At the same time this gives a suitable lower bounds for the OS level scheduling jitter. What ever is able to interrupt or intrude the two consecutive rdtsc calls is also able to

impact any other instruction (note there is no cpuid instruction used as we are interested in the possible impact of the pipelines) thus even if the OS does better at scheduling the application would not benefit from this simply because the scheduling tests will indicate when the application code got started, but not when the code was able to actually perform the intended operation. If we know how intrusive the OS environment can be then we get a better estimate of the actual jitter of any action code might be taking.

Note that we have not used a serializing instruction to read the TSC (rdtscp or cpuid+rdtsc) as this does not improve the timestamp precision - quite the contrary, serializing instructions have a profound negative impact on timestamp precision and are in fact one of the main limitations to timestamp precision as cpuid it self can take up to a few hundred cpu-cycles on multiprocessor systems in the worst case.

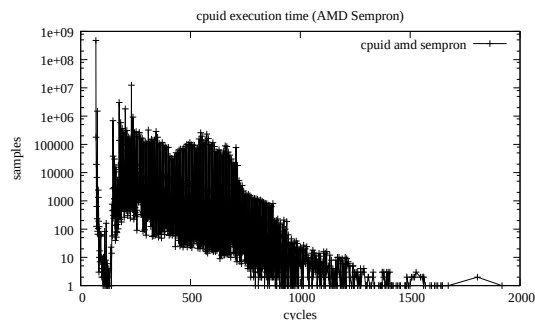


FIGURE 2: *cpuid execution time in cycles (AMD Sempron)*

4 De-randomization

To actually confirm that on of the sources of inherent randomness in a modern computing system can be attributed to the CPU, we investigated the ability to construction code that would not exhibit random execution times. The code in question is very simple code and the constraints needed to achieve this deterministic behavior preclude use of these methods for any real-life code - never the less it is interesting in-sight in the level of randomness of a modern CPU.

Measures taken to "de-randomize" code include:

- warmup loops (to ensure cache hot code/data)
- Fitted into L1 cache or even a single cache line (instruction and data cache)
- only local data involved that fit into a L1 D cache or a single cache line.
- nosmp on multicore boxes
- interrupts disabled
- dynamic cpu frequency scaling disabled

- well selected CPU frequency - not all CPU frequencies allow constant execution times to be attained
- serializing instructions (cpuid)
- manually tuned loop, both length and instruction distribution, to fit pipelines
- simple set of instructions (typically a single one-byte instruction, i.e. "inc

Even with these quite excessive de-randomization efforts which mainly focus on limiting the physical resources involved in the execution of the code to an absolute minimum, even within the CPU, it is not reliably possible to achieve absolutely constant execution times - though the variance can be reduced to roughly $10E-9/10E-10$. The CPU in this setup was not using most of its execution units (FP,SSE,complex-ALU,barel shifter) but rather limited to a subset, which tentatively explains why the randomness was reduced.

A side note on serializing instructions - cpuid - does not increase the precision of rdtsc - it only prevent out of order execution - but the penalty of cpuid it self is so high that it actually reduces the precision of timestamp association. On a 1.8GHz AMD Sempron a cpuid causes an overhead of 65 cycles, reducing the variance of rdtsc - but the offset is far larger than the variance - so the net precision is reduced. In fact one of the sources of decreased randomization in the random.org device based on background radiation monitoring might well be the cpuid instruction along with the serial line used for triggering the cpuid rdtsc - this is a bit speculative at this point though as we have not been able to verify these claims due to lack of appropriate equipment.

Further it showed that the actual parameters (i.e. inner and outer loop length CPU-frequency selected) had to be re-adjusted to fit each CPU, we from this conclude that it is not possible to write any meaningful code for a modern CPU that would actually provide constant execution times over even a closely related set of CPUs, and thus de-facto any real-life code exhibits inherent randomness.

5 Conclusion

The maybe most provocative conclusion at this point - modern CPUs are inherently random and a complex general purpose OS on top amplifies this inherent randomness substantially. More work on this is in the pipeline, and we hope to provide more evidence of this speculation in the neer future.

From the current work we have drawn three main practical conclusion are:

- Real Time metrics must be found that can reflect the inherent randomness of modern complex hardware/software systems
- A set of accepted metrics to describe the basic system parameters of inherent randomness and timestamp precision is needed to make systems comparable
- A probabilistic approach to real-time performance to us seems the only meaningful one.

As a starting point we propose the use of:

- timestamp precision in cpu cycles based on a simple test-code that iterates over priorities and considers serializing instruction impact.
- system randomness based on entropy, chi square, Monte Carlo simulation of Pi and serial correlation.

While we were able to find a, in our opinion, convincing demonstrating of the inherent randomness of modern systems by providing a software based random-number generator, we think this is currently not much more than a discussion input and should lead to a widely accepted metric for system randomness to base "high-level" real-time metrics on. The second contribution is the definition of, though trivial, timestamp precision tests. Essentially it is not relevant to which specific phenomena the variance can be attributed, essential is only that these phenomena can't be evaded with acceptable penalty as the de-randomization tests showed and thus any real-time metric will have to take this inherent uncertainty of time-stamps into account.

Practical use of these conclusions are proposed for the initialization of random number pools at boot time of GNU/Linux or for entropy generation of systems that don't have sufficient sources from asynchronous events (typically on embedded systems with only low-bandwidth peripherals). Work on this is under way.

In future works we hope to cover inherent randomness in more depth and find suitable models that allow estimations of execution times for actual real-time systems based on complex hardware/software like GNU/Linux with its real-time extensions on COTS hardware.

References

- [1] Matthew Davis & Sameer Niphadkar, *LibMT-PRNG: A Multithreaded Pseudo Random Number Generator* Dr. Dobb's Journal April 20, 2009

- [2] Sameer Niphadkar & Matt Davis, *Random Number Generation via Threading*, December 12 2009
- [3] Viega, J. *Practical Random Number Generation in Software* Proceedings of the 19th Annual Computer Security Applications Conference. December 2003.
- [4] Jake Edge, *On entropy and randomness* <http://lwn.net/Articles/261804/>, December 12, 2007
- [5] Maurice Herlihy, Nir Shavit *The art of multiprocessor programming*, Morgan Kaufman, February, 2008, ISBN 978-0-12-370591-4
- [6] Pierre L'Ecuyer, *TestU01: Testing Random Number Generators*, www.iro.umontreal.ca/~simardr/testu01/tu01.html 2002
- [7] Wikipedia, http://en.wikipedia.org/wiki/Pilot_wave, 2009
- [8] random.org, *RANDOM.ORG - Introduction to Randomness and Random Numbers*, <http://www.random.org/randomness>
- [9] HotBits.org, *HotBits: Genuine Random Numbers*, <http://www.fourmilab.ch/hotbits/>
- [10] hblogo.zip, *HotBits driver* <http://www.fourmilab.ch/hotbits/source/hblogo.zip>
- [11] John Walker, *random.zip* <http://www.fourmilab.ch/random/>
- [12] Nicholas Mc Guire, Qingguo Zhou, *Benchmarking - Cache issues*, Proceedings of the 7th Real Time Linux Workshop, Lille 2005

Use of cookies in real-time system development

M. Gleixner, M. Mc Guire

monika@tglix.de
die.frau@hofr.at

Abstract

While typical scientific works have focused on the technical aspects of real-time development and real-time systems this paper will focus on the caloric requirements that profoundly can impact development and notably scientific dissemination. Though the use of cookies and respective protocols in computer science are well documented [1] we will not cover security aspects, notably related to excessive accumulative effects of consuming large amounts of cookies, rather we will focus on there creation, deployment, assessment and finally there consumption and the positive impact on the real-time Linux community we were able to observe.

Key Words: cookies, cookie life-cycle, assessment

1 Introduction

2 Underlying Principles

Traditionally cookie protocols in computer science have been bidirectional. An in depth analysis of historic records indicates though that the original protocols in fact were unidirectional - bidirectional protocols limited to the error case where cookies were rejected. Unidirectional cookie protocol have a lot of advantages:

- They are lock free
- Simple to implement
- Portable to almost all flavors of real-time developers

To optimize the use of cookies we found that a donation principle is advantageous, notably with respect to the speed of consumption. A donation protocol is typically quite simple and does not necessarily mandate a hand-shake. The only limitation we found was that early donation, that is donation of only partially initialized cookies can lead to negative response of the consumers as well as the provider. Interruption of cookie generation by cookie consumers

also has generally been found to degrade performance of production, thus this should be prevented by utilizing standard locking procedures, locking out consumers at this early stage - for this traditional locking was found to be sufficient.

3 Case study 1

3.1 The RT-Preempt acceleration cookie pool initialization

The RT-Preempt acceleration cookie initialization needs multiple setup functions which are serialized. In theory the initialization could be parallelized to a certain degree but no proof of higher efficiency has been found in the literature. The simplified source code of the initialization procedure is provided below for clarity. Inline functions and defines can be found in appendix A, further technical description in appendix B.

Note that the various error code pathes have been substituted by `panic()` function calls to simplify the code flow. For the unexperienced cookie pool administrator this seems to be the safest option. The recommendation for experienced cookie pool administrators is to patch the initialization functions with

common sense and handle the various error scenarios gracefully, albeit there is no known way to handle the `panic()` in `setup_cookie_incubator()`. As far as we know there is no ongoing research to find a solution for that problem, but it might be possible to add more incubator types. The default case which indi-

cates a non existing incubator will probably remain unresolved for the next decades.

Trivial cookie pool administrator helper functions have been omitted as well and can be retrieved from the standard admin library `lib_cookies-devel`[3].

Listing 1: init code

```
int rt_acc_cookies_init(struct temp_plate *cookie_pool,
                       int cookies_need_bits,
                       int cookies_need_pieces)
{
    struct cookie_mass mass;
    struct base_container *base_container;
    struct content *content;
    struct baselayer *baselayers;
    struct wrap *wraps;
    struct raw_cookie *raw_cookies;
    struct container *container;
    int nr_cookies = 0;

    mass = mass_of_cookies();
    if (check_storage(mass))
        return -EOUTOFSUPPLY;

    setup_cookie_incubator();

    container = find_container(20);

    base_container = init_base_container(mass);
    start_base_timer(BASE_RISING_TIME);

    content = init_content(mass);
    /*
     * Note: init_content() might have taken longer than
     * BASE_RISING_TIME. This is not a failure.
     */
    wait_for_base_timer();

    baselayers = init_baselayers(base_container);
    add_to_baselayers(baselayers, content);

    /*
     * The following functions are conditional and not necessary
     * for the basic variant, but the full flavoured cookie init
     * will call them.
     */
    if (cookies_need_bits)
        add_bits_on_content(baselayers);
    if (cookies_need_pieces)
        add_pieces_on_content(baselayers);

    /*
     * The next step is mandatory to avoid the cookie rejection

```

```

    * due to incompatibility with the consumers intrusion detection
    * system.
    */
    add_cookie_authorization(baselayers);

    /*
     * Convert the baselayers
     */
    wraps = wrap_baselayers(baselayers);

    /*
     * Init the container to ensure that the cookies can be
     * retrieved later
     */
    init_container(container);

    /*
     * Convert the wrapped base transport layers to raw cookies and
     * store them in the container.
     *
     * Note, that the advanced algorithm is to call the combined
     * split and save function split_wrap_and_save() which takes
     * wrap, nr_cookies and container as arguments Omitted here as
     * it requires well coordinated handling of the various
     * resources.
     */
    raw_cookies = split_wraps(wraps, 20);

    save_raw_cookies(raw_cookies, container);

    /*
     * Phew. The hard work is done!
     */
    add_to_incubator(incubator, container);

    /*
     * The conversion from raw_cookies to final cookies takes some
     * time. Set a timer which alerts us to avoid conversion to
     * unconsumable cookies.
     */
    start_incubator_timer(INCUBATOR_TIME);

    /*
     * Premature removal from incubator must be avoided
     */
    wait_for_incubator_timer();

    remove_from_incubator(incubator, container);

    nr_cookies = transfer_to_pool(container, cookie_pool, 20);
    return nr_cookies;
}

static void setup_cookie_incubator(void)
{

```

```
    switch(incubator_type) {
    case TYPE_C:
        init_c_incubator(160);
        break;
    case TYPE_E:
        init_e_incubator(190);
        break;
    case TYPE_G:
        init_g_incubator(3);
        break;
    default:
        panic("No_incubator_found");
    }
}

static struct container *find_container(int nr_cookies)
{
    struct container *available, *cur;

    available = get_available_containers();

    for_each_container(cur, available) {
        if (match_container(cur->capacity, nr_cookies))
            return cur;
    }
    panic("Hardware_error!");
}

static struct base *init_base(ssize_t mass)
{
    struct baseresource *res;

    res = alloc_base_resource(mass);
    if (!res)
        panic("check_storage_is_buggy!");
    return do_init_base(res);
}

static struct content *init_content(ssize_t mass)
{
    struct contentresource *res;

    res = alloc_content_resource(mass);
    if (!res)
        panic("check_storage_is_buggy!");
    strip_content_res(res);
    remove_core_from_content_res(res);
    return slice_content_res(res);
}

static struct baselayer *init_baselayers(struct base_container *bc)
{
    /*
     * Flattening the base needs to be done carefully to avoid
     * root holes in the cookie transport base layer.
     */
}
```

```

        */
        return flatten_base(bc);
}

static void add_to_baselayers(struct baselayer *bl,
                             struct content *c)
{
    /*
     * Even distribution is important otherwise consumers might
     * reject the consumption due to missing content. Minor
     * unevenness is acceptable and provides the individual touch
     * of the final cookies.
     */
    distribute_content_evenly(bl, c);
}

static struct wrap *wrap_baselayers(struct baselayer *bl)
{
    /*
     * Wrap up the baselayers so content, bits, pieces and authcode
     * are safely encapsulated by the base transport layer
     */
    return do_wrap_baselayers(bl);
}

static void init_container(struct container *c)
{
    /*
     * cookies require the container to be formatted with FAT.
     */
    format_fat(container);
}

static struct raw_cookies *split_wraps(struct wrap *wraps,
                                       int nr_cookies)
{
    int length;

    length = wrap_length(wraps[0]) + wrap_length(wraps[1]);
    length /= nr_cookies;

    if (length < MIN_COOKIE_LENGTH)
        panic("Math_error!");

    if (length > MAX_COOKIE_LENGTH)
        panic("Math_error!");

    return do_split_wraps(wraps, length);
}

static void save_raw_cookies(struct raw_cookie *rc,
                             struct container *co)
{
    struct raw_cookie *cur;

```

```
    for_each_raw_cookie_safe(cur, rc) {
        list_del(&cur->list);
        store_in_container(co, cur);
        if (container_full(co))
            panic("Math_error!");
    }
}

static int transfer_to_pool(struct container *co,
                           struct temp_plate *cookie_pool,
                           int nr_cookies)
{
    struct temp_plate temp_plate;
    struct cookie *cookies, *cur;
    int cnt = 0;

    cookies = dump_to_temp_plate(co, temp_plate);

    for_each_cookie_safe(cookies, cur) {
        del_from_temp_plate(cur);
        save_to_pool(cur, cookie_pool);
        cnt++;
    }

    if (cnt < nr_cookies)
        alert("Cookie_underrun_detected!");
    if (cnt > nr_cookies)
        alert("Cookie_overcommit_detected!");

    return cnt;
}
```

3.2 The RT-Preempt acceleration cookie pool distribution

The distribution of these cookies is initiated by the allocation of the cookie pool template on a free accessible storage space and the cookie protocol server is initialized. The advanced server technology intercepts the HTCPCP protocol described in RFC2324[1].

When the developer initiates the GET method of HTCPCP he receives a notification that the Accept-Additions header field has been expanded and needs to be filled in on the request. The default selection for the cookie addition is set to "accept".

Most of the developers tend to confirm the default and therefor receive the cookie as an addition to the content requested over HTCPCP. If the developer who received the first cookie of the fresh cookie pool happens to be in a room with other developers then the advanced HTCPCP server experiences massive concurrent requests short time after the first delivery.

3.3 The RT-Preempt acceleration cookie pool underrun

The time between the start of the advanced HTCPCP cookie server and the pool underrun depends mostly on the number of developers who are able to connect to the server.

A single developer is mostly unable to empty the pool in one series of HTCPCP requests even if he tries hard, but while the cookies should be consumed rather fast due to the instability of the content it is possible to distribute the consumption across several sessions.

A developer team usually empties the pool pretty fast which is partially driven by the fact that each team member fears that it might miss the opportunity to consume enough cookies[6].

In the rare case that the number of developer is larger than the number of cookies per pool allocation the advanced HTCPCP server needs to be instructed by the pool administrator to split the cookies before serving until the first round of requests has been satisfied[7].

3.4 The RT-Preempt acceleration cookies side effects

The consumption of the cookies has shown very positive side effects. The rich content of the cookies requires the full attention of the developers[5] so there is no ability anymore for the usual chit-chat and other counterproductive office habits including phone calls.

Negative side effects are almost unknown except some unexpected prints[4] on the screen which can be traced back to the FAT formatting of the cookie container.

4 Case Study 2 - 1 byte cookies

While much of the core routines of Case study 2 are comparable to those of case study 1 we will focus on design issues and only briefly outline the main differences to the RT-Preempt related example in the previous section.

Case study 2 will focus on nano-kernel compliant solutions. At the lowest level the design parameters are:

- width of cookies - typically 4 bytes are used
- Alignment - one or two dimensional array
- selection strategy - Compare and swap (CAS or CAS2)
- minimizing overall complexity

4.1 Width of cookies

While no significant overhead for decoding 8 or 16 byte cookies was detected, throughput seems to be optimum at 4 byte cookies, we explain this with the fact that resorting to smaller types increases the overhead of access while larger types incur a penalty due to resource contention on the side of the consumer (i.e. increased efforts in garbage collection).

4.2 Aligment

Alignment primarily concerns with the alligment of containers which can impact the performance of key functions like `save_raw_cookies` and the utilization of the incubator. Obviously multibyte cookies require more thought on alignment, notably as unaligned cookies tend to cause exceptions that can infuriate producers. while one dimensional arrays are simple to use (even RT developers can handle this task) they degrade the utilization of incubators and thus two dimensional arrays with proper alligment should be at least concidered.

4.3 Code discussion

These nano cookies have quite obvious performance advantages notably allowing unnoticed transition to multicookie protocols without raising exceptions in other consumers.

Listing 2: init code

```
int nano_cookies_init(int nr_cookies)
{
    struct cookie_mass mass;
    struct base *base;
    struct raw_cookie *raw_cookies;
    struct container *container;

    mass = mass_of_cookies(nr_cookies);
    if (check_storage(mass))
        return -ENOENT; /* strict POSIX! */

    setup_cookie_incubator();

    container = find_container(nr_cookies);

    base = init_base(mass);
    store_base(base);
    start_base_timer(base, BASE_REST_TIME);
    wait_for_base_timer(base);

    init_container(container);
}
```

```
raw_cookies = split_fold(base, nr_cookies);

save_raw_cookies(raw_cookies, container);

add_to_incubator(incubator, container);

/*
 * note that timing is critical due to * these being 1 byte
 * cookies.
 */
start_incubator_hrtimer(INCUB_TIME);

/* non-preemptible section */
spin_on_incubator_timer();

remove_from_incubator(incubator, container);

nr_cookies = transfer_to_pool(container, nr_cookies);
return nr_cookies;
}
```

Functions not described were reused from CS1 so they are not relisted here. Essentially the main difference can be seen in the `init_base()` function which is optimized for the case that resources are available, which is plainly assumed. A minor improvement is to introduce resource conservation by dumping the stack if limits are exceeded and thus operating loss-free by rebasing. Finally fine-tuning of access patterns, i.e. by prefetching have been introduced.

Listing 3: sys cookies core

```
static struct base *init_base(ssize_t mass)
{
    struct baseresource *res = NULL;

    /*
     * don't do this if you are the
     * only one at home !
     */
    while(!res)
        res = alloc_base_resource(mass);

    return do_init_base(res);
}

static struct raw_cookies *split_fold(int nr_cookies)
{
    int length;
    batch_t batch;

    batch = ACCESS_ONCE(base);
    length = base_length(base);
    length /= nr_cookies;

    /*
     * is this possible with 1 byte cookies ?
     */
    if (length < MIN_COOKIELENGTH)
```

```

        panic("Math_error!");

    if (length > MAX_COOKIE_LENGTH) {
        dump_stack();
        rebase(CURRENT | FUTURE);
    }

    return do_split(batch, length);
}

static struct raw_cookies *do_split(struct batch *batch, int length)
{
    int count = 0;
    struct raw_cookie *next, *list;

    while (list) {
        next = list->next;
        prefetch(next);
        csum_fold(next);
        list = next;
        if (++count >= limit(container))
            break;
    }

    if (!count)
        panic("Math_error!");
    else
        raise_batch_soft();
    return list;
}

```

5 Preliminary assessment

A first study of cookie impact was done during the 1st Real Time Linux Workshop in Vienna (TU-Wien 1999) with the general response being positive - in fact in some cases this seems to be the most persistent trace of the event in memory.

To assess the effectiveness of the proposed cookies, preliminary studies were conducted on isolated individuals in Germany and Austria. Positive effects could be found, though it was not clear if these could be generalized. To confirm the findings a mass-study at the 11th Real-Time Linux WorkShop was conducted by exposing a large set of randomly sampled developers to a set of pre-allocated cookies.

5.1 Performance

Unidirectional cookie protocols typically scale $O(N)$ which was found to be sufficient due to the inherent limit of consumers which lead to fairly consistent and strict bounds on N . While multicookie protocols have been observed generally greedy the algorithms were found not to be suitable for cookie protocols,

optimizing for these cases has thus been deferred.

Monitoring of low-water-marks on cookies is essential to achieving a good throughput, unfortunately the cookie production has still a too high latency to allow refill events with small periods, thus pre-allocated pools are the dominant strategy in use.

5.2 Optimization

Surprisingly diversification has shown to be an effective way of maximizing cookie consumption. One observation that could explain this is that we found that multiple concurrent consumers tend to sample different pools, showing a preferred pool depending on specific (local) parameters. Thus the most effective optimization strategy, empirically found, is to initialize sets of diverse cookies, and allow effective parallel access via lock-free mechanisms.

We also found a small number of consumers defaulted to a grab-and-run strategy - while this can't be quantified effectively yet, this strategy does seem to be quite effective.

6 Conclusion

A vast improvement of productivity can be achieved by an effective use of unidirectional cookie protocols for real-time Linux development. Although it can be speculated that such could be deployed in other fields, our current study was limited to real-time developers. Both an increase of productivity as well as biological storage capacity (notably of participating

individuals) could be observed, leading to the conclusion that cookies have a general positive impact on development. Thus we conclude that the use of unidirectional cookie protocols is advantageous and further development of these capabilities is needed.

Though greedy protocols have been discussed in the past, we found that considering these has negative impacts on developers long term and thus are deprecated.

7 Appendix A

Listing 4: header

```
#ifndef OPSYS_METRIC
# error "Operator lives in the wrong universe"
#endif

struct cookie_mass {
    grams_t    wheat_flour;
    ml_t       milk;
    grams_t    yeast;
    pieces_t   eggs;
    pieces_t   apples;
    grams_t    sugar;
    some_t     butter;
    optional_t cinnamon;
    optional_t raisins;
};

static inline struct cookie_mass mass_of_cookies(void)
{
    struct cookies_mass mass;

    mass.wheat_flour = 500;
    mass.milk = about(250);
    mass.yeast = 20;
    mass.eggs = 2;
    mass.apples = about(6);
    mass.sugar = about(100);

    return mass;
}

/* Incubator types */
#define TYPE_C    "convection_oven"
#define TYPE_E    "electric_oven"
#define TYPE_G    "gas_oven"

struct baseresource {
    grams_t    wheat_flour;
    ml_t       milk;
    grams_t    yeast;
    pieces_t   eggs;
```

```

};

static inline struct base_container *do_init_base(struct baseresource *r)
{
    struct bowl *small_bowl, *large_bowl;
    struct base_ball *ball;
    struct board *board;

    small_bowl = get_small_bowl();
    put_yeast_into_bowl(small_bowl, r->yeast);
    add_some_lukewarm_water_to_bowl(small_bowl);
    stir_yeast_and_water(small_bowl);
    put_bowl_aside_for_10min(small_bowl);

    if (!twiddle_thumbs())
        do_something_useful();

    large_bowl = get_large_bowl();
    put_wheat_flour_to_bowl(large_bowl, r->wheat_flour);
    add_eggs_to_flour(large_bowl, r->eggs);
    add_yeast_mix_to_flour(large_bowl, small_bowl);
    move_to_dishwasher(small_bowl);

    do {
        if (!r->milk) {
            if (milk_availabe())
                refill(r->milk);
            else
                refill_with_water(r->milk);
        }
        add_some_of_the_milk_to_flour(large_bowl, r->milk);
        kneed_ingredients(large_bowl);
    } while (flour_not_fully_absorbed(large_bowl));

    board = get_board();
    base_ball = remove_base_ball_from_bowl(large_bowl);
    put_soft_dough_on_floured_board(board, base_ball);
    kneed_until_dough_is_smooth(board, base_ball);
    put_smooth_dough_ball_back(base_ball, large_bowl);
    cover_bowl_with_dish_towel(large_bowl);
    put_bowl_aside_to_warm_place(large_bowl);

    return (struct base_container) large_bowl;
}

#define BASE_RISING_TIME until_base_has_doubled_size

struct contentresource {
    pieces_t    apples;
    grams_t     sugar;
    optional_t  cinnamon;
    optional_t  raisins;
};

static inline void strip_content_res(struct contentresource *r)

```

```
{
    strip(r->apples);
}

static inline void remove_core_from_content_res(struct contentresource *r)
{
    remove_core(r->apples);
}

static inline struct content *slice_content_res(struct contentresource *r)
{
    struct content *slices;

    slices = slice_into_thin_slices(r->apples);

    return slices;
}

static inline struct baselayer *flatten_base(struct base_container *bc)
{
    struct base_ball *dough;
    struct baselayer *doughs;
    int i;

    dough = get_the_dough_from_bowl(bc);
    doughs = split_dough_in_half(dough);

    for (i = 0; i < 2; i++) {
        put_dough_on_floured_board(doughs[i]);
        do {
            apply_rolling_pin_to_dough(doughs[i]);
        } while (length != 40cm && width != 30cm);

        brush_a_bit_liquid_butter_on_flat_dough(doughs[i]);
    }

    move_to_dishwasher(base_container);

    return doughs;
}

static inline void
distribute_content_evenly(struct baselayer *bl, struct content *c)
{
    distribute_apple_slices_on_baselayers(bl, c);
}

static inline void add_bits_on_content(struct baselayer *bl)
{
    add_cinnamon_if_you_like_the_taste(bl);
}

static inline void add_pieces_on_content(struct baselayer *bl)
{
    add_raisins_if_you_like_the_taste(bl);
}
```

```

}

static inline void add_cookie_authorization(struct baselayer *bl)
{
    /*
     * The amount of sugar depends on the sourness of the apples.
     *
     * Note that the dough contains no sugar to have a better
     * contrast between dough and apples.
     */
    distribute_sugar(bl);
}

static inline struct wrap *do_wrap_baselayer(struct baselayer *bl)
{
    int i;

    for (i = 0; i < 2; i++)
        bl[i] = wrap_baselayer_to_long_roll(bl[i]);

    return (struct wraps*) bl;
}

static inline void format_fat(struct container *c)
{
    butter_the_casserole(c);
}

#define MIN_COOKIE_LENGTHT about(4cm)
#define MAX_COOKIE_LENGTHT about(4cm)

static struct raw_cookies *do_split_wraps(struct wrap *wraps, int length)
{
    return cut_the_rolls_into_pieces(wraps, length);
}

static inline void
store_in_container(struct container *co, struct raw_cookie *rc)
{
    roll_the_raw_cookie_in_liquid_butter(rc);
    put_roll_in_casserole_on_flat_side(co, rc);
}

#define INCUBATOR_TIME about(20min)

```

8 Appendix B

Technical description of RT-preempt acceleration cookies

Model name: Apfelschnecken

Alternative model name: Apfelnudeln

Origin: South Germany

9 Appendix C

```

/* times in seconds */
#define INCUB.TIME      8*60
#define BASE.REST.TIME  2*60*60

struct baseresource {
    grams_t    wheat_flour;
    grams_t    vanille_sugar;
    grams_t    almonds;
    grams_t    butter;
    pieces_t    eggs;
};

static inline struct cookie_mass mass_of_cookies(int nr_cookies)
{
    int m = nr_cookies / 40;
    struct cookies_mass mass;

    if (!m)
        panic("nr_cookies_to_low!");

    mass.wheat_flour = m * 180;
    mass.butter = m * 100;
    mass.eggs = m * 1;
    mass.vanille_sugar = m * 50;
    mass.almons = m * 50;

    return mass;
}

static inline struct base *do_init_base(struct baseresource *r)
{
    bowl_t bowl;

    bowl = get_large_bowl();
    push(wheat_flour, bowl);
    push(split_egg(EGG_JOKE), bowl);

    /*
     * notably if child processes are involved
     */
    set(IGNORE_MESS);

    do {
        push(grind(mass.almonds), bowl);
        kneed_ingredients(bowl);
    } while(STICKY | MESSY | !SMOOTH);

    return dough_in_large_bowl;
}

```

References

- [1] <http://www.encyclopedia.com/doc/1G1-128446233.html>

- [2] <http://www.faqs.org/rfcs/rfc2324.html>
- [3] <http://www.powells.com/biblio?isbn=9780764554209>
- [4] <http://www.funnyforwards.com/fortuneteller1.htm>
- [5] http://www.waltsense.com/storage/articles/20090707_MouthFull.jpg
- [6] http://www.amazing-planet.net/slike/rodents/prairie_dog_food_fight.jpg
- [7] http://www.osthessen-news.de/Media/09/08/News090804_3_WettessenHerren.jpg

Linux based demonstrator of a failure tolerant numerical control system

Hans-Peter Bock

Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW)
Seidenstraße 36, 70174 Stuttgart, Germany
Hans-Peter.Bock@isw.uni-stuttgart.de

Prof. Dr.-Ing. Alexander Verl

Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW)
Seidenstraße 36, 70174 Stuttgart, Germany
Alexander.Verl@isw.uni-stuttgart.de

Abstract

In current research work the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) has developed a continuous concept for a redundant, failure tolerant numerical control system. The results of this research project meet the increasing demands on the reliability of machines and plants and also the frequent applications of numerical controls in safety-critical areas. In this case, reliability means the capability of a numerical control to function according to its specification during the entire machining time. This is achieved by a redundant design and will be shown on a demonstrator. This demonstrator based on linux is introduced in this paper.

1 Introduction

In the meantime, numerical control systems are not only used in the conventional applications like milling or drilling. Meanwhile they also control fairground rides or surgery assistants [1]. These new application areas place higher demands on the system reliability than the manufacturing applications. The safety-oriented approach where a machine is stopped in case of a failure is no longer adequate. In the case of fairground rides for example this means that the passengers have to hold out for a longer time in an uncomfortable position. With surgeries a failure would cause a prolongation and possible complications. With certain surgery methods, as for example the endoscopic method with surgery assistants, a small hole is drilled through the skullcap. For a manual intervention the opening would have to be

enlarged considerably [2].

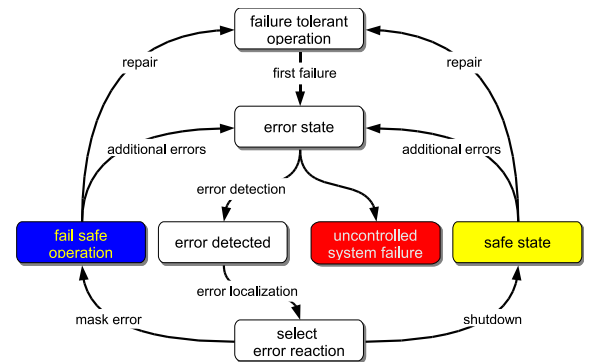


FIGURE 1: *states in a failure-tolerant control system [3]*

Figure 1 shows the intended states in a failure-tolerant control system [3]. Usually the control system is in a state of failure-tolerant operation. After the first failure the system is in a state of failure. If the failure is not detected, the system will sooner or later fail. This must be avoided. Therefore, it is nec-

essary to identify occurring failures and to localize them. After the localization of the failure, an adequate reaction has to follow. With safety-oriented systems a safe state is aimed for, which is mostly achieved by stopping the machine.

If a safe state can only be reached after the machining process has finished, a shutdown of the system cannot be accepted. In this case the operation has to be continued, which can be achieved without interruption by masking the failure. Masking a failure means that faulty values are replaced by correct values by several and independent repetitions of the same calculations. After the masking the system is only in a failure-safe operational state, but it can still continue with the machining.

The objective of this research work is the increased reliability of the control system. This means that the specified function of the control, the generation of correct setpoints, is still maintained in spite of occurring failures. Here the masking of failures during operation is necessary.

2 State of the art

In control systems there are already many mechanisms for failure detection and the reaction to them. Figure 2 shows the common methods for the various components.

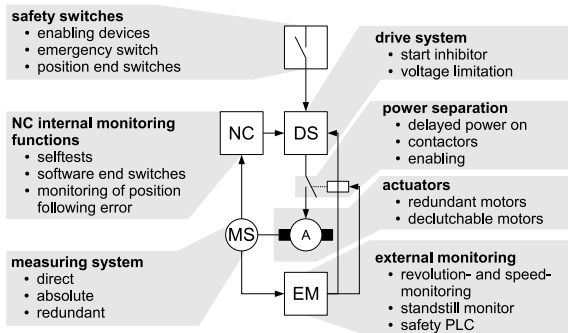


FIGURE 2: common methods for failure detection for the various components [3]

For all components there are relating failsafe and redundant parts. With redundant measuring systems the measured values can be checked for plausibility and thus individual sensors can be tested for their error-free function. Monitoring units check whether certain measuring values surpass their permitted value range and take safety measures if needed. Actors can be redundant in order to compensate the failure of a particular actor.

But for the control itself, the central component of the control system, there is no overall failure detection. It is nearly impossible for the rest of the components to detect that the control generates faulty setpoints. The existing measures for error detection in the control like self-tests, software end switches, etc. are suitable to only a limited extent for the detection of all failures. For masking the failures they are not sufficient.

2.1 Failure classification

As shown in figure 3, failures can originate from three different places. Outside the system, in the the system hardware or in the system software. Failures outside the system are failures of the basic conditions, which the system relies on. One example for that is the power supply. Because failures outside the system can independently be detected and handled, this work focuses on failures which originate inside the system boundaries.

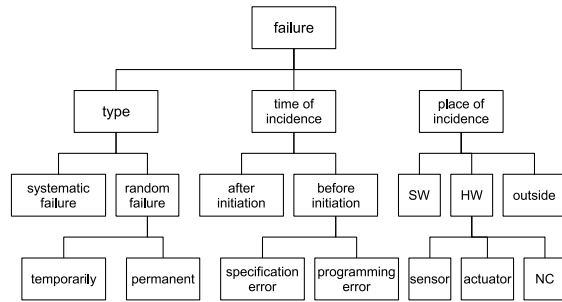


FIGURE 3: classification of failures [3]

Failures inside the system can either be present in the software or occur in the hardware. Software failures are generally systematic failures. They consist of specification- or programming errors and get into the system before its initiation. An error free software will never generate wrong values since software does not grow old. But a software containing errors will always generate wrong values.

Software errors can either be avoided before the initiation or be detected during operation. Actions to avoid software errors are software analysis, failure simulation tests or the usage of special tools which analyse source code. Actions to detect software failures is additional monitoring during operation like range or plausibility checks. Another possibility to detect software failures during operation is to cross check the values computed by redundant and divers developed software.

Hardware failures can either be systematic or random. Systematic hardware errors happen before

the initiation. They originate of specification, design or programming errors. These systematic hardware failures can be detected during operation only by divers and redundant hardware.

Random hardware failures occur during operation. They can be detected by several self tests which check e.g. the processor or memory. It is difficult to run these self tests during operation, since the control will need most of the system resources for itself. With redundant hardware it is possible to detect random hardware failures during operation by cross checking the computed values.

The main focus of the work is put on the masking of systematic and random hardware failures. In order to safely identify hardware failures, the control hardware needs to be designed redundantly and diversly. Redundantly, so that an accidental failure is identified by the redundant control system. Diversly, so that systematic specification and production failures do not occur in the redundant systems at the same place and at the same time.

In aviation as well as in mechanical engineering such redundant and divers control systems are employed in safety-oriented programmable logic controls (PLCs) with "Fly by wire". For particularly challenging applications of PLCs there is a failure-tolerant type with triple redundancy, the so called "Triple Modular Redundancy". For numerical controls there is up to now only one failure-safe version. It will be presented in the following section.

2.2 The concept of the failsafe control system

Figure 4 shows the existing concept of a failsafe numerical control system [3]. It consists of two numerical controls which run on divers hardware platforms. The two controls are synchronized by a shared memory by which they also exchange computation results. The machine only receives setpoints by control #2.

Each of the two controls contains a reaction manager. He compares the own computed values with those of the other control. If he notices differences, he can autonomously disable the actors.

If in this control system a failure occurs in one of the hardware platforms, it will lead to falsely computed setpoints. Even if the failure is so extensive that the hardware platform cannot recognize it anymore, the redundant platform will diagnose this and

bring the machine into a safe state by shutdown.

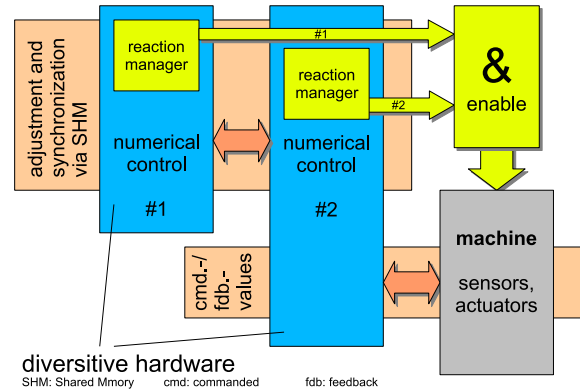


FIGURE 4: *concept of a failsafe numerical control system*

3 Deficits

The design of the failsafe numerical control is satisfactory for applications in which by machine shutdown a safe state can be achieved at any time. In failsafe control systems failures may be detected reliably, but the reaction manager cannot localize them in the case of differing values. Due to the lacking failure localization he is not able to mask a failure. There is only the possibility to bring the machine to a safe state. The machining process is interrupted. For applications in which the operation need to be maintained over a specific period, this design is not adequate.

4 Approach

Failures that are not easy to locate are in other application areas - like in the control of airplanes in aviation or in safety PLCs - localized and masked by multiple redundancy. This approach is very complex because three or more systems have to deal with the same task. The computation results of the systems are compared and differences are masked by majority decision. This high expense is only justified when a high reliability of a system is required. Such requirements are less often encountered with conventional machine tools, but more so with medical surgery assistant systems.

In contrast to the safety PLCs which compute a simple logic between input and output, a numerical control conducts per millisecond complex calculations for the timely output of the correct position data. Compared to the failsafe numerical control, the majority decision for masking faulty setpoints is new.

Contrary to a failsafe numerical control the failure-tolerant system needs at least three divers controls. This design is shown in figure 5 [4].

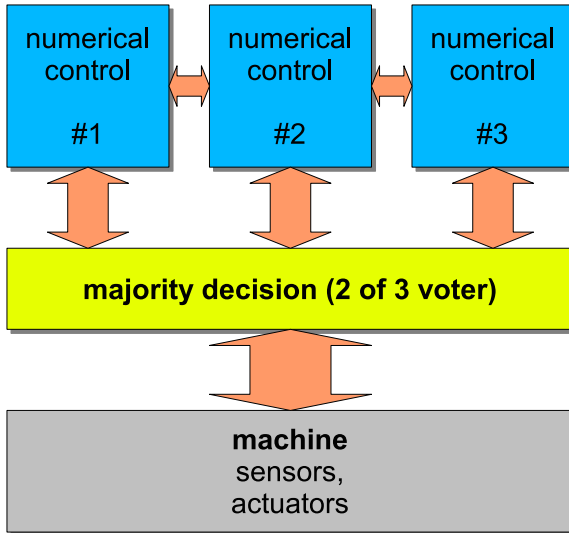


FIGURE 5: *majority decision as approach [4]*

4.1 Voter in every actuator

Figure 6 shows a failure tolerant control layout, in which every single actor contains the necessary voter. All voters receive the setpoints of all controls to pass one setpoint chosen by a majority decision on to the actuator. In case of a failure of a single actuator, the control can mask this failure using redundant actuators.

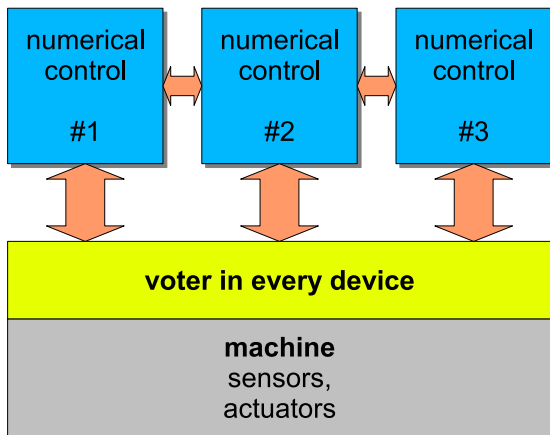


FIGURE 6: *voter in every single actuator*

The advantage of this layout is, that the actuators already contain software. This software can be extended to run the majority decision. A disadvantage is, that it might happen, that different

actors use setpoints of different controls. This can be avoided by synchronization between all actuators which results in a more complex system setup.

4.2 Voter inside communication system

Another possible layout is shown in figure 5. The actuators are not modified. The majority decision takes place inside the communication system. The result is passed on to the actuators.

The advantage of this layout is, that the actuators do not need to be modified. The disadvantage is, that communication systems usually do not possess any computation power. The communication system would have to be extended at an appropriate place to run the majority decision. This extension also needs to be redundant, so that a failure of it does not lead to a breakdown of the whole control system.

4.3 Voter inside numerical controls

Figure 7 shows the possible layout to place the voter inside the numerical controls. The connection from each numerical control to the communication system can be interrupted by the other numerical controls. In normal operation, the first numerical control provides its setpoints to all actuators. The other numerical controls will do the same calculations and synchronize themselves to the first one. If one voter notices incorrect setpoints, it will try to disconnect the faulty control from the communication system. If two controls do this, the faulty control will be disconnected. If the disconnected numerical control has been the one providing setpoints to the actuators, one of the remaining controls will resume this task.

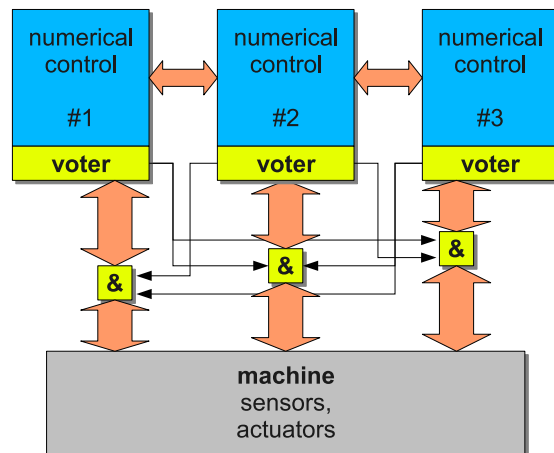


FIGURE 7: *voter inside numerical controls*

The advantage of this layout is, that the actuators do not need to be modified. The voter can easily be integrated into the numerical control systems. The disadvantage is, that the communication system needs to be extended, so that it can disconnect single controls and switch over to another one.

5 System design

For the demonstrator, the failure tolerant layout with the voter in every actuator, figure 6, is chosen. This layout does not need any extensions for the communication system. It also does not insert a single point of failure into the system, as long as redundant actuators are provided.

To work around the difficulty of extending the firmware of actuators, no real machine is applied for the design of the demonstrator. Rather the software package Virtuos by the company ISG is used. This software package allows real hardware in the loop simulation of a complete machine [5, 6] as well as the integration of the required majority decision.

As operating system, linux with rt-preempt has been chosen for its excellent availability and costs. The missing safety certification does not disturb at this point, since this demonstrator is a proof of concept.

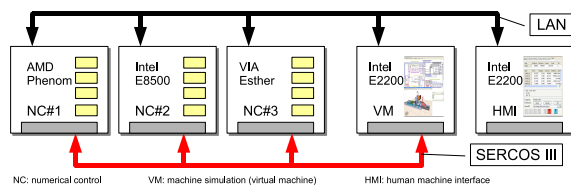


FIGURE 8: design of the demonstrator

Figure 8 shows the design of the demonstrator which is based on five conventional PCs with divers hardware as well as network and SERCOS III interface cards for communication. Each of the first three PCs is running linux with a numerical control kernel. One of them has a SERCANS III master pci card from Bosch Rexroth and runs the free linux kernel driver¹. The two others have a SERCOS III slave pci card from Automata with a proprietary slave stack. The fourth PC runs linux and simulates the machine to be controlled with the solver of Virtuos. Connected are these PCs by the ethernet-based industrial communication system SERCOS III. The fifth PC runs

windows for the user interface of the controls and Virtuos as well as the visualization of the simulated machine.

6 Conclusions

The demonstrator setup shows how to redundantly operate numerical controls. The new cross communication concept of SERCOS III is used for the exchange of data as well as for synchronisation. Errors as well as the breakdown of a whole control is tolerated by the concept of synchronised numerical controls and the majority decision based on setpoints. The failure tolerance of the demonstrator will be shown by error simulation tests.

The presented failure tolerant system enables numerical control systems to gain higher reliability. A machine vendor trades of this advantage against a considerably complex system design along with raised costs. Nevertheless, a failure tolerant system is justifiable, if a very high reliability is required. This high reliability is required in fairground rides or in medical applications, e.g. surgery assistants.

References

- [1] G. Pritschow, K.-H. Wurst, A. Haubelt, G. Bauer; *Ergonomisch gesteuertes Chirurgie-Assistenzsystem*; wt Werkstatttechnik online 94; Düsseldorf; 2004
- [2] A. Hein; *Eine interaktive Robotersteuerung für chirurgische Applikationen*; VDI-Verlag; 2000
- [3] U. Laible; *Aufbau numerischer Steuerungssysteme für sicherheitskritische Anwendungen*, Jost-Jetter Verlag, Heimsheim; 2005
- [4] A. Verl, H.-P. Bock; *Fehlertolerante numerische Steuerung*, wt Werkstattstechnik online, H.5, Düsseldorf; 2006
- [5] G. Pritschow, S. Röck; "Hardware in the Loop" *Simulation of Machine Tools*; Annals of the CIRP 53, S. 295...298; 2004
- [6] S. Röck, H. Rüdele, G. Pritschow; *Real-time simulation environment for machine-tools*, Virtual Modelling and Rapid Manufacturing, S. 285...290; 2005

¹SERCANS III linux kernel driver announcement: <http://tinyurl.com/luapuv>

An Open Implementation of Profibus DP

TRAN Duy Khanh, Pavel PISA, Petr SMOLIK

Czech Technical University in Prague, Faculty of Electrical Engineering

Department of Control Engineering

Karlovo namesti 13, 121 35, Praha 2

info@pbmaster.org

<http://www.pbmaster.org>

Abstract

This paper presents a project named PBMaster, which provides an open implementation of the **Profibus DP** (*Process Field Bus Decentralized Peripherals*). The project implements a **software implementation** of this very popular fieldbus used in factory automation. Most Profibus solutions, especially those implementing the master station, are based on ASICs, which require bespoke hardware to be built solely for the purpose of Profibus from the outset. Conversely, this software implementation can run on a wide range of hardware, where the UART and RS-485 standards are present.

1 Motivation

Profibus is a fieldbus that can be used both in production automation and process automation and which has become a global market leader. Worldwide, over **28 million Profibus devices** were installed by the end of 2008.

Although the Profibus was initially standardized in the late 1980s it is not easy to find materials to help design and system engineers develop new products. Despite many interesting features like deterministic media access or fast data exchange, Profibus still seems to be only in the domain of professional applications and commercial solutions. The main reason is probably due to the high price of all Profibus products, whether hardware or software solutions.

Commercial solutions commonly need special hardware, and the software is mostly proprietary - which also significantly increases the price of Profibus applications. In addition, the primary supported platform for these solutions is the operating system MS Windows. Support for Unix-like systems is very low.

2 PBMaster project

Into this marketplace, **PBMaster** [1] comes with its solution within the field of this popular industrial bus. The project is still under extensive development, but aims to offer a cheap solution that will make it possible to use Profibus not only in commercial applications, but also in universities, homes or semi-professional applications. The key to achieving these objectives is in using common **inexpensive hardware** and **open source software**.

Initially, the project aimed to offer a solution for connecting common personal computers running Linux to this industrial bus. Now, it runs on several operating systems (**Linux**, **FreeBSD** and **NetBSD**) and sys-less embedded hardware based on the **ARM** architecture. The objectives are to offer multi-platform drivers, libraries and applications capable of carrying out the master, slave and analyzer functions of the Profibus. In the future the project will try to offer a complex and inexpensive software based solution for applications using Profibus.

The following picture describes the software structure of the PBMaster project. We will take a closer look at each component later in this paper.

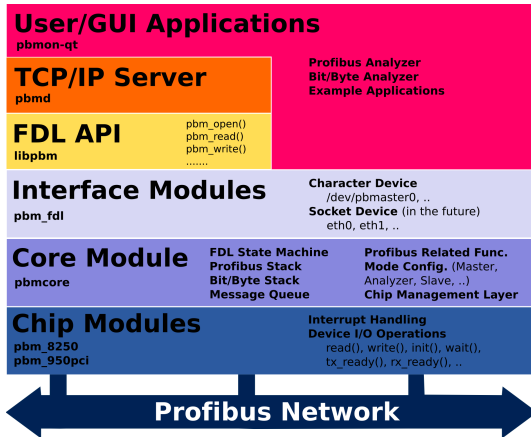


FIGURE 1: *Software structure*

3 Open source vs. Profibus patents

The project was established to offer an **open implementation** with all the components to be released under the GNU GPL v.2 or a later version. Earlier this year **distribution of the source code was been suspended** due to patent problems preventing the distribution of the implementation under an open-source license. The patents appeared this year even though they were applied in 1992. Two of the three patents related to the Profibus implementation affect the FDL master implementation within the PBMaster project.

Despite very positive feedback from the Profibus experts after the presentation of the project at the Profibus Conference in Krakow in July 2009, the PI (*Profibus International organization*) and patent holders declared they did not want to support the community and repeated that **Profibus is an “open” but not “open-source” standard**. The PI grants rights for using the patents to Profibus members. This means the **PI members can use PBMaster’s implementation**. The situation unfortunately does not allow distribution of the master implementation, which covers a significant part of the project, in the community until the patents are valid.

4 Hardware System Structure

As mentioned before, implementing the standard by software makes it possible to run on a wide range of hardware including that which was not designed

for the purpose of Profibus. It is also one of the means by which the solution can be made inexpensive. The software implementation can run on hardware integrating UART (*Universal Asynchronous Receiver/Transceiver*) circuits with RS-485 output, the physical standard used by Profibus DP. The current version supports three types of hardware. We will discuss them in the following sections.

4.1 RS-232/RS-485 Converter

The converter offers an economic solution in order to connect a PC to a Profibus network. The essential requirement is for the connected PC to have a serial port, which could be a problem as modern computers lack this communication port. The electric level of RS-232, generated by the common PC, is converted using this dongle to RS-485, used by Profibus DP. The dongle is powered directly from the serial port of the PC. It is the reason why the converter is very small with very few components integrated. The lack of external power supply limits use of the dongle in very extensive networks with long cabling; furthermore, it’s not possible to use cable termination and the number of nodes on the network is limited to around 10 nodes.

A highest attainable Profibus speed of 19200 bit/s is another limit of this converter. The limitation is caused by differences in speeds supported by RS-232 of the personal computer and the speeds supported by Profibus DP. This converter can be run in mode Master and Analyzers. It is a very low cost solution. All schematic and assembly materials are available in the project’s repository.

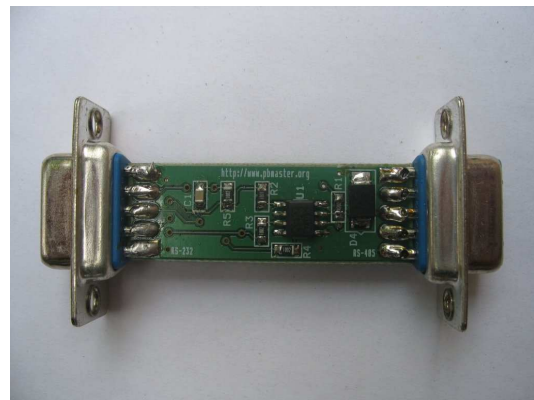


FIGURE 2: *RS-232/RS-485 Converter*

4.2 PCI based cards

In order to achieve higher speeds it is necessary to use some UART integrated extension cards. The current drivers support PCI based cards integrating the OX16C954 chip, probably the fastest UART integrated circuit with a PCI interface on the market. It is a high performance UART with 128 byte FIFOs by Oxford Semiconductor. The IC integrates a 16C550 compatible UART offering an ample FIFO size and many other interesting features. One of the most important features is the maximal bus speed of 15 Mbit/s in normal mode and 60 Mbit/s in external clock mode. Using this IC the drivers **achieve a communication speed of 12 Mbit/s**, the maximal speed of Profibus DP. The PCI card contains up to four ports. Each port can run in Master or Analyzers mode up to 12 Mbit/s independently of the others. The PCI card does not allow usage in the Bit analyzer mode at the moment as there is no built-in hardware support (could be added easily).

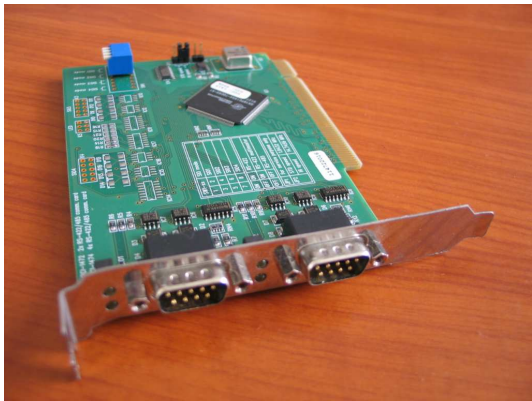


FIGURE 3: *UART Integrated PCI Card*

4.3 ARM based boards

Having the option to use the Profibus drivers on an embedded system is especially interesting. This option makes connecting to the Profibus network easy, cheap and removes the need for big hardware support. Reading sensors and controlling actuators would be very easy and available for common applications like home automation, robot controlling or even control of a production line. The embedded system offers a lot of options for creating a bridge between different standards, as the micro-controllers usually offer many types of interfaces.

The following figure shows a board integrating an LPC 2148 ARM with UART interfaces by NXP Semiconductors. The highest tested Profibus speed

was 500 kbit/s (limited by the transceiver). It is expected to achieve a Profibus speed 1.5 Mbit/s or higher with a faster transceiver. The software runs on a target without operating system. It is a robust solution and can be used in Master or experimentally in Slave mode.

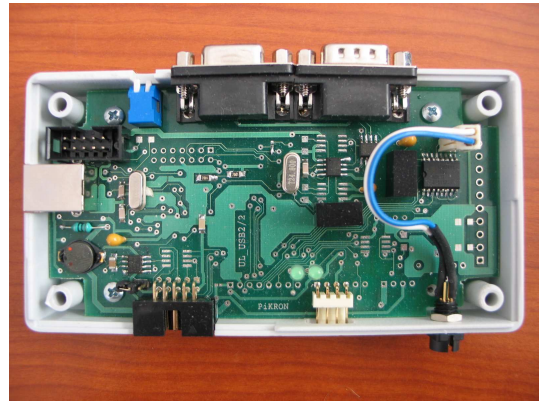


FIGURE 4: *Embedded Board Integrating an ARM LPC 2148*

This very tiny module integrating ARM7 by Atmel offers many interesting interfaces like Ethernet, USB 2.0, SPI, I2C, CAN transceiver, UART + RS-232/RS-485 transceivers etc. It runs FreeRTOS, a highly portable real-time operating system. Profibus support for this module is under development and is expected to achieve very high speeds thanks to the DMA channel and very high speed RS-485 transceiver.

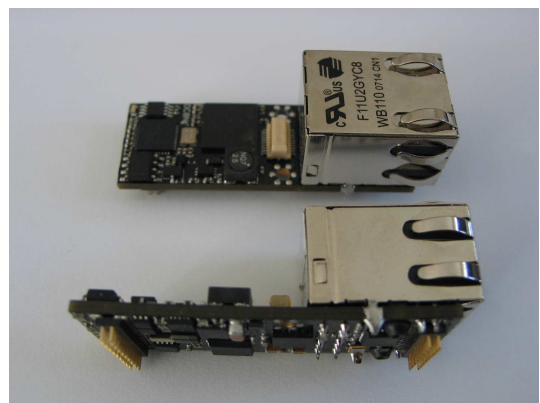


FIGURE 5: *ARM7 board running FreeRTOS*

5 Device Drivers

Almost all hardware needs software to support the functionality it was designed for. In our case, this

takes the form of a device driver implementing low-level support called FDL (*Fieldbus Data Link*). Most of the hardware supported solutions use ASICs to implement this FDL layer. In contrast, the PB-Master solution is implemented solely by software. Along with the FDL layer the drivers cover also the low-level part of the Profibus analyzer and UART Bit/Byte analyzers. The software structure was designed to be as modular as possible. The modularity allows porting to other platforms without extensive modification of the base part. The created framework offers a simple way to write driver support for new hardware.

The current driver design is divided into three groups of modules:

The core module – (*pbmcore*) – is the main driver implementing the basic device data structure, chip related defines, a bit/byte stack implementation and the most important part, the Profibus FDL stack. The implementation of Profibus covers *Finite State Machines* of an FDL Master station, FDL Slave station, low-level part of the Profibus Frame Analyzer and UART Bit/Byte Analyzer.

The module is platform and hardware independent and by itself does nothing but offer functions to other chip drivers. This allows use of several chip drivers simultaneously without any code redundancy. In addition it makes possible supporting new hardware without deep knowledge about the Profibus standard.

Chip drivers – (*pbm_950pci*, *pbm_8250*, ..) – implement **basic I/O operations** with real hardware like chip initialization, read, write etc. The chip module, upon loading, registers to the core module these basic operations and provides hardware and system dependent support for things such as resource allocation or interrupt registration. The remaining actions are controlled by the core module through the state machine.

User space interface module – (*pbm_fdl*) – the third group of modules implements an interface between user applications and the FDL layer in the kernel space. Currently, the module is a **character device** but in the future a **socket module** will be developed to allow communication using a socket-based technique. An application writes its request to, and reads responses from, the device. This data is passed to the core module and then is addressed correctly to an appropriate chip device. Regardless of whether the user chooses to communi-

cate using a file or a socket, the API between an application and a kernel is unified, as well as the API between an interface module and the core module.

6 FDL API

Once the drivers are installed, communication with another station on the bus is provided by sending bytes in the **raw form** directly to the Profibus device. Even though it is simple to realize communication in this way, the fact that a developer needs to understand frame structures, and even the Profibus specification, make it unsuitable and inconvenient for most application developers. Hence an API was developed to offer a unified and simple mechanism to access the Profibus network.

The API is provided in the form of a library. The library offers a unified system and architecture-independent programming interface handling the transfer between FDL applications and Profibus stations.

The project had a plan to fully support the FDL programming interface by Siemens. In the end, backwards support was not implemented due to the proprietary license of that programming interface. Nonetheless, the project's API partly supports backward compatibility with the Siemens's FDL programming interface. Everything related to Siemens's Request Block should be removed as the library does not work with that structure. Replace the Request Block with a buffer. This backward support is implemented as macros calling the proper API functions described previously. Mapping between the project's API and the API by Siemens is listed below:

pbm_open	SCP_open
pbm_close	SCP_close
pbm_write	SCP_send
pbm_read_poll	SCP_receive
pbm_errno	SCP_get_errno

7 Profibus DP

The Profibus DP (*Decentralized Peripherals*) layer is based on top of the FDL layer. There are three groups of specification for the DP layer. The basic version *DPV0* should be supported by all DP devices which involve support for Cyclic Data Exchange and Diagnostics. The *DPV1* extensions are an integral part of the Profibus PA specification providing Acyclic Data Exchange, Process Alarm Han-

dling etc. Extension *DPV2* introduces additional enhancements that are used in high-speed servos and drives and in functional safety systems. *DPV0* will be supported by the project and its implementation is under development. This will be an important step for certification of the implementation.

8 TCP/IP Server

Until now, the software supported only Unix-like operating systems or embedded ARM based systems. With the server, any station supporting TCP/IP can connect to the Profibus network remotely.

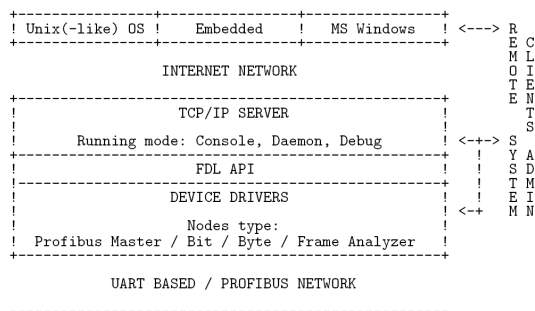


FIGURE 6: Client-Server Architecture

The figure illustrates a client-server model. The server is installed on the machine along with the device drivers. The device drivers implement installed nodes on the bus in several modes. The server provides access to devices created by the drivers over the Internet network. On one side it uses the FDL programming interface for accessing those devices. On the other side socket communications are established to serve the client demanding remote access to devices.

9 Bus Analyzer and Monitor

Applications in industry often demand for tracking problems on the fieldbus. The problems can be diagnosed and localized by listening to and analyzing communication on the bus. By analyzing the captured data, it is possible to detect problems like frames with errors (often caused by signal reflection), station inactivity, address collisions, too short Slot Times of the Master stations, even bad timing of stations.

This section introduces a graphical analyzer and monitoring program. The program was written in QT and as with other components of the project, was

designed to be fast, reliable and modular. Thanks to its modularity, it will be more simple to integrate new features into the program so it could become a versatile industrial bus control and monitoring program. The current version supports Profibus Frame analyzer and UART Bit/Byte analyzer.

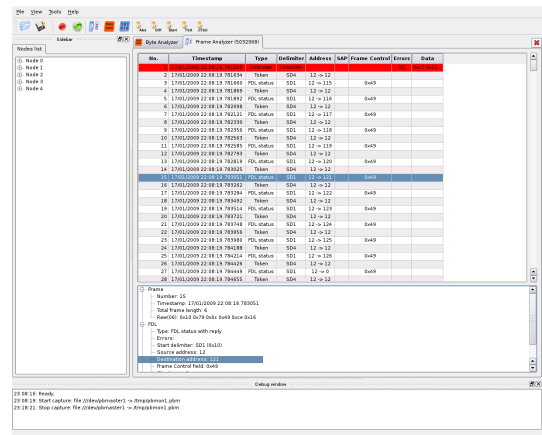
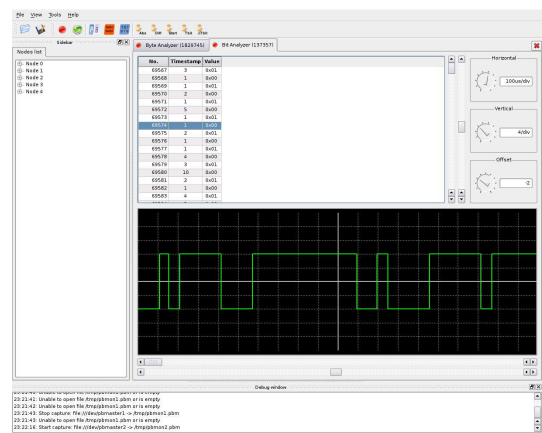


FIGURE 7: Profibus FDL/DP Analyzer

The main window contains menus, tool bars, side bar, debug window and tabs with specific functions. The global buttons like *Start Capture*, *Refresh View*, *Timestamp settings* affect only the visible tab. The debug window is collective for all tabs. The side bar shows information about the nodes. Several tabs of the same type can be present at the same time. The analyzer offers other interesting features like on-line view during capture, offline view of saved files and timestamp display in five formats. Timestamp can be in *absolute time*, *time difference between data*, *time from the start of capture*, *bit time difference between data and bit time from the start of capture*. Capture trigger, view filter and statistics of captured data will be implemented in the next version.



All addresses are specified using the URL mechanism (Uniform Resource Locator). A URL identifier could be `pbm://server.pbmaster.org:11000/0`, `file:///dev/pbmaster0` or simply `/dev/pbmaster0`. The first URL refers to a device number 0 on a remote server using port number 11000, the latter two refer to a local file.

10 Live Linux CD

The PBMaster project is an effort to develop and maintain an open-source implementation of Profibus DP. Running in open-source software has also disadvantages. Probably the biggest disadvantage is that almost everyone using computers is familiar with commercial operating systems like MS Windows or Mac OS, but not everyone has experience nor the ability to get on with Linux, BSDs, etc. Despite rapid improvements and the spreading use of open-source software, applications based on commercial software and MS Windows are still dominant and this situation applies especially for Profibus industry.

That is why the project has developed a **Live Linux CD** offering an easy way to use the Profibus solution without need for software installation. It is not necessary to install any operating system nor other program - just plug in the hardware, insert the Live CD and start working with Profibus.

The CD is based on the Debian Lenny [6] distribution. It contains utilities and useful tools like Xfce desktop environment and its programs, Firefox internet browser, Kate text editor, Openoffice.org office suite, OpenSSH client and server, GCC compiler collection and Make utilities, Midnight Commander, Vim editor, GIT, CVS, SVN, media player and many more programs including all necessary libraries to compile the project's components. The number of packages in total is more than 750.



FIGURE 9: *Xfce Desktop with Profibus Utilities*

Imagine having an environment supporting Profibus and other UART utilities in just a few minutes. There is **no need for installation**, no manual configuration required - just put the Live CD into your drive and reboot the computer.

11 An Overview of Applications

The figure shows a model production line running Profibus using PBMaster. The drivers implement a master station with an integrated PCI card, based on an OX16PCI954 chip, running at 1.5 Mbit/s. It is possible to run up to 12 Mbit/s without any problem. The speed limitation is caused by one slave station. Actuators and sensors are connected to slave stations by WAGO.

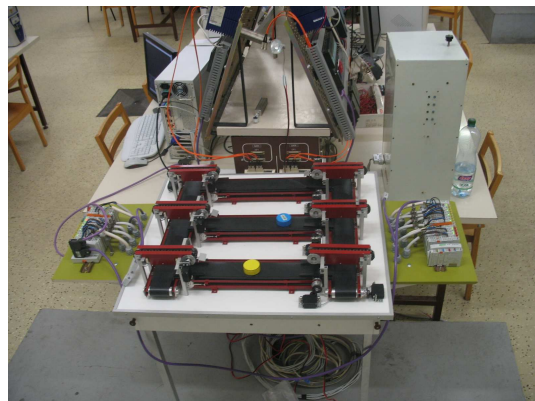


FIGURE 10: *A Model Production Line*

The figure shows several Profibus networks. One network consists of an ARM based board running in master mode, a slave station with digital inputs/outputs and an analyzer running in bit mode. The network communication speed is 500 kbit/s. The second network consists of two Profibus nodes communicating at 12 Mbit/s. One node simulates a master station, the second node runs in frame analyzer mode. The rest 4 unconnected ports are masters running at 187.5 kbit/s, 500 kbit/s, 1.5 Mbit/s and 6 Mbit/s. All nodes run simultaneously.

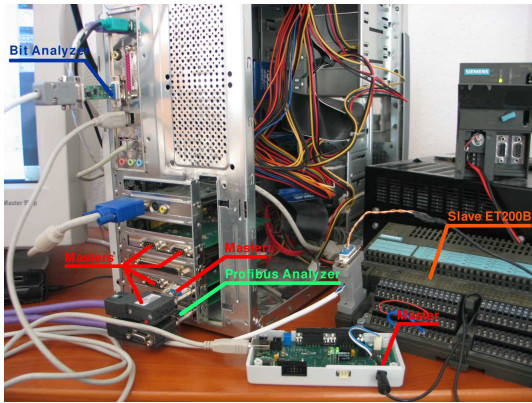


FIGURE 11: *Master-Slave Communication and Bus Monitoring*

The figure shows two Profibus networks. There is one master, one slave and one bit analyzer on the first network running at 19200 bit/s. The second network consists of one master and one frame analyzer station running at 12 Mbit/s. Data from the analyzer is sent over the Internet network to the analyzer and monitoring program running on a notebook. More examples can be found on the project's website [2].

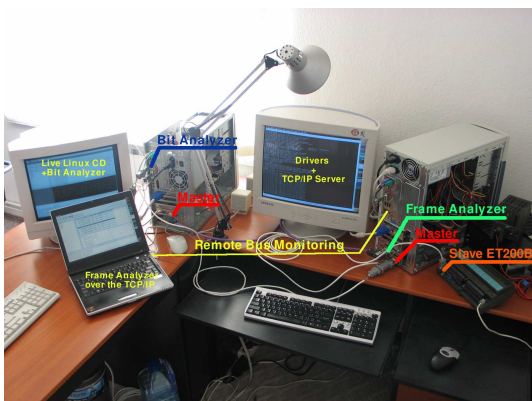


FIGURE 12: *Remote Bus Monitoring*

12 Friendly projects

I would like to mention two projects, **uLan** and **ProfIM**, which have similar objectives as PBMaster. Some solutions and ideas used in PBMaster have been taken from these projects.

The project uLan [4] provides a 9-bit multi-master message oriented communication protocol, which is transferred over the RS-485 link. Characters are transferred in the same way as for the RS-232 asynchronous transfer except for the parity bit,

which is used to distinguish between data characters and protocol control information. The physical layer consists of one twisted pair of leads and RS-485 transceivers. The project is developed and maintained by Pavel Pisa and Petr Smolik. For more information please refer to the project homepage.

ProfIM [5] is a project implementing simulation of master stations using only UART integrated PCI cards or a simple converter of RS-232/RS-485. It leads to an inexpensive solution as there is no need of any expensive proprietary hardware nor software. In addition it offers an FDL programming interface compatible with the API by Siemens. The project supports only Windows OS. One of the disadvantages is that the project has been inactive since 2004. The driver was not designed to be modular and from my point of view it is inefficient and even the stability of the driver is not favorable. The project was developed by Pavel Trnka and maintained by Petr Smolik. For more information please refer to the project homepage.

13 Compiling with OMK

At present, the device drivers are built inside the source code directory using system's default make scheme. On the other hand, the project's applications are built out of the source tree using Make system called OMK (*Ocera Make System* [7]).

OMK [3] is a Make system developed and maintained by Pavel Pisa and Michal Sojka from the Department of Control Engineering at Czech Technical University in Prague under the *OCERA project*. The main objective of the OMK system is to simplify compilation of components on the host machine as well as cross compilation for the target. In addition the system provides for a better directory and file structure. The make system allows building out of source trees and storing results of the compilation in a separate directory structure to simplify testing and program installation.

A key solution is to have a central Makefile with compilation rules for most sub-components and components. This solution allows faster and smoother changes to the system, such as kernel updates. Having most rules in a central file, Makefiles in source directories can be very simple.

OMK was not designed to support the BSD systems. A number of changes have been made in order to use this make system on FreeBSD and NetBSD. Although not all OMK features are available yet under FreeBSD/NetBSD, it is possible to compile all

project's applications using this system. The OMK system is particularly useful for cross-compiling to the final architecture (e.g. embedded ARM).

14 Advantages and disadvantages

One of the advantages and a strong point of the project is based on the modular design of the software. It makes the implementation portable and multi-platform. The software achieves 12 Mbit/s with very good performance. Thanks to the software implementation, applications based on RS-485 could be switched to use the Profibus, therefore it is not necessary to invent a new protocol in the application based on this physical standard.

There are of course many disadvantages of the software implementation. Even though it works reliably in most cases, it is not possible to guarantee fully deterministic responses on common operating systems due to their design (e.g. when the CPU is occupied by handling a lot of interrupts from another devices). This could be solved by using Real-Time operating systems (tested on Linux with Real-Time Preempt patches). Another option to achieve fully deterministic responses is to use embedded boards without operating system.

This is a software implementation without hardware support. It means there is no oscilloscope mode nor speed detection by analyzing the signal. Another very important disadvantage is that this implementation is not yet certified (under development).

15 Conclusion and Future Work

An open implementation of Profibus has been presented throughout this paper. One of the stated goals for the project was the creation of multi-platform software capable of running on systems with operating systems as well as embedded boards without OS. The components were designed to facilitate porting to new platforms. The software is implemented to be fast, reliable and requiring minimal system resources.

Currently, the project offers an implementation of Profibus FDL master stations, FDL slave stations, Profibus FDL/DP frame analyzer, bit and byte analyzer for UART based bus, an FDL programming interface, Live Linux CD, a TCP/IP server for remote access, a set of examples and free Profibus documen-

tation. The software runs on Linux, FreeBSD and NetBSD operating systems as well as ARM based embedded systems. A few open hardware solutions are also available.

There are many applications, where Profibus can help to make our life more comfortable. The Profibus can be use for example in automatic door systems, power saving systems, temperature regulation in buildings, etc. Application of this open solution may be appreciated by universities and companies, as it is currently necessary to have very expensive commercial hardware and software for any experimentation with Profibus.

Despite these achievements, there are still many problems to be solved to facilitate use in professional applications. The future objective is to implement the Profibus DPV0 layer and to create an unofficial certification of the implementation. Even though the slave abstraction layer is working, there is still a lot of work to be done to improve it. The project will extend support for new hardware and operating systems like FreeRTOS and MS Windows. Libraries, graphical analyzer and other applications will be improved.

The problems related to Profibus are very complex and much effort is required to overcome them. My hope is that this project will make more people familiar with this popular fieldbus solution, so the extent of use of the Profibus standard would not be limited only to commercial applications. The software implementation can **spread the use of the standard** to applications, where hardware supported solutions are inconvenient or not possible.

References

- [1] *PBMaster Project*, Tran Duy Khanh, <http://www.pbmaster.org>
- [2] *PBMaster - Wiki Pages*, Tran Duy Khanh, <http://wiki.pbmaster.org>
- [3] *OMK Project*, Pavel Pisa, Michal Sojka, <http://rtime.felk.cvut.cz/omk/>, CTU in Prague
- [4] *uLan Project*, Pavel Pisa, Petr Smolik, http://cmp.felk.cvut.cz/~pisa/ulan/ul_drv.html, CTU in Prague
- [5] *ProfiM - Profibus Master*, Pavel Trnka, Petr Smolik, <http://profim.sourceforge.net>, CTU in Prague
- [6] *Debian Project*, <http://wiki.debian.org/DebianLive>
- [7] *OCERA Project*, <http://www.ocera.org>

RTAI based measuring and control over EtherCAT

Fodrek Peter
Kamensky Jaroslav

Department of Information and Communication systems,
Institute of Automatic Control and Industrial Informatics,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology

Ilkovicova 3, 812 19 Bratislava, Slovak republic
peter.fodrek@stuba.sk

Abstract

This paper describes educational stand for teaching distributed control systems. It uses industrial real-time Ethernet framework called EtherCAT. For I/O we used modules from German company Beckhoff. EtherCAT master kernel module is used for Etherlab in Linux with RTAI. We mention Hardware configuration, system software and application architecture as well as some troubles in the solution. This paper is based on master thesis of Jaroslav Kamensky from June 2009 supervised by Peter Fodrek.

1 Introduction

Paper is about distributed control over EtherCAT. This technology is replacement of the fieldbus based on Ethernet. It solves main problem of the fieldbuses that is its cost because of low count of sold components compared to the standard hardware as Ethernet. Problem of Ethernet is that it was not designed for low latency and hard realtime but for soft realtime of a large amount of data. EtherCAT or Ethernet POWERLINK solves this problem. Then there is section about Ethercat I/O modules from German company and EtherCAT inventor Beckhoff. We used module from it as they were taken to us as a gift. Software contains three modules: 1st for communication over EtherCAT, 2nd for control and 3rd a only user space module for GUI in JAVA. We use analog Computer to simulate analog process in task of prove functionality. We used discrete PID to control non-linear system but there is possibility to use any other control law.

2 Ethernet

As we mentioned using Ethernet as the control loop fieldbus replacement we saw three main disadvantages:

- I. Too much ISO/OSI levels and therefore too much metadata for device communication with communication with low amount of data.
- II. Too much wiring and therefore too addition cost for low number of devices and CPU performance.
- III. Not enough realtime capabilities because of too much CPU utilization.
- IV. Star topology is not suitable for industrial control in many cases.

Because of this there was developed addition and changes to Ethernet to use it as fieldbus in distributed control. Most common are Ethernet PowerLink and EtherCAT. In the paper there we will be

mention only EtherCAT.

3 EtherCAT

Best addition is to add linear structure of the cables as show on the Figure 1

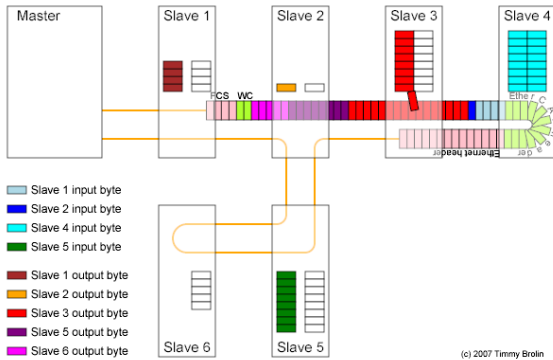


FIGURE 1: *Ethercat Priciple*[2][3]

Frame of the Ethercat based on Ethernet is shown in the Figure 2

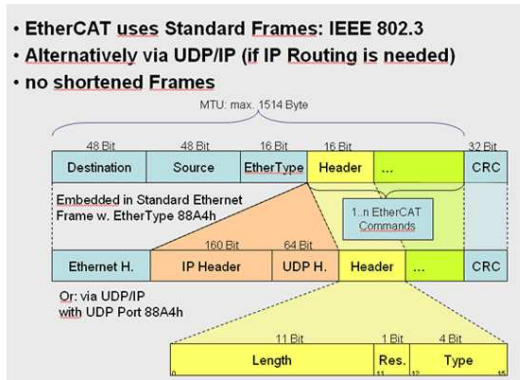


FIGURE 2: *Ethercat Frame*[2][3]

EtherCAT used on the fly read/write to the Ethernet Frame which is produced by EtherCAT master. EtherCAT master is software and therefore standard Ethernet adapter to be used. Only devices must be EtherCAT devices. This approach made author of EtherCAT German company Beckhoff famous. Each device can have several bytes on several places in the EtherCAT frame as shown in the Figure 3.

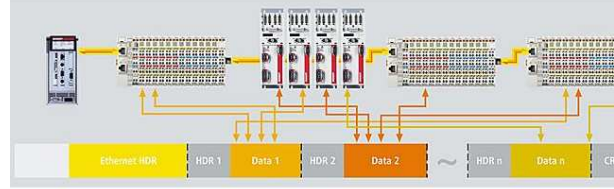


FIGURE 3: *Ethercat Device data in the EtherCAT Frame*[2][3]

Another problem is how to date after all modifications will return to the master for processing in the PC and GUI. Each system has virtual ringbus architecture with possibility to use hardware ringbus when device can be send in which of go-return part of the frame processing will process which bit of the frame as shown on the Figure 4.

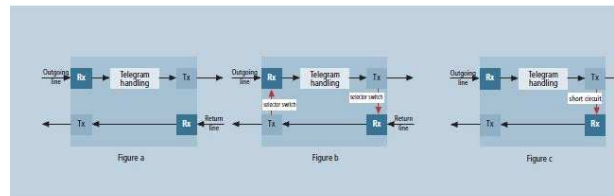


FIGURE 4: *EthreCAT ringbus*[2][3]

These changes are enough to achieve hard real-time capability with not non-realtime device on the net. There is possibility to use non-realtime hardware, but then there is need a special EtherCAT master to avoid frame collision with not realtime communication. Hard realtime hardware module is for example configurable module from Beckhoff as seen

on the figure 6

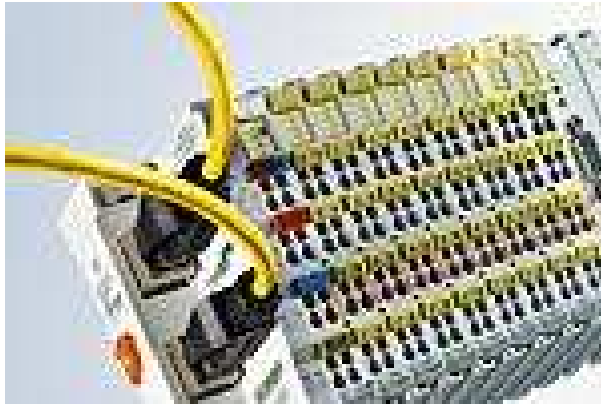


FIGURE 5: *Beckhoff device we used*

For our application only three modules are used communication module on the Figure 6, Analog input module for E-bus on the Figure 7 and Analog output module on the Figure 8. all these modules are from Beckhoff. We were connected to the communication module with much more modules on the e-bus but we use these modules

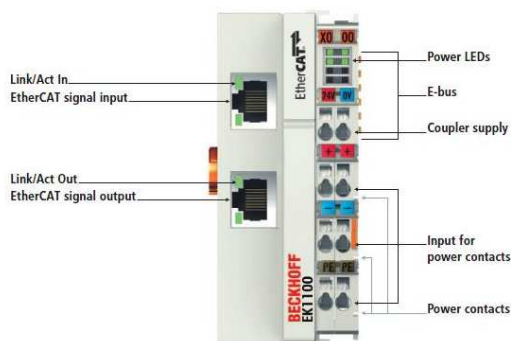


FIGURE 6: *Ethercat block*

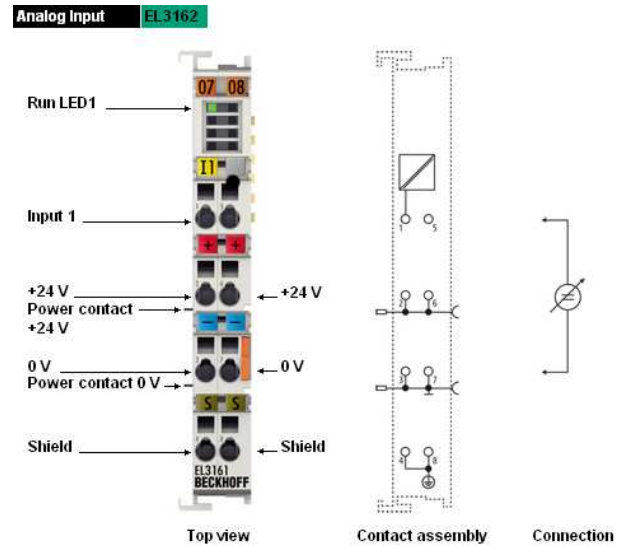


FIGURE 7: *Ethercat block*

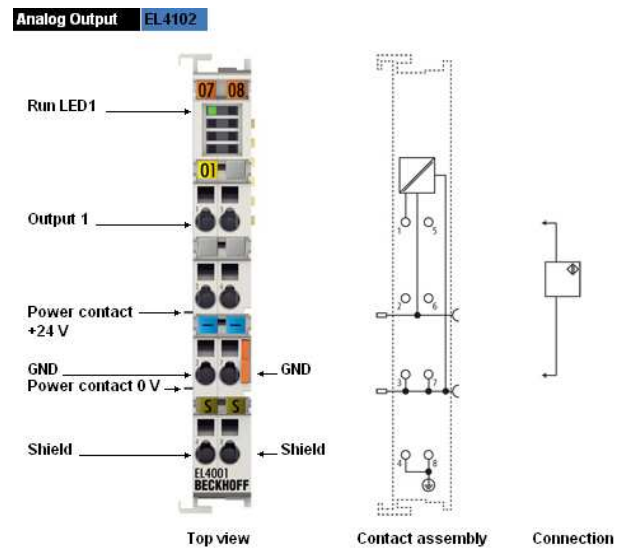


FIGURE 8: *Ethercat block*

4 Analog Computer

We have connected analog output and input to the analog computer MEDA-50 (MEDA=Maly Elek-tronicky Diferencialny Analyzator or Small Elec-tronic Differential Analyzer) as shown on the Figure



FIGURE 9: Analog Computer Meda-50

As we shown TV is not requirer but can be used in some cases to monitor proces but we do not used it.

5 Applictaion architecture

We used RTAI as Realtime nanokernel, EtherCAT master program form project Etherlab with Realtek Ethernet adapter. Therefor only several versions of the Ethterlab will work on specific kernel version with it. We use one mole to communicate with two kernel modules of the Etherlab master. One of it if for Ethernet adapter and one for API generalization. Our module set Etherlab master make communication domains and channels for specific and very well documented hardware form Beckhoff. Next module is used to count controller output or control law. it communicated with user space JAVA GUI application via Realtime FIFO pipes. GUI is able to change type of discrete PID with windup effect of controller output limiter and with anti-windup correction. It also use two types of discrete PID for both cases. Each of this four controllers can have own parameters of the gains and period of the timer of the control loop or discrete system period.

6 Application

We produced application in the Slovak language to manage control loop and for displaying values as of Figure 9. Values are like in control theory: w - for wish value, e - for deviation of the wish value and process output y for process output and u - for controller output. There are two controller outputs u_{poc} for algorithm output and u_{real} for real input of the system with mentioning bounds. Really only u_{real} is send to the Output modules because it does not accept larger value then maximum output. As seen on the Figure 10. there are 4 controllers and parameters are changes only when Zapis=write for controller is pressed Choice of the active controller is applied automatically. You can choose which of the values to display in the graph. As well as the wish value. It is applied when nastv=set button is pressed. Controller can be enabled(zapnuty) or disabled(vypnuty) and as well as integral part to be reset to the 0.. Graph can be static (staticky) or dynamic (dynamicky). static graph means that data are refreshed after data for new full screen graph arrived. Dynamic plots each time value arrive to the GUI application. For controller you can set gain for (P)roportional, (I)ntegral and (D)ifferential part a well as time period T .

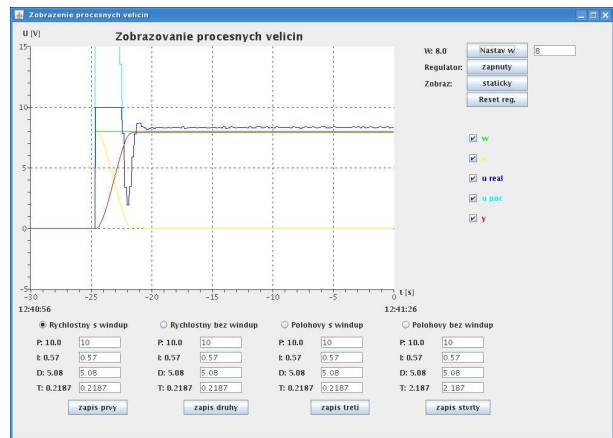


FIGURE 10: Application

7 Conclusions

We described our application used to demonstrate possibility to control in the industry via RTI and EtherCAT. It is not perfect and needs future improvements but student was learned ho to write kernel module RTAI application and JAVA GUI. As well as Java Native interface to use RTAI FIFO pipes for part of application to communicate.

References

- [2] <http://en.wikipedia.org/wiki/Ethercat>
- [3] <http://www.ethercat.org/en/technology.html>
- [1] <http://www.etherlab.org/en/ethercat/index.php>
- [4] <http://www.rta.i.org>

This paper is supported by grants from APVV and VEGA which solution finished in 2008 or first half of 2009.

RTAI in CAN bus based Network Control of CNC for plasma cutting

Fodrek Peter

Foltin Martin

Department of Information and Communication systems,
Institute of Automatic Control and Industrial Informatics,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology

Ilkovicova 3, 812 19 Bratislava, Slovak republic
{peter.fodrek ; martin.foltin}@stuba.sk

Blaho Michal

Farkas Ludovit

Poliacik Michal

Department of systems and Signals,
Institute of Automatic Control and Industrial Informatics,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology

Ilkovicova 3, 812 19 Bratislava, Slovak republic
{michal.blaho ; ludovit.farkas ; michlal.poliacik}@stuba.sk

Murgas Tomas

Systemy priemyselnej informatiky, s.r.o.
Kopcianska 14, 851 01 Bratislava, Slovak republic
tomas.murgas@syprin.sk

Chren Vladimir

Prva zvaracka, a.s.
Kopcianska 14, 851 01 Bratislava, Slovak republic
vladimir.chren@pzvar.sk

Abstract

Paper deals of RTAI usage in cutting machine of metal materials with plasma beam controlled via CANOpen protocol on CAN BUS interface. System uses JAVA/JNI for GUI design and communication with kernel modules dealing with RTAI hard realtime capabilities. The machine prototype is well running at Prva Zvaracka, a.s. Paper describes main parts of the system from control and software point of view.

1 Introduction

We have been developed plasma cutting machine control system for a commercial engineering subject. Paper describes main principles of this project especially of the software design. There is main problem to break engineers to permanently change specifications which degrades original lasagna-ravioli code design in direction to produce only spaghetti code. System contains mechanical portal with one motor on the X-axis but two pinion-rack mechanism with torsion shaft. Y- and Z-axis have also one motor each. Movement of each axis is provided by Permanent magnet synchronous motor with corresponding drive with CAN-Open interface, There are also binary and analogue input/output modules with CAN interface for technology. Another parts are two CAN-Open operator panel with 3 line display and industrial computer with additional two port CAN interface.

2 Hardware configuration

In this section we describe used hardware and the lowest level of the software which is in lasagna code equal to hardware drivers

2.1 CANbus interface in PC

In the prototype we used Adlink PCI-7841 because of cost and well documented Philips SJA1000 controller. We were mentioned that SJA1000 is capable to achieve bandwidth of 1Mb/s, but on this card it is impossible. We were used this card because of previous RTLinuxFree based project and own driver derived from CanFestival project's code for hardware which is not yet included in the code. In that code there was an remark that driver does not work on some revisions of Adlink PCI-7841. Code does not worked with our card, so we made some extraction and customization of the driver's code as it is not driver but only module with exported symbols which are function names. When we used only BasicCAN functions then driver worked, but when used another features of the SJA1000, system crashed. BasicCAN is mode of compatibility with Philips PCA82C200 which is pin and electrically compatible predecessor of the SJA1000. There is possibility to use SJA1000 in the PCA82C200 board but then SJA1000 can act only as PCA82C200 or only in BasicCAN mode of SJA1000. About half of the cards with PCA82C200 supports only 500Kbit bandwidth although PCA82C200 chips has 1Mbit capability. During research we tested German Peak system

PCAN-PCI dual channel SJA1000 based card. And there difference. It works on RTAI with RTDM driver on the 1Mb/s well. There was RTDM driver for that card from the manufacturer. It worked extremely well except that there are no examples for kernel mode use of the driver and examples were not able to compile when used RT=RTAI option of the make command which causes to compile driver for RTAI. There were misspelled prefix of `t_` for transmit instead of correct `r_` for receive in receive function or vice versa. When corrected all examples compiled correctly with even with RT=RTAI option. Just now there is newer version of that driver available on the Peak system website. We have to test it, yet.

Because of listed we have found that there is reason to look at the boards and find differences. Although PCB for PCAN is much smaller than that for PCI-7841, number of ICs is higher for PCAN. All ICs on the PCI-7841 are present on the PCAN but there are additional ICs on the PCAN. As of this we found that our revision of Adlink card is board for PCA82C200 which is not in production for a long time and it is replace by SJA1000 as card is sold incorrectly as SJA1000 card.

Therefor we will use PCAN card in the future version as well as we are to test PCAN-USB opto-decoupled PCAN interface for USB in the future. We used this USB dongle with Peak system Windows only software to monitor CANbus during development for system's unpredictable behavior debugging before testing PCAN-PCI. But because of incompatible interface with current driver it is needed to rewrite lowest level of the application. Here is shown why there is needed lasagna code strategy of development instead of pure ravioli code which is best code strategy for code maintenance.

2.2 Motors and drives

We use 3 motors from German vendor Lenze with drives with CANopen interface from same vendor. There are drives with or without PLC. We used version without PLC because we do not need this function when limit switches are processed as well in non PLC version. Drive also handles over-current, over-voltage over-heating and maximum velocity or position error exceed. We can transmit wished position and velocity to the drive and read current values of velocity position and limit switches statuses with period of 2nd milliseconds instead of planned 1 millisecond with. This achieve wished accuracy of 0.1 mm with speed of 50mm/s when normal speed is 30 mm/s and maximum is 80 mm/s. Accuracy is not dependent on the size of arc as in previous using solutions. If in the previous solution circle has diameter 1 m

then accuracy in previous DOS based solution has dynamic accuracy of 15.7 mm when diameter of and circle is 1 cm accuracy of 0.157 mm. If arc has 90 degrees then accuracy 3.93 mm for arc with diameter 1m and with 1cm is it 0.0393mm for speed of 157.08 mm/s but accuracy of 0,12 with velocity 50 mm/s and 0.21 mm with velocity of 30 mm/s. theses velocities.

Our new solution has dynamic accuracy of 0.32 mm at 160 mm/s; 0.16 mm at 80 mm/s; 0.1 mm at 50 mm/s, 0.06 mm at 30 mm independently of the diameter of the arc. Only parameter new solution depend on is velocity because time interval is fixed. Accuracy is counted as time interval of 1 ms times velocity. Opposite is old solution where two factors limits accuracy. First of them is maximum count of interpolation points of the arc which is 200. Second is minimum time period between points of 1ms.

Therefor new solution has canceled one of two limits of the old solution but because of badly chosen PC interface for CAN and drives with minimum refresh wish value 1-2ms depending on used CANbus bandwidth the second limitation has been worsen. This can be solved by replace CAN interface from Adlink to Peak system card. Only digital drives that are capable to refresh wish value in period lower than 1 millisecond are drives of product line Sinumerik from german vendor Siemens. Sinumerik is only drive product line that is specialized for high precision high speed CNC and therefor their cost is at least two times higher than drives from Lenze, ABB, Rockwell Automation/Allen-Bradley, Control Technique, Rexroth or Invertek which are only companies in second league in drives behind only first league Siemens.

2.3 Input/Output modules

On the CANbus there are also several binary inputs output for example for toxic gas exhaust section select and start/stop of the exhauster. Machine is divided into exhaust section when only one section is enabled because then you do not need to use that high power exhauster as needed by just one section. Using several sections then make machine more green as it consumes lower electricity. Exhauster is needed because cutting metal material by plasma is based on the sublimation of the metal. Produced metal gas reacts with components of the air with product of toxic gases. Exhaust has also cooling effect which is needed to achieve minimal cutting gap. Another binary I/O are used to start/stop plasma and check plasma generator's state because there are part of rapid move between cutting trajectories. Analog output is used to set power of plasma which is pa-

rameter depending on thickness of cutter material. Analog input is used to measure deviation voltage of electric arc of plasma which indirectly measures deviation of hight of cutting head from the material which must be zeroed for best cutting process and minimal cutting gap. There are also binary inputs from operator's panels buttons and binary outputs to change color of the button, to inform operator which buttons are allows and which not. It is possibility not to under-light button with any color.

2.4 CAN bus

All control system components are connected by field-bus from German company Bosch named Controller Area Network which is primarily used in automotive industry. We chose this fieldbus because in 2007 we were world's number one car maker per capita, when car production in Slovakia reached a record 571,071 that means the average of 105.7 cars manufactured for every thousand inhabitants[1]. Therefor knowledge of the CAN bus adds capability to achieve job after crisis. After crisis is it because our top car bodywork designer Jozef Kabana chief-designed Bugatti Veyron, and was a chief bodywork designer of Audi in time they made latest facelifts of Audi A4/A6/A8. Bratislava's Volkswagen produces Audi Q7, Porsche Cayene and VW Tuareg. Our project made major success in area of CAN bus kernel framework as listed in software part. Volkswagen team has been beaten by us in this part of kernel but there are different approaches by doing this.

2.5 Industrial PC

For the GUI and control we are using industrial panel PC with resistive touchscreen without mouse or keyboard. PC is in moveable rack, which could be moved across the machine. the only cables from an to rack are power cable a two four wire CAN bus cables. industrial PC has only two slots for additional cards one PCI and second PCI Express x1. Now we use PCI, for PCI-7841, peak system card is as well for PCI Express x1 as for PCI, so we can have four CAN ports. Processor is Pentium IV based Celeron on 3Ghz. There are two Ethernet adapters one for 1 Gbit/s and one for 100 Mbit/s for use to remote monitoring via Internet or by executives of the company, where machine will be working after sold. As well for planned automatic software updates. Graphic adapter is integrated on the board from Silicon Integrated Systems. for the future we plan use integrated GMA3000/3500 or Radeon HD4200 for much faster graphics and OpenCL usage.

3 Software

In this section we will discuss software. It has three modules working on different computers. First is design of what to cut by engineers in their own CAD system. This leaves unchanged. Second is system to transform CAD image into NC or CNC program into text file with commands in language that machine understands. This was upgraded from DOS application to another multi-platform solution. Third part is control system running at the industrial PC. This basic architecture is as show ravioli architecture of the code because each part of system is independent and communication is strictly defined by DXF(Drawing Exchange Format) format between designing CAD and transforming CAD/CAM software to the own NC program language REZ. NC program is by Prva Zvaracka called Cutting plan but any other firm call it NC program. Machine must be able to read text file with REZ language command and interpret them. It is not ravioli code because independent entities are not as small as it can be. But there is not design of code but design of architecture.

3.1 CAD

CAD could be chosen by customer only condition is that he export drawing into DXF which is format designed to exchange CAD drawings between incompatible CADs. Therefor customer can use amateur's CAD like QCAD, half-professional CADs like Autodesk AutoCAD, which is only available for windows and is most used in Slovakia or as well professional CADs like Bentley Microstation, IBM/Dassault Systemes CATIA or Parametric Technology Corporation Pro/Engineer. All of the professional CAD are able or was able in previous versions to run on Linux or other Unixes as well as majority of amateur CADs. Half-professional CADs are rare for GNU/Linux but there are several noticeable as Weber Systems Synergy, VariCAD or U.S. Army Ballistic Research Laboratory BRL-CAD. There is open source project openCASCADE which produces platform to build multi-platform CAD especially CADs for GNU/Linux. It is main project of Google summer of code of previous and this year with potential to be finished in 2010 as framework that has all function of AutoCAD except DWG file reading. This is because openDWG library from Open Design Alliance[2] which supports DWG is paid, but last year there was a change that Adobe opened specification of the DWG via Open Design Alliance for free and there are several CADs for linux that supports Au-

todesk's AutoCAD native format DWG but it is still no perfect. Project to build free Linux library similar to openDWG, which is free for anyone for Microsoft Windows but paid for Linux is another key project of Google Summer of code 2009. openDWG is technical free Linux, but Linux version is only available for members of Open Design Alliance but membership has obligated yearly fee. To download Windows version of openDWG membership of Open Design Alliance is not required, but to download any Unix version of openDWG is membership required.

3.2 CAM - conversion to the REZ

in this chapter we will discuss three offered solutions of converting DXF to NC program

3.2.1 DCPC

Old version of the software is Slovak language based Design CAD for PC. It has two parts Imsisoft Design CAD for Drawing and Basic script which produces NC programs from the drawings named DCPC (Design CAD for Processing Converter). DCPC work only on DOS and only of specified version of the DesignCAD and only with its Czech localization.

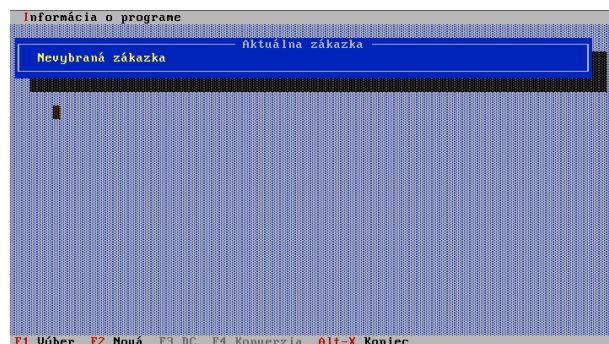


FIGURE 1: *DesignCAD for PC main screen*

As seen of the Figure 1 GUI of this software is based on DesignCAD and looks like traditional Borland's IDE for Pascal or C language. It works on DOS or MS Windows 98 or older but not with NT kernel. For newer Windows it does not work. So there is time to replace it, but some engineers want not to change known GUI. For them there is solution with Sun VirtualBox and virtual FDD for mounting *.img as Windows disk which enable transfer data to and from the DOS machine. Main and only problem is with CODEPAGE for localization. FreeDOS does not support national codepages (national character sets and number formats) at least via needed

country.sys file in stable FreeDOS release. In development releases country.sys is present. But there are free to download supplemental files for MS-DOS 6.0 including country.sys at Microsoft website[3]. There is a problem that FreeDOS is fully compatible with DOS 6.22 not 6.0 and for DOS 6.22 Microsoft does not offer country.sys. Therefore we need to use MS-DOS to run this program with correct display of the national characters. There is not technical barrier to use MS-DOS in VirtualBOX but this is only legal problem. There is only solution via Microsoft's license legal guide to downgrade rights[4] there is written: "Downgrade rights grant the right to use prior versions of Microsoft software. Microsoft Volume Licensing programs include specific downgrade rights. The OEM License Terms for most OEM versions of systems software do not grant downgrade rights. The exception is the OEM License Terms for the WindowsXP Professional operating system and the Windows Vista Business and Windows Vista Ultimate operating systems, which grant downgrade rights." This not mention replacement of the newer system when used older as usually mentioned. This is only Microsoft's that do not mention Windows for downgrade, so if we have Windows License we can use DOS in Virtual BOX legally, when do not use other Windows application simultaneously. You just to have legal copy of the DOS installation disks or copies from subject that owns it legally. This is true for us. Prva zvraracka has Windows licenses as well one license for MS-DOS licenses from the past. They also has valid DCPC and DesignCAD multi-license, but only one computer capable to run MS-DOS.

3.2.2 QCAD/CAM Expert

Ribbonsoft QCAD is GNU GPL licensed multi-platform 2D CAD. CAM expert is proprietary addition to QCAD for transforming DXF to standard or user defined NC program language. GUI is shown on the Figure 2. Software is fully localized due to usage of QCAD code. Q in the name means usage of Trolltech's multi-platform Library Qt. After Nokia has bought Trolltech the name was changed to Qt software. CAM Expert costs 149.00 EUR per single seat as shown on the Zurich, Switzerland located Ribbonsoft home page[5]. CAM Expert is tested on several OS's Windows: 98, ME, XP, 2000, Mac OS X: Mac OS X Tiger (PPC, 10.4), Panther (PPC, 10.3) and from Linux: SuSE 8.0, 8.1, 8.2, 9.0, 9.1, 9.2, 10.0; Redhat 7.2, 8.0, 9.0; Fedora 3; Knoppix 3.3, 3.4, 3.8; Xandros 2; Mandrake 9.0, 9.2, 10.0, 10.1; Linspire 4.5, 5.0; Slackware 8.0; Puppy 1.0.5; UHU-Linux 1.2 and Turbolinux 7.0, 8.0. We have tested it on Slackware 9.0 and 12.0, Mandrake 9.0, Open-

SUSE 11.0 and 11.1 and MacOS X 10.5.7 and 10.5.8 (Leopard) on Intel based Mac. We tested MAC OS 10.4 version on that MAC.

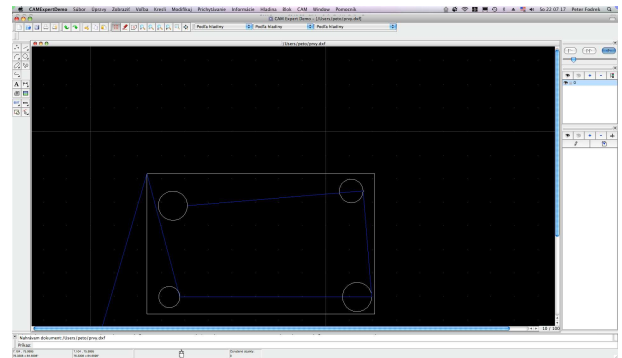


FIGURE 2: *CAM Expert GUI*

In the CAMExpert are located all menus to selected language except CAM with CAM Export and Reorder entries items. First of the is used to export drawing to the NC program with selected language. Simple language definition files are in the machines directory of the directory where CAM Expert is installed. Files are named language.cxm where cxm means Cam eXpert Machine. You can select several languages included G-code which is ISO 6983 norm currently redefining by drafts and approved parts of ISO 14649 series. Part 14 named ISO 14649-14 for laser and EDM e.g. plasma beam technologies process control commands are not yet approved. Preparing stage of ISO 14649-14 was finished this year and on June the 25th, 2009 was started approval process which belong at least half a year. ISO 14649 draft is not currently supported by CAM Expert, but in Q4 2004 there will be new generation of CAM Expert introduced to the market or you can add it via creating own cxm file.

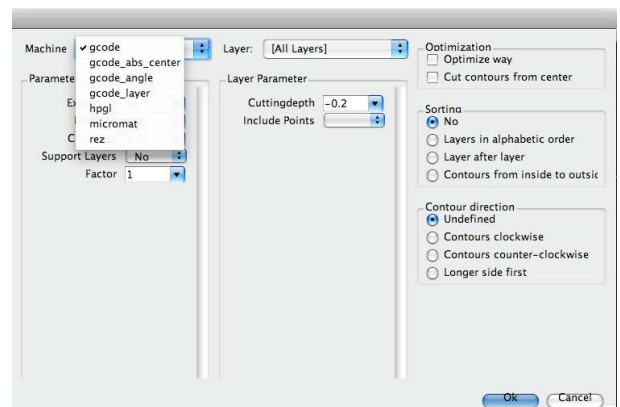


FIGURE 3: *CAM Expert CAM Window*

CAM dialog is shown on the Figure 3 where language choice is upon Extension, Feed rate and Clear-

ance parameters. Main advantage of CAM Expert is its capability to have user interface in 17 languages including English, German, French, Italian, Czech, Slovak, Hungarian, Polish and Russian but it is far less than 42 languages in QCAD. Where untraditional languages like Bask, Catalan / Valencian, Breton or Irish is possible to set. Other advantages are that program run on many OS-es including Solaris, BSD and listed before OSs as well as simplicity to add own NC language. Main disadvantages are that CAM menu is only in English as well as that it is not capable at current version to automatically order arcs and lines in other order then order the arc and line they were drawn or saved by another CAD. You must reorder arcs and lines manually which means to click on the each part of the image in order you wish to get as a result. Only exception is that next parts of image are in right order then you must not click on that parts. Another disadvantage is that you must add arcs or lines for coming the trajectory a leaving out of trajectory before and after trajectory is cut. These additional parts are usually used for accelerating and decelerating from the cutting speed as they are must for plasma and other EDM technologies but not needed for milling or turning. CAM Expert can do simulation of the cutting to determine order of cutting. You can also choose speed of cutting, but when you use play button then you only see order of the lines and arc and only when lowest speed is chosen, because system is so fast. When you use fast forward button instead of play button you will take point by point simulation which is slower than part simulation via play button.

3.2.3 HeeksCAD/HeeksCNC

HeeksCAD[6] is multi-platform 2.5D CAD at range of low end half-professional CAD that is based on OpenCascade library covered by new BSD license. Technically it is an GUI for openCASCAD, made partially in C and partially in python. HeeksCNC[7] is python library written in C for HeeksCNC which act as an extension of the capabilities of the HeeksCAD. Extension adds one menu bar item and one toolbar to the heeksCAD GUI you have to go to the File menu and choose plugin and add shared library (shared object) and then you need to restart HeeksCAD to start installed plugin. For Microsoft Windows there is not separate installation of HeeksCNC because if you install HeeksCNC you will install HeeksCAD with integrated HeeksCNC plugin. Both HeeksCAD and HeeksCNC are projects maintained by Dan Heeks and it is one of the projects under Google summer of Code 2009. Main legal advantage of the BSD license is that you can take all my work and use it

for your own commercial application. That means we can close modified source code for a customer. Main advantage is that HeeksCNC adds automatically accelerating and decelerating additional part of the trajectory, but only as arcs. This is no problem although DCPC uses arc or lines as such parts, but arcs are more suitable for the mechanics of the machine. Arcs has only 90 degrees so it is longer than similar linear item of the path. This slows cutting a bit compared to linear acceleration deceleration item. You can also choose a point of middle of the arc for starting and as well as finishing trajectory or more precisely point near it. Program has several modes of the transfer as shown on the Figure 4 on the 0.7.x series of the HeeksCNC.

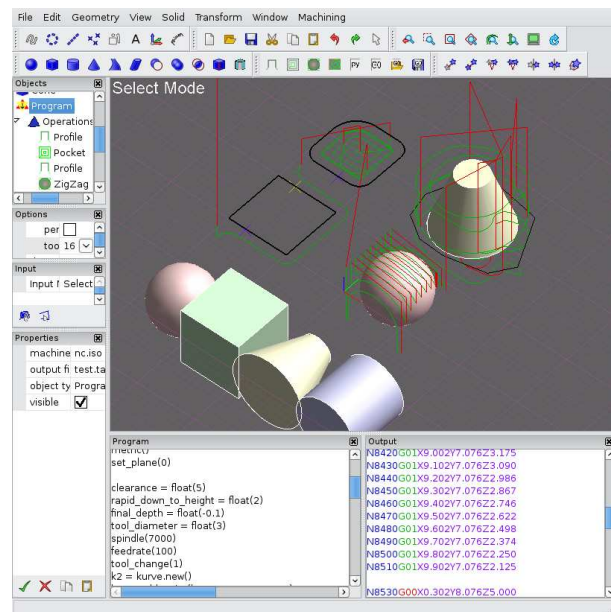


FIGURE 4: Heeks CNC 0.7.x

Current version is 0.8.1 for Linux and source code version and 0.8.2 for Windows. HeeksCNC menu bar item is Machining and toolbar for HeeksCNC starts under scissors and finished with diskette with "GO" label. First four buttons of this toolbar sets profile mode (in HeeksCAD called operation) for current trajectory or 3D object here known as sketch. For sketch we can set clockwise or counterclockwise direction as main of the parameters. For profile created from the sketch we can set correction for tool dimension like left or right, which means on which side of the trajectory when cutting selected direction is correction applied as well as diameter of the tool. There is applied correction as half of the tool diameter which is logic because we control position of the middle of the tool. As of 0.8.x series there is possibility to choose 0 mm diameter. For the profile we can select position

of the start and stop of cutting as shown on the Figure 4 with two crosses near the square. Yellow is for start and blue is for the end. When you choose the at the same point only finish is shown as also visible on the Figure 4 on the object right of the square. "Py" labeled button generates python script from the drawing and "Go" labeled button executes it. From the 0.8.x version there is no "Py" button but "Go" button acts as sequence of "Py" and "Go" buttons. Function of "Py" button is available from the Machining menu in the 0.7.x as well as in 0.8.x series. Script is shown in the Program sub-window. Script contains several import directives but one is dynamic and generated dependent of the the property named machine of the Program object as seen on figure 4. In the 0.7.x there is relative path to the library file containing functions needed by python script to generate code in proper language. In the python there is / in the path replaced by "." and final .py or .pyc in the name of the library is omitted. So nc.iso means path ./nc/iso.py or ./nc/iso.pyc where .py is used for source code of the library in the PYTHON language and .pyc means PYTHON compiled. compilation of the .py to .pyc is done by first call of the .py file or using import directive which includes library. when nothing changes to the .py file then .pyc is not regenerated and pyc is used. Therefore we can distribute binary python library without source code for generating library of the REZ language. 0.8.x series adds to the nc directory text file index.txt where is written for iso following "iso G-CODE iso6983". This causes that user does not write name of the library file but chooses from the list possible values in the machine parameter string "G-CODE iso6983" for HeeksCNC to add nc.iso the import directive. After run script correctly there is generated file /tmp/test.tap for Unixes and C:\Windows\TEMP\test.tap or similar for Windows. In that file there is generated code but not for user. After this HeeksCNC calls another file that has same name as generating script but with additional suffix _read so if NC code is generated via rez.pyc then there is to call rez_read.py or rez_read.pyc. This file reads /tmp/test.tap file which is sent to the script as parameter and could be changes via property of the HeeksCNC Program object for both generating and reading script. Read script must read line by line a parse output file. In includes library with five main functions add_text that is suitable to add process commands. Similar is add_comment which adds text but this text is shown in the output window with blue color instead of black for add_text. Because REZ language differs remarks and process commands we used both. Next function is add_rapid which adds rapid move between Profiles, this can be only linear move and command

and path of the machine in the display window and output windows is red for part added by add_rapid. Function add_feed adds linear cutting move which is displayed in both output is green. Both add_rapid and add_feed has parameters of final point in 3D in mm. Last command is add_arc which adds arc to the trajectory. Arc is in graphical window green but arc command is cyan in the output list. Arc has different parameters for HeeksCNC and REZ language. REZ language has 3 parameters X,Y coordinates of the arc center and signed angle of the arch, where sign means direction but HeeksCNC had 6 parameters and the direction. First 3 parameters of heeksCNC is 3D vector from current position of start arc to the centre of the arc and second three parameters are coordinates of the final point of the arc. It exists simple conversion from parameters in REZ to the parameters of HeeksCNC by vice versa conversion is much more complicated. The read script produces /tmp/test.tap.nc.xml file which is pure XML a this file imports HeeksCNC after script finished. This file is used to display colored real trajectory of the Cutting head center. Open folder icon baton reads NC command file and displays it via read script. Diskette saves displayed NC program to the specified file not in the working directory.

Main disadvantages are that building new languages via two Python scripts is much more difficult then language definition in the CAM Expert, that system is in beta testing state and that windows version crashes when saving to the Heeks own format where all CNC settings is saved. Saving into DXF or using Linux version has no problem. Last problem is that there is no simulation like in CAM Expert but when you click on the lines of code in the output windows then corresponding arc or line will be bold in the image window. This can replace simulation but it is less practical. Sometimes drawing is not refreshed correctly so you need to change size of program windows two time to see all image. CAM Expert and HeeksCNC has same bug that when you save file. File Save dialog displays only *.nc and *.cnc files but saves into chosen endings of the name(in Windows Extension). HeeksCNC is better because when you choose *.rez file and same file exists HeeksCNC ask if you allow replace file but CAMExpert do not ask and rewrite file without notice. It is surprise because CAMExpert's language definition file has parameter default extension and HeeksCNC has no such parameter. Surprise doubled when you find that CAMExpert saves file with default extension instead of displayed extension. Therefore is HeeksCNC best solution for the future although author of new language can not find if is presently cutting on the left or right part of the trajectory. this function will be helpful

for us but engineers from Prva zvaracska uses only one side and they determine if contour is in or out via clockwise or counter-clockwise direction just now. It runs in Windows and openSUSE 11.1 form rpms in version 0.7.5 and from compiled source of version 0.8.1. on the MAC OS X 10.5.7 and 10.5.8 there is problem with incompatible vxGTK and wxWidgets libraries C language SDK a therefor we can not prove if it works.

3.3 Control system on the machine

In this section we mentioned how is machine software designed. At the top level there is ravioli design. There are three objects: Control application, Reading REZ format and transform it to set of Position and velocity wish values in time and the last one is Module of standard shapes.

3.4 Rez interpretation

Rez format interpretation is C language program that takes parameters as filename of the REZ language program, cutting velocity repeating and spaces between repeatings of the file in each axis. When cutting multiple plans and correction for size of plasma beam which is same as in HeeksCNC but without left or right direction which is written to the REZ file. Next parameters how long before start cutting is cooling gas open so gas comes in the place where will be cut started and how long after shutting down plasma will be gas opened. Last parameter is how long will be plasma beam on before start moving to crosscut thick material. Inputs are well defined and output is file with copy of memory of defined structures of values. Content of this structure has been changed during development and we produced this part based on previous two solutions but done by Mr. Blaho who does none of previous solutions. Rewriting code and redesigning structure made much faster code and better maintainable code but there was needed to add new feature of correction and this code needs to know line that will be interpreted by program in the future as well as current a previous non comment line of the REZ file. This was possible done only by multiple pointers because of bad definition of the REZ language back in 1990's and 1990's. Using REZ language is must because of old system working machines at the Prva zvaracska customer fabrics plants and their need to upgrade to the compatible system. Three different pointers and rewriting working code with new feature has shifted code bit toward to the spaghetti code but no more then it was needed as we have not enough time to

write completely new code with better architecture for adding new features. Before starting adding new features code was based on the lasagna code principals.

3.5 Module of standard shapes

This module was need by Prva zvaracska to allow worker test machine by cutting predefined shapes with parameterization of any aspect of the shape. It is made in Java and it allow user to choose shape, then to write parameters into it, view how parameterized shape looks like with correction of hazardous or not allowed combination of the parameters and then generates program in REA language in predefined directory with predefined name. GUI JAVA program calls this full screen application when chosen to open it and after program finished GUI must start Rez interpretation module. Main problem is that industrial machine has no keyboard nor mouse. It has only touch screen and user that is unable to use general purpose virtual keyboard like xvkbd. There was designed own virtualkeyboards that is able to use worker. Code is in java with strict ravioli internal structure of code.

3.6 Control application

Control application is designed upon lasagna code principles which is must because one part is in kernel space and another in user space. Each of this is different ply (of lasagna) and there is strictly defined which ply is lower level and thus has priority of execution because kernel space part is RTAI nanokernel application which run on same ply of system architecture as kernel. As we have found it run in Ring 0 of the processor. Realtime part is written in C and GUI user space part is written in JAVA using JNI(JAVA native interface) to access RT-FIFO pipes files in /dev. These pipes acts as communication channels between parts of application

3.6.1 Hard Realtime part

This part is designed originally as lasagna code with several layers first is communication with CAN card, then there is an layer of the CANOpen protocol, then there is layer of the devices which is designed as ravioli code where objects (ravioli) are communication with Lenze drives over CANOpen protocol, then there are communication objects for digital inputs, digital outputs, analog inputs and analog output. All these modules reads and writes values to the mem-

ory area of the process values. Next layer is layer of the control where are position controllers and mode of operation tasks as RTAI threads as well as communication via five realtime FIFOs. This layer is too complex and allow to produce spaghetti code directly from design because control depends on communication and thereof GUI could produce problems in the realtime parts. Very special problem is that GUI could ask to change mode of operation of realtime code. There is basic control which does not allow to accept request in any time, but we can not prove that it is enough. This is worst part of the realtime application. Best part is part of CANopen protocol. This part was developed at that level which allow one of the tasks Vladimir Chren's master thesis to be internationally accepted in CANFestival project[8] with "Wow" reaction in January/February 2008. it was the first working CANbus protocol in kernel space. CANFestival is CANopen framework originally from France and Vladimir was maintainer of the kernel port of the framework. But he was busy working on this project which not allow enough time for maintaining code last year. Vladimir overcame five years development of five developers in Volkswagen in Wolfsburg which developed similar CAN protocol kernel framework but VW superseded Vladimir in introduction to the kernel as VW code was introduced in 2.6.25 kernel in April 2008. VW code is still not working well as of 2.6.30 kernel but may be better in 2.6.31. To be correct VW's code used network stack instead of serial/tty stack which was used by Vladimir. As correct lasagna code each layer is written as individual kernel module and therefor modprobe for loading all is used in rc.local file.

3.6.2 GUI part

GUI is written in JAVA so it is ravioli code. As all code of the project it is written in Eclipse and by manual writing of SWT using code because of very bad Visual editor in the eclipse especially three years ago, when project started. The biggest problem is that JAVA does not allow to access /dev files and therefor there is needed C code compiled with JAVA Native interface Libraries and JNI classes for the code. Main problem was that C and JAVA use different ENDIANs so correction of the byte order is needed. JNI has endian correction functions. Application Loading before kdm or as only shell of automatically logged in user allows users to simply shutdown computer and developers are allowed to test new version within usage of old version.

3.6.3 RT FIFO

Biggest problem of control application is communication between parts. It is done by five RT-FIFO of the RTAI 3.6 on the Slackware 12.0 There is one unsolved problem here. We transfer data as structures put in to the one direction pipes. But when size of the fifo where data to display like velocity and positions are sent or wish values to the control module is not equal of integer multiply of the size of send data at once system crashes. As well as system may crash when GUI waits for confirmation of the state change from kernel space and realtime part waits for ask for state change. Both parts are tested individually via C application reading and sending data to the pipe as anticipated when everything works well but when they are tested together it sometimes fails. This is sign of the spaghetti code between layers. Spaghetti code is well known as badly written code where independent parts depends on each other. Spaghetti code is caused by principal misfires of the project/team leader as Peter Fodrek is for us. Team-leader's task is not to allow customer to take not full or not to correct specifications at the start and to change specification during programming. Failure was that specifications were not written and signed before project starts by customer. It was because friendship between company executives. Another task of the team-leader is to be available for programmers to help them with problems and check their codes and third party codes as well.

4 Conclusions

We have designed function working application with stability issue needed to check and repair. Working application will be presented via video on the conference. There is need to change team-leader or at least to force his to believe in programmer analysts which coded Control application as much as it was in this project. And check their work on communication part and not allow them to be forced by continuous specification change to produce spaghetti code.

References

- [1] http://www.industryweek.com/articles/slovak_car_industry_production_almost_doubled_in_2007_16083.aspx
- [2] <http://www.opendesign.com>
- [3] <http://support.microsoft.com/kb/124382>

- [4] <http://www.microsoft.com/downloads/details.aspx?FamilyID=4F4B3CFD-7F4E-46CB-8117-8275F7683D3F&displaylang=en>
- [5] http://www.ribbonsoft.com/store/index.php?main_page=product_info&cPath=69&products_id=193&zenid=37a23ee796e0a6edce7c6a933f3be043
- [6] <http://code.google.com/p/heekscad/>
- [7] <http://code.google.com/p/heekscnc/>
- [8] http://sourceforge.net/mailarchive/forum.php?thread_name=200801191554.49767.vlado.chren%40gmail.com&forum_name=canfestival-devel
- [9] http://kernelnewbies.org/Linux_2_6_25#head-0957d99bd917f82924f1eab1f3f17945e1c4dc90

This paper is supported by grants from APVV and VEGA which solution finished in 2008 or first half of 2009.

On the RTAI-Lab implementation of an observer-based controller for a DC motor

**R. Salas-Cabrera, J. C. Mayo-Maldonado, E. Y. Rendon-Fraga,
E. N. Salas-Cabrera, H. Cisneros-Villegas**

Instituto Tecnológico de Ciudad Madero

Departamento de Ingeniería Electrónica

Maestría en Ingeniería Eléctrica

Av. 1o. de Mayo S/N, Madero, Mexico

salascabrera@aol.com, jcarlos_mayo@hotmail.com, erendonfraga@hotmail.com,

nacu_salas@hotmail.com, hcvillegas@yahoo.com.mx

Abstract

This work deals with the experimental implementation of a state space controller for a DC motor that has separate winding excitation. The output to be controlled is the rotor position. Linux and RTAI-Lab are the open source tools that we use in this implementation. The control law is derived by using a model that consists in a set of ordinary differential equations. Additionally, in order to avoid several issues regarding the measurement of the state variables, we implement an experimental discrete-time state observer to estimate the transient values of the rotor speed, armature current and even the load torque. The real time program calculates the discrete-time state feedback including an integrator, a full order discrete-time state observer and a rotor position sensing algorithm. Custom-made digital, analog and power electronics designs are fundamental components of the hardware used in this closed loop implementation.

1 Introduction

This paper presents the implementation and the experimental results of an observer-based control law. RTAI was used in order to perform the experimental implementation of a state space controller for a DC motor. The field and armature windings of the DC machine were fed from independent sources, [1]. The output to be controlled is the rotor position.

An interesting contribution is presented in [2], the author uses RTAI-Lab, Scilab and a data acquisition hardware for implementing a transfer function-based controller for a DC motor. The main contribution in [2] is related to testing several open source real time tools (RTAI, Comedi, Scilab, xrtailab). Experimental results regarding high gain observers are presented in [3] where RTAI-Lab and Scicos/Scilab are used for the real time implementation. The system to be tested in [3] was a series DC motor. Valuable information about a problem associated with the analog outputs of the acquisition card was pre-

sented in [3].

Linux and RTAI-Lab are the open source real time platform that we employed for this implementation. In particular a version of knoppix 5.0 was used for this work, this version was developed by Gianluca Palli and it can be downloaded from his home page [4].

The real time program that we implemented consists of a discrete-time state feedback including an integrator, a full order discrete-time state observer and a rotor position sensing algorithm. In this control law implementation the only state variable that is measured is the rotor position of the motor. In our case, a set of five difference equations (fifth-order discrete time dynamical system) is computed every 0.2 milliseconds (5 kHz). Important components of the experimental system are: a personal computer, a National Instruments PCI-6024E data acquisition card [5], comedi driver library for Linux [6], a free open source real time platform [7], a custom-made

power electronics converter, an incremental encoder and signal conditioning circuits for measuring the rotor position.

2 DC Motor Model

The equations that establish the DC motor behavior are obtained by using fundamental electrical and mechanical laws [1], this is

$$V_f = r_f i_f + L_{ff} \frac{d}{dt} i_f \quad (1)$$

$$V_a = r_a i_a + L_{aa} \frac{d}{dt} i_a + L_{af} i_f \omega_r \quad (2)$$

$$J \frac{d}{dt} \omega_r + T_L = L_{af} i_f i_a \quad (3)$$

$$\frac{d}{dt} \theta_r = \omega_r \quad (4)$$

In order to improve the performance of the closed-loop system, a new linear state equation for the load torque is proposed, this is

$$\frac{d}{dt} T_L = K_0 \omega_r + K_1 T_L \quad (5)$$

Several transient and steady state tests were performed in order to obtain the nominal parameters of the DC motor. Since the field winding is fed with an independent constant voltage source, the equation (1) can be omitted and the field current is considered as a constant parameter. The notation of the variables and the calculated values of parameters of equations (2)-(5) are shown in the following table:

θ_r - Rotor Position
ω_r - Rotor Speed
i_a - Armature Current
i_f (0.46 Amps) - Field Current
V_f - Field Voltage
V_a - Armature Voltage
r_a (6.615 Ohms) - Armature Resistance
L_{aa} (0.0645 H) - Armature Inductance
L_{af} (1.7686 H) - Mutual Inductance
J (0.0038 kg/m ²) - Inertia
T_L - Load Torque
K_0 (0.20907) - Coefficient of Load Torque
K_1 (-9.8297) - Coefficient of Load Torque

A nominal dynamic model is determined by substituting the numerical parameters in the equations (2)-(5). Then, employing 5 kHz as a sample rate and following the procedure presented in [8], it is possible

to obtain the discrete time dynamic model. Also it is possible to obtain the discrete model by using the Scilab comand *dscr*.

The nominal linear time-invariant discrete time dynamic model obtained is

$$\begin{bmatrix} \theta_r(k+1) \\ \omega_r(k+1) \\ i_a(k+1) \\ T_L(k+1) \end{bmatrix} = \begin{bmatrix} 1 & 0.000200 & 0.000004 & -0.000005 \\ 0 & 0.999945 & 0.042383 & -0.052578 \\ 0 & -0.002497 & 0.979644 & 0.000065 \\ 0 & 0.000041 & 0.9 \times 10^{-6} & 0.998035 \end{bmatrix} \begin{bmatrix} \theta_r(k) \\ \omega_r(k) \\ i_a(k) \\ T_L(k) \end{bmatrix} + \begin{bmatrix} 4.4 \times 10^{-9} \\ 0.0000659 \\ 0.0030691 \\ 9.201 \times 10^{-10} \end{bmatrix} V_a(k) \quad (6)$$

In addition, parametric uncertainty will be addressed by using an integrator which is defined in equation (7), where x_I is the integrator state variable, r is the setpoint and $y = \theta_r$ is the output to be controlled.

$$x_I(k+1) = x_I(k) + e(k) = x_I(k) + y(k) - r(k) \quad (7)$$

3 Controller

The armature voltage defined by the controller is

$$V_a(k) = - [K_I \quad K_1 \quad K_2 \quad K_3 \quad K_4] \begin{bmatrix} x_I(k) \\ \theta_r(k) \\ \omega_r(k) \\ i_a(k) \\ T_L(k) \end{bmatrix} \quad (8)$$

The gains K_I, K_1, K_2, K_3 and K_4 are calculated by using the pole placement technique explained in [8]. Also it is possible to calculate the gains by establishing new poles for the closed-loop system and using the scilab command *ppol*. In this case, the gains are

$$[K_I, K] = [0.0006168, 1.2288494, -0.6467532, -4.021708, -2.4009488]$$

The main Scicos program of the implemented controller is shown in figure 1. The main program contains the previously calculated gains K_I, K_1, K_2, K_3 and K_4 which multiply the state variables for defining the armature voltage. In figure 1,

the integrator of equation (7) is implemented by using the $1/Z$ block of the linear palette [9]. The gain A in figure 1 scales the numerical value of the calculated armature voltage. The scaled voltage is the numerical representation of the actual voltage applied to the power converter. The COMEDI A/D block provides access to the data acquisition card which is connected to the power electronics converter.

There are two super blocks in the main program shown in figure 1, one of them is a state observer which estimates the speed, the armature current and the load torque. The other super block is a position measurement algorithm used for conditioning the data of the analog and digital inputs.

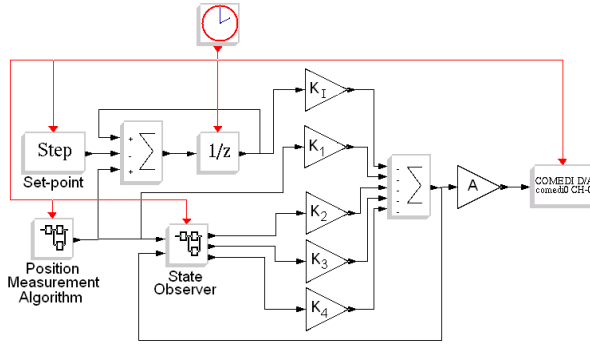


FIGURE 1: Position Control Program in Scicos

4 State Observer

Since the rotor position is the only measured variable, it is necessary to include a state observer for the state feedback. The implemented observer is based on the following equation

$$\hat{x}(k+1) = G\hat{x}(k) + Hu(k) + K_e[y(k) - C\hat{x}(k)] \quad (9)$$

Clearly, G and H are known, they were previously defined in equation (6). The observer gain K_e is a 4×1 constant vector associated with the output error $y(k) - C\hat{x}(k) = \theta_r(k) - \hat{\theta}_r(k)$ and is calculated by employing the scilab command *ppol* [9]. In this case the observer gain is

$$K_e = \begin{bmatrix} 0.0015523 \\ 0.1544085 \\ -0.0392419 \\ -0.0014389 \end{bmatrix} \quad (10)$$

The Scicos program related to the state observer

is shown in figure 2. This program is basically the expression (9) made of scicos blocks.

5 Rotor Position Measurement

A design for measuring the rotor position is implemented. There are several components of this hardware/software design. One of them is the Pepperl+Fuchs encoder that is physical attached to the motor shaft. The second one is a ATmega8535 microcontroller-based signal conditioning circuit and the third one is the real time software that calculates the rotor position from the digital signals provided by the microcontroller. The ATmega8535 microcontroller is programmed for being used as a 16-bit binary UP/DOWN counter. A signal conditioning circuit is necessary to determine the shaft direction, which is implemented by using a flip-flop integrated circuit. The flip-flop uses the pulses of channels A and B of the encoder and provides a binary 1 or 0 depending on the shaft direction. Channel A (or B) of the encoder is then connected to a micro-controller terminal without modifying the voltage as they both the encoder and the microcontroller have 5 volts as a nominal voltage. The microcontroller is in charge of counting the pulses provided by the encoder. In this case, the encoder generates 1024 pulses per revolution. The micro-controller count goes up or down depending on the flip-flop binary signal. The complete electronics diagram of the rotor position measurement setup is shown in figure 3.

The micro-controller provides a 16-bit binary count that represents the rotor position. On the other hand, the National Instruments PCI-6024E data acquisition card has only 8 digital inputs [5]. In order to be able to read the 16-bit binary data, 8 additional digital inputs are obtained by employing 8 analog inputs of the acquisition card and using them as digital inputs. The algorithm to interpret those analog inputs as digital inputs is coded in the Scilab/Scicos source program. The routine that reads binary data by using digital inputs is shown in figure 4. The algorithm consists of multiplying each bit (1 or 0) by its corresponding value in the decimal system, then the results are added. In order to read the data of digital input channels, the COMEDI DI block was used. The same algorithm is employed to convert bits that were obtained by using the analog inputs.

In addition, a signal conditioning routine is employed to use those analog inputs as digital inputs, see figure 5. A comparator determines the binary value (1 or 0) of the signal that was read at the

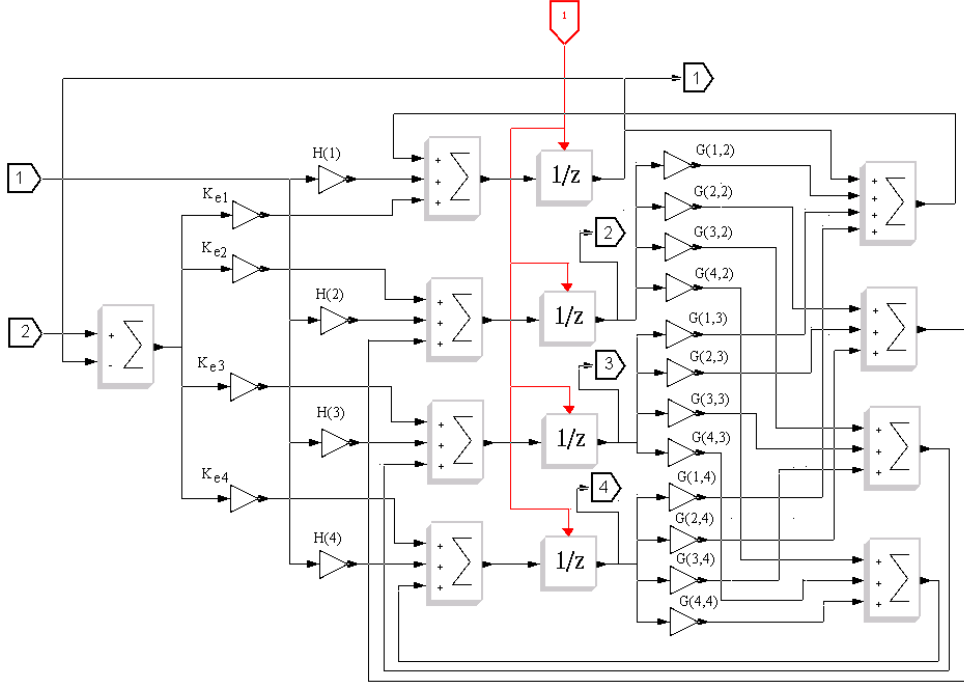


FIGURE 2: *State Observer Program in Scicos*

terminals of the analog input. This routine is implemented for each one of the 8 signals obtained by using the analog inputs. Combining the results provided by these routines a decimal measurement of the rotor position is obtained.

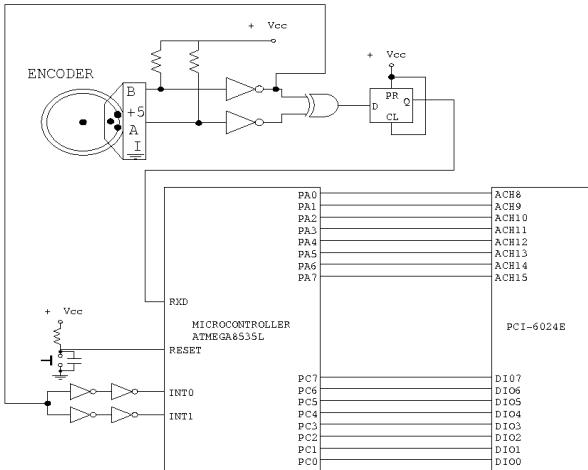


FIGURE 3: *Rotor position measurement setup*

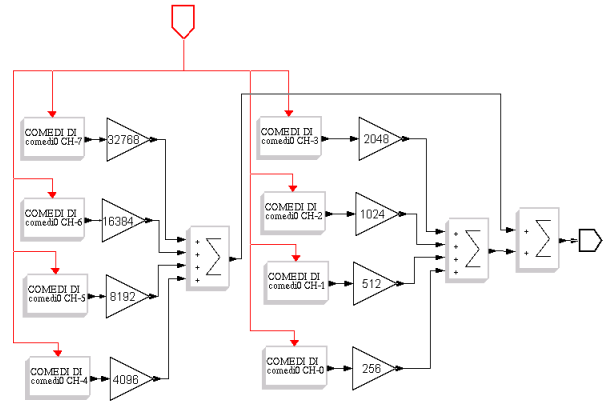


FIGURE 4: *Subroutine associated with the digital inputs of the position measurement algorithm*

A final super block called Position Measurement Algorithm (see figure 1) was created to include the described code, the program contained in this super block is shown in figure 6. This program adds the data of the previous routines. Then, the obtained value is converted into *radians*.

It is important to mention that the rotor position setup is able to measure positive and negative values. In order to accomplish this feature, the microcontroller was programmed to have an initial count that

is located at the middle of the 16-bit count range. Due to the lack of space, reader is referred to [10], where information regarding the ATmega8535 micro-controller assembler code can be found.

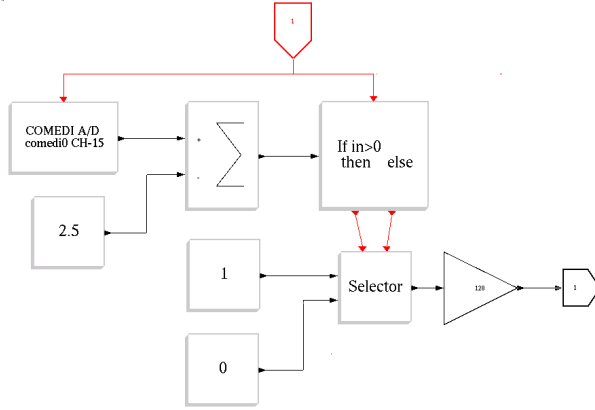


FIGURE 5: *Software-based signal conditioning for one of the eight analog inputs used as digital inputs*

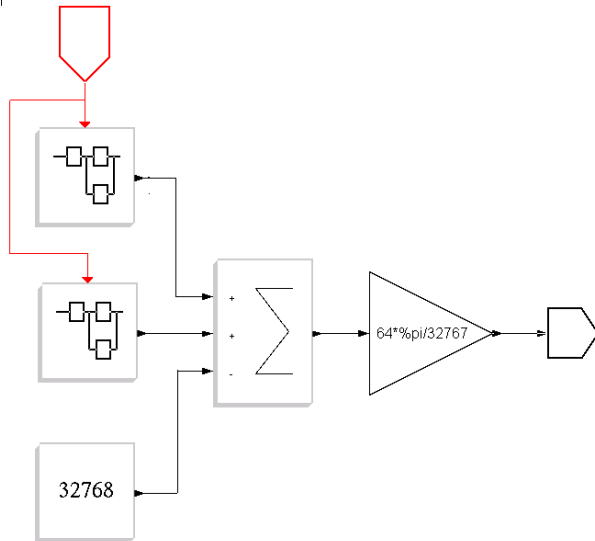


FIGURE 6: *Routine employed for converting measured rotor position values into radians*

6 Power Converter

A Pulse Width Modulation-based MOSFET H-Bridge converter was designed and implemented. This type of power electronics device is commonly used to drive DC motors when bidirectional speed/position control is needed [11]. The numerical value of the armature voltage is defined by the

state feedback and calculated by the computer. This value is written by the real time software to one of the analog output channels of the data acquisition card. The power electronics converter is shown in figure 7.

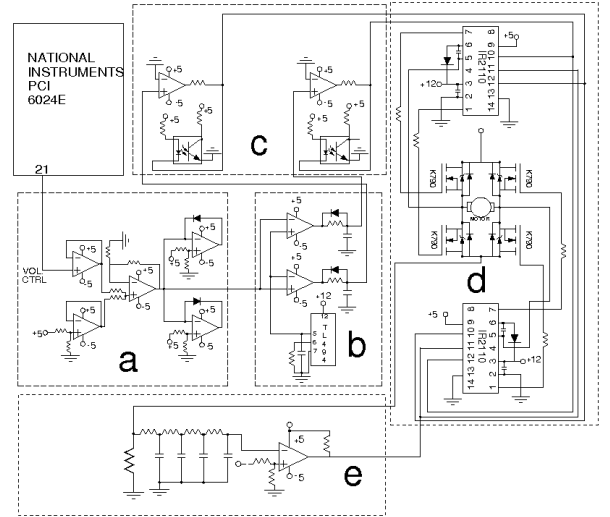


FIGURE 7: *Power Converter. a) Signal Conditioning. b) PWM and switching delay. c) Isolation. d) H Bridge and DC motor. e) Short Circuit Protection*

7 Experimental Results

The final program is compiled by using the RTAicodegen tool in Scicos, this procedure is explained in [7]. The rotor position and the experimental observer-based variables were saved by using FIFOs. In this case, a FIFOin block was employed for each variable and the data were saved by following the procedure indicated in [7].

The experimental trace shown in figure 8 illustrates the dynamic characteristic of the rotor position following a 25.1328 radian (4 revolution) reference command. The simulated model-based trace of the rotor position is also shown in figure 8. It is clear that these signals (simulated and measured) are similar to each other. Initially, the rotor position was at zero radians. The position begins to increase immediately until the position error is close to zero, which occurs approximately at 1.2 sec. Figure 9 shows the experimental observer-based rotor speed. Figure 10 shows the experimental observer-based armature current. The armature voltage computed by the controller is shown in figure 11.

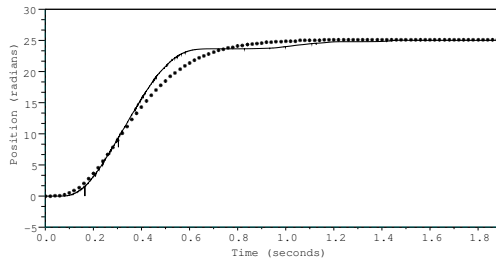


FIGURE 8: *Rotor Position: Experimental measured, ** Simulated model-based*

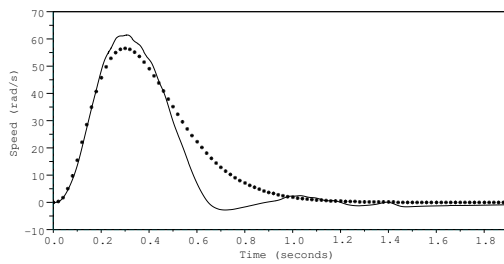


FIGURE 9: *Rotor speed: Experimental observer-based, ** Simulated model-based*

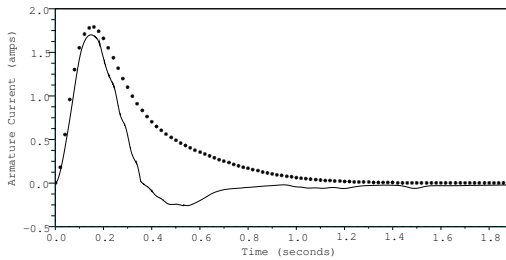


FIGURE 10: *Armature Current: Experimental observer-based, ** Simulated model-based*

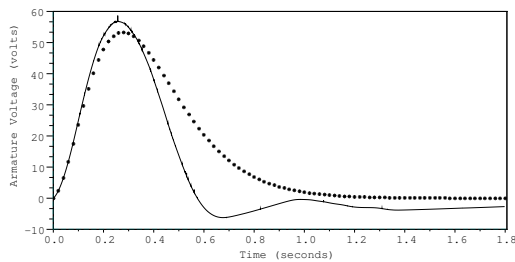


FIGURE 11: *Armature Voltage: Experimental Computed voltage, ** Simulated model-based*

The STEP block is employed in the main program (see figure 1). The parameters of the STEP block can be modified by using the xrtailab interface while the executable program is running [7]. In this case, this feature is used for selecting a set point value. Figure 12 shows the xrtailab interface and 3 *radians* as set point. Figure 13 illustrates the closed-loop system behavior with different set point values, 6π , 12π and 18π (3, 6 and 9 revolutions) respectively.

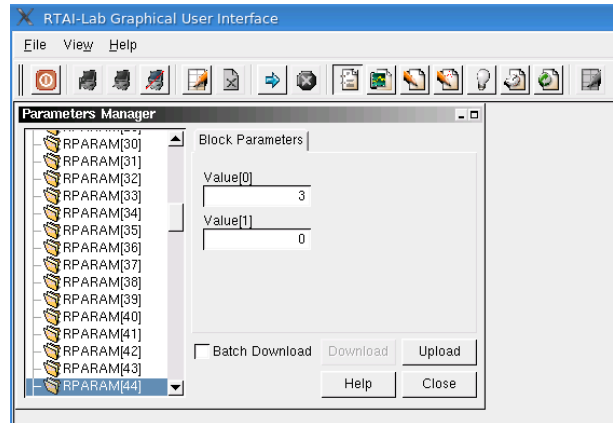


FIGURE 12: *Set point selection in the xrtailab interface*

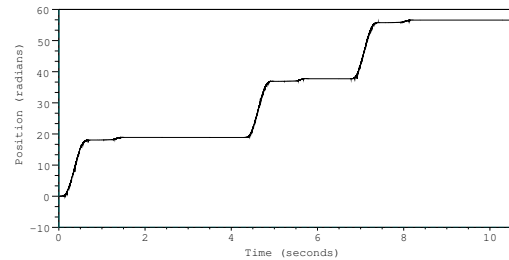


FIGURE 13: *Closed-loop system behavior with different set points*

8 Conclusions

The real time platform employed in this work provides ideal tools for this type of applications. The experimental results are similar to the simulation. The real time software is able to accomplish the control algorithm under strict time requirements. A future work might be related to the idea of implementing a new nonlinear control scheme, keeping the hardware platform and changing just the source program running under the same real time software platform.

9 Acknowledgments

Authors wish to thank Michael Griffin and Carlos A. Cruz-Villar for their valuable suggestions during the early stage of this work. This work was supported in part by the Instituto Tecnológico de Cd. Madero and the Fondo Mixto de Fomento a la Investigación Científica y Tecnológica COTACyT-Gobierno del Estado de Tamaulipas.

References

- [1] Krause, P. C., Wasynczuk, O., y S. D. Sudhoff (2004). ANALYSIS OF ELECTRICAL MACHINERY AND DRIVE SYSTEMS. *John Wiley and Sons, IEEE Press Power Engineering*.
- [2] J. Jugo (2004). *Real-time control of a DC motor using Scilab and RTAI*. SCILAB INRIAROC-QUENCOURT
- [3] Boizot, N., Busvelle, E. y J. Sachau (2007). *High-gain observers and Kalman filtering in hard-time*. RTL 9TH WORKSHOP
- [4] G. Palli. Home Page: <http://www-lar.deis.unibo.it/people/gpalli/>
- [5] National Instruments (2006). PCI-6024E NATIONAL INSTRUMENTS MANUAL.
- [6] D. Schlee, F. Hess, H. Bruyninckx (2003) THE CONTROL AND MEASUREMENT DEVICE INTERFACE HANDBOOK.
- [7] Bucher, R., Mannori, S. y T. Netter (2008). RTAI-LAB TUTORIAL: SCILAB, COMEDI AND REAL-TIME CONTROL.
- [8] Franklin, G. F., Powell, J.D., y M. Workman (1998). DIGITAL CONTROL OF DYNAMIC SYSTEMS. Addison Wesley. Menlo Park, CA.
- [9] Campbell, S. L., Chancellier, J. P. y R. Nikoukhah (2006). MODELING AND SIMULATION IN SCILAB/SCICOS. *Springer*.
- [10] E. Y. Rendon-Fraga. ROTOR POSITION CONTROL OF A DC MOTOR USING A REAL TIME PLATFORM, MSc. THESIS, IN SPANISH. *Instituto Tecnológico de Cd. Madero. Cd. Madero, Mexico; 2009*.
- [11] H. Rashid Muhammed (2004). POWER ELECTRONICS: CIRCUITS, DEVICES AND APPLICATIONS. *Prentice Hall*.

Getting Linux into a train HVAC (Heating, Ventilating and Air Conditioning)

Markus Kreidl

Liebherr Transportation Systems
Liebherrstrasse 1, A-2100 Korneuburg
markus.kreidl@gmail.com

Prof. Nihcolas Mc Guire

OpenTech EDV Research GmbH
Lichtenstein Str 31, A-2130 Mistelbach
der.herr@hofr.at

Abstract

Open source and free software is great constantly growing technology pool consisting of millions of tools, rapidly developing technologies and a solution for nearly every technical problem. Some parts of the open source world are often determined by asynchronous development and some parts are determined by a kind of strict, slim and goal oriented work cycles. Both parts can be a prolific environment for new technologies and innovations. Industry - more specific railway industry - is one the most conservative markets with respect to technologies, standards and the corresponding products. Those products are developed over years adhering to a strict life cycle and must be maintained over decades. Typically railway solutions (products) are embedded solutions with the focus on low resource and safety constraints. The open source community offers many solutions, which fit into those industrial applications but some of them cannot be used because of the gap between the "Fast and Cool" open source world and the long-term life cycle focused industry. This paper is motivated by the possibility to minimize the gap between the open source and a standards dominated worlds, to show ways how to integrate open source into industrial DLCs, to give facts between proprietary and Linux total development costs and to give a feeling how many systems in different vertical markets are already in use and finally to decrease prejudices (on both sides).

1 Introduction

FLOSS (free libre open source software) over the past years has developed methods and tools to create diverse packages with a high code quality. More and more companies are realizing that those packages, driven by a strong community, can replace commercial products. During the last decades industry developed various processes and standards to assure code and product quality. Most of those process have been followed by companies for a very long time. So industry has no mechanisms at hand to integrated FLOSS in their daily work and simply don't know how to deal with FLOSS.

2 Definition FLOSS

In general FLOSS (Free/libre open source software) is the higher abstraction for open source and free software.

2.1 Free software definition[1]

"Free software" is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech," not as in "free beer".

Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve

the software. More precisely, it refers to four kinds of freedom, for the users of the software:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and change it to make it do what you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits (freedom 3). Again, access to the source code is a precondition for this.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either gratis or charging a fee for distribution or the services related to implementing changes, to anyone anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission.

2.2 Open source definition[2]

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

3. **Derived Works:** The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
4. **Integrity of The Author's Source Code:** The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.
5. **No Discrimination Against Persons or Groups:** The license must not discriminate against any person or group of persons.
6. **No Discrimination Against Fields of Endeavor:** The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.
7. **Distribution of License:** The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
8. **License Must Not Be Specific to a Product:** The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.
9. **License Must Not Restrict Other Software:** The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.
10. **License Must Be Technology-Neutral:** No provision of the license may be predicated on any individual technology or style of interface.

3 What can be interesting for industry?

Many diverse software packages and complete distributions for embedded systems are part of FLOSS and can be of interest for industrial embedded designs. Selecting the right FLOSS components can fit into almost any design. A basic embedded system consists of a suitable kernel for an architecture, a library which defines system calls and a large amount of generic user land applications. FLOSS components also contains a wide range of tools for building, debugging, tracing and maintaining software. The Linux kernel covers a wide range of 14 different architectures. The Linux kernel is a conservative implementation of a well-specified Unix OS. Despite a rapid community development, portability and documentations make the integration easy. Any Unix-like operating system needs a C library: the library that contains kernel space user space boundary (system calls) and standard I/O and generic functionalities. A wide range of different diverse user land applications is available to provide a full-featured embedded system, which can fulfill challenging industrial requirements. Compilers, build systems, debug and code editing tools are available. Architecture depended tools like compilers and debuggers are covering all Linux kernel target architectures. FLOSS also contains diverse components for bug tracking, source control management and test benches.

3.1 Industrial software development life cycle

What does development cycles do for the industry? A DLC defines required tasks for specifying, developing and maintaining software. A DLC process gives all participants during a development of a product the possibility to interact in a well-defined way. This should give companies the possibility to make repeatable, and predictable developments and to improve of productivity and assure quality, but this is ideal. A typical simplified software DLC would be:

- Definition of software requirements
- Design
- Implementation
- System test
- Maintenance

4 Open Source DLCs

Open source development is not dominated by DLC like industrial product development. Standardization, a strict organization and dedicated responsibilities are instruments to increase the probability of a high code quality. Open source development life cycles attributes are:

- Async package development: FLOSS packages are developed in extremely modularized manner by different cultural parts all over the world.
- Peer review: Agreements are achieved by open public review process.
- Infrastructure: Bug-fixing databases and source control management are maintained in a global way.
- Well defined policies: Well maintained FLOSS packages provide transparent and traceable policies for the package life cycle.
- Integration aspects: FLOSS packages are integrated in a wide range of applications, distributions and architectures.
- Feedback mechanism: On of the most distinctive features of the FLOSS DLC is its feedback mechanism a side of high the high quantity feedback due to open infrastructure (e.g. mailing lists) the feedback quality is also high informed users produce informed feedback.

DLC processes are applied in several big distributions. Debian for example exercises a strict development life cycles with dedicated responsibilities down to the package level. The infrastructure for administration, versioning, test benches, source control management is highly consistent developed to handle a global distributed development at a high level. The reliability of this process is well reflected in the industrial usage.

top500 super computer server statistics [3]

Operating system Family	Market share	
	Count	[%]
Linux	443	88.6
Windows	5	1.0
UNIX	22	4.4
BSD Based	1	0.2
Mixed	29	5.8

Mobile communication sector (Thousands of Units)[4]

Company	Market Shares 2008	
	Sales	%
Symbian	72,933.5	52.4
Research In Motion	23,149.0	16.6
MS Windows Mobile	16,498.1	11.8
Mac OS	11,417.5	8.2
Linux	11,262.9	8.1
Palm OS	2,507.2	1.8
Others	1,519.7	1.1

A marked research from the group Venture Development Corp. (VDC) [5] researched willingness to migrate to Linux from a commercial OS. The result of the survey shows that a embedded system manufacturers are increasingly favouring Linux as possible choice.

Future projects	%
Opt free in-house Linux	71
Plan to use a commercial Linux	16
Plan to use a Commercial OS	12
Stick at a in-house OS	1
Usage in current projects	%
Commercial OS	39
Linux free in-house	29
Linux commercial	12
No usage of a formal OS	7

Referring to VDC Linux is a an attractive OS choice for embedded development teams. The main reasons in their opinion are:

- Royalty free run-time costs
- Advanced networking capabilities
- Technical features and the large
- Base of engineers familiar with the OS.

A statistic from Market Share [6] showed that Linux based Desktops in July 2009 reached a market share of 1%. In general the workstation market is a slow market which is hard to enter.

5 Kernel DLC

5.1 Developer Hierarchy

To understand the Linux kernel development it is important to know something about the hierarchical organization. Structuring in well defined subsystems is a base for the four levels - ordinary developer, maintainer, trusted lieutenants, and the benevolent dictator - developer hierarchy. An ordinary developer can write patches or proposals and submits his work to a maintainer of a particular component of the kernel. The driver maintainer sends the tested patch to the subsystem maintainers, who are in direct contact with benevolent dictator, which is Linus Torvalds at the moment. He writes code and issues the general direction and plays an integrative role. Andrew

Morton also takes a special place, he is the maintainer of the Linux 2.6 kernel series.

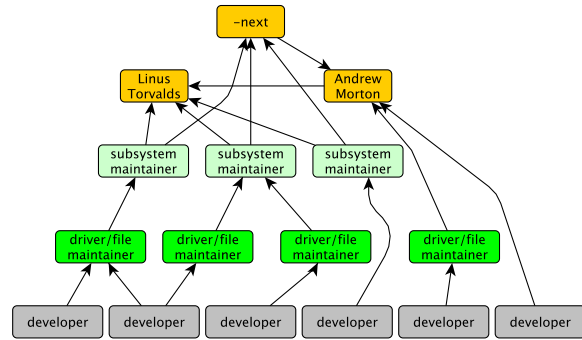


FIGURE 1: Developer hierarchy

For the kernel release 2.6.27 a total set of 10625 patches were submitted from 1015 persons. Averagely, every person submitted 10 patch sets[7].

5.2 The staged process

The kernel development DLC is a well-defined strict process. The process was adapted and optimized over the years according the needs of a global active community.

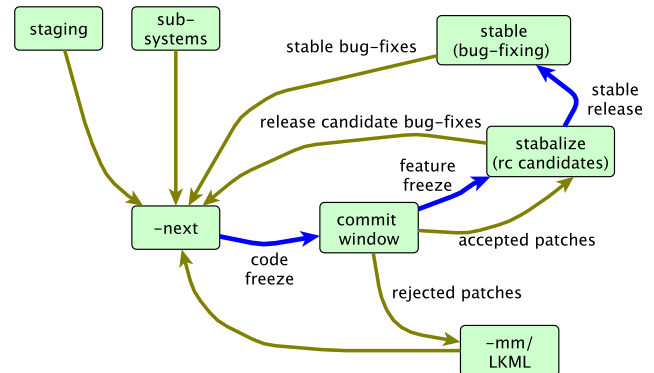


FIGURE 2: Linux DLC

Linux-staging tree The linux-staging tree was created to hold drivers and file systems and other semi-major additions to the Linux kernel that are not ready to be merged at this point in time. It is here for companies and authors to get a wider range of testing, and to allow for other members of the community to help with the development of these features for the eventual inclusion into the main kernel tree Greg Kroah-Hartman (employed by Novell working in the Suse Labs division)

Linux-next tree The linux-next tree contains patches that have the possibility to go into mainline kernel after the current release gets stable. This should help Linus Torvalds to integrate between the subsystems

balanced patches and Andrew Morton can work on future kernel versions without drifting away from the current release. The linux-next tree setup consists of an automated build system, with different build configurations where developer is notified by mail to correct their faults.

The rules for linux-staging:

The linux-next integration testing is not a judgment of your code. The purpose of linux-next is for integration testing and to lower the impact of conflicts between subsystems in the next merge window. You need to ensure that the patches/commits in your tree/series have been:

- submitted under GPL v2 (or later) and include the Contributor's Signed-off-by,
- posted to the relevant mailing list,
- reviewed by you (or another maintainer of your subsystem tree),
- successfully unit tested, and
- destined for the current or next Linux merge window.

LKML (Linux kernel mailing list) For the Linux kernel development is the Linux kernel mailing list the main mailing list. The majority of the announcements, discussions, debates and coordinates are done. The LKML is the the top level mailing list of many different subsystems mailing lists. The LKML is a very high level mailing list which usually receives 200 to 300 mails each day. The mailing list can be seen as the principal communication channel.

The current kernel release is the base for the development. After a 2 weeks the merge window is closed, and a 7 to 10 weeks stabilizing phase start. This time is determined for small adaptation, testing and integration. Various automated test benches should assure the quality of the system.

6 FLOSS usage in industrial environments

To choose single packages like a xml parser for application development is a common way. For embedded systems it is not common to utilize single packages and to build your own system out of these. Choosing a whole distribution moves a lot of work into the community, so the company has reduced work with package compatibilities and the danger of using orphaned packages. Different scenarios for integrating FLOSS in an DLC industrial environment are:

- Grab and run

The Grab and run method seems to be the easiest way for companies to start development. A software snapshot is taken and used decoupled from

the community. A reason for companies doing it this way is, that they have everything they want for the moment and do not need more. In fact they need a certain amount of time to integrate the software but they have no philosophy how to benefit from the community in the future. There commonly is not even a strategy how to deal with bug fixes, new releases and maintaining....

- Integrate the community

This method has the advantage of consuming the advantages of a big community without paying something back. In this model the company is an active consumer that means there have been methods integrated to dealing with FLOSS during a product life cycle. This way of dealing with FLOSS has no plans to move IP independent developments into the community. The drawback is that the company has no say in what direction development will go in the future. Further more the willingness of the community to accept needs from a particular company will depend on this companies acceptance in the community - pay in cash or pay in code...

- Integrate into the community

This model is similar to the integrate the community. The company gives improvements or drivers back to the community. These modifications feed back in to the community get tested by a bigger spectrum and the overall quality increase. Another positive effect is more extensive bug tracking and peer review. When a company decides to work in a stronger relationship with FLOSS it is important, to be sure about licenses, to be aware the community redundancy, to assign time for communication and to not violate/modify interfaces.

7 Total Cost of Development

The total cost development (TCD) research [8] from the company Embedded Market Forecasters (EMF) is used to reflect how different companies deal with Linux or proprietary operating systems. EMF has analyzed a survey of 606 developers and has reflected the design undertakings and design outcome in relation of embedded Linux versus commercial RTOS.

The following parameters have been analyzed:

- Time from design start to product shipment (design time in month)
- Total lines of new and modified source lines – excluding reuse, commercial off-the-shelf and FLOSS
- Total number of source lines of code written – including reuse, commercial off-the-shelf and FLOSS.
- Percentage of designs that are behind schedule.

- Once the integration cycle starts, developers are asked to grade the accuracy of final design hardware and software versus pre-design expectations. Expectations in performance, systems functionality, and features and schedule are examined. EMF records the percentage of responses for which the final design achieved within 20% of pre-design expectations and within 30% of pre-design expectations.

Vendors Compared: Linux and commercial OSes

Linux	Proprietary OSes
Wind River Linux	Green Hills Integrity
LynxWorks BlueCat Linux	LynxWorks LynxOS
CodeWarrior for Linux	Mentor Graphics Nucleus
MontaVista Linux	Microsoft Windows XPE
RedHat Linux (Ent.)	Microsoft Windows CE
Timesys Linux	Wind River VxWorks

7.1 Design comparison

The following table presents a comparative design undertakings between RTOSes and Linux. Total lines of code written and lines of code indicates the complexity of the undertaking. Costs of design are reflected by

- The average number of software developers per project
- Designs behind schedule
- Number of months behind schedule

	RTOSes	Linux	Ind. Avg
Avg # SW developers/project	40.5	47.5	30.9
Avg lines of code written x1000	198.5	252.6	186.3
Avg lines of code x1000	704.4	1000	749.3
Total design time (months)	15.5	14.4	15.5
% behind schedule	43.0%	35.8%	41.4%
Avg months off-target for behind schedule projects	3.8	3.9	4

The survey says that taking a superior time-to-market can save a few ten thousands of dollars difference in up-front costs. As a result of the compared data from 2006 and 2007 showed that 7 more developers per project were need in Linux projects than commercial RTOSes while the average design time for Linux project were 1.1 months less. In fact this study gives a reason for convincing to evaluate non commercial OSes.

7.2 Project expectations

EMF examined for 2 years data from more than 1300 respondents. Broadly data was gathered across 10 different vertical markets from a number of different OSes. The following table represents the collected data:

expectations	RTOSes	Linux	Ind. Avg.
below 20% of Performance	66.8	70.0	64.0
Sys functionality	68.4	71.3	65.5
Features & schedule	55.6	59.5	54.3
below 30% of Performance	74.6	75.6	71.8
Sys functionality	75.4	76.2	72.1
Features & schedule	68.1	67.9	64.7

The table presents the results of the data where the results were below a 20% and a 30% pre-design expectations. A significant difference exist between Linux and commercial RTOS design outcomes.

7.3 Linux vendor comparison

EMF took a look at the comparative data between established commercial Linux vendors and the in-house (roll your own Linux). The analysis showed that in-house developers wrote fewer lines of code and fared poorly when it came to final design outcomes compared to their pre design expectations. A plus of the in-house Linux development is the high level of expertise and the lowered level of complexity. The value for insufficient resources stays nearly at the same level for proprietary RTOSes, Linux and in-house Linux. Proprietary RTOSes have a little advance of 2%. A big plus has commercial Linux against RTOSes and in-house Linux regarding the time to develop their concepts and bring them to the market in-time. An in-house Linux is a question of the in-house available capabilities and the willingness of the management to invest in there engeneers.

7.4 TCD Analysis

EMF has compared the results of the survey in the year 2003 and 2007 and asked the question why the results for embedded Linux-based designs have improved over this 4 years. The answer is tools. In 2003 tools were very limited and 40% of Linux designs involved in-house sets or limited tools that were available for free download. Different commercial Linux companies developed a comprehensive set of design tools or aquired the assets of other companies like Wind River from FSM Labs. In fact nowadays Linux has the same design outcomes compared with traditional RTOSes.

7.5 TCD Conclusions

The development of embedded Linux design changed a lot in the last years. In general designing with a Linux OS is no longer an expensive and risky undertaking. The use of a commercial Linux OS is more cost effective to an in-house Linux development undertaking. The spectrum of Linux for embedded designs is no longer restricted to a few applications and the designs outcomes data demonstrates the developers can consider using a Linux OS without concern that their product will inferior to end products for a proprietary RTOS.

It should be noted though that some of these numbers may well change in the near future as some of them may be in part due to Linux having only a short history in many companies and thus exhibiting "startup"-costs.

8 Do's and don'ts

8.1 Risk assessment

Before integrating a single piece of software or a whole distributions it is important to make a risk assessment to check out what pain may occur in the future. A fundamental risk assessment needs some time at the beginning, but can save a lot of time and money. A risk assessment that answers the following question can easily be integrated to an industrial DLC during the design phase.

- How is the project/distribution history? Are there many forks? Is the project/distribution developed continuously?
- How big is the community behind the project or distribution?
- How fast is the responsiveness?
- The quality of documentation and bug tracking?
- Can the company deal with different licenses?
- Bugs/patches total received
- Project/mailstone planing

8.2 Do not stick at a version

Sticking at a version is a often used strategy because for the moment everything is satisfied and the effort is straight forward but in fact a decoupling between in-house and the community happens unnoticed. Open source automation development labs (OSADL) has realized this problem, so OSADL offers a stable linux kernel version for over one and a half year that contains bug fixes of versions ahead the current stable version. A continuous upgrade strategy avoids a lot of problem that are occuring when backports a done. In general it is better when defining requirements not to refer to packages + versions number or to refer to a special cpu like powerpc 405gr, but ensure that requirements are generic allowing to satisfy them with current FLOSS offerings.

9 Conclusion

9.1 DLCs related

In general when companies want to use FLOSS in their products they have to adapt their software development process in some parts. That means a longer evaluations phase with a risk assessment during the desing phase is a must. The FLOSS community efforts have resulted in diverse packages for different problems, but the needs for the corresponding projects must be strictly defined to select the best-fit variant. Further more the introduced risk assessment limits risks for the future respectivly to get awareness to what can happen in future.

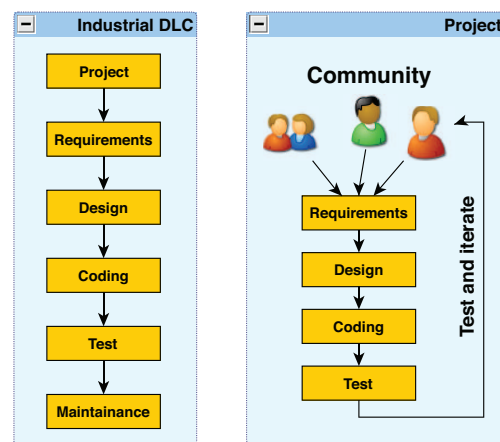


FIGURE 3: *Linux DLC*

The left diagramm shows an industrial DLC which passes straight through its phases. Requirements are made based on the project requirements. The coding phase is done based on the architectural design. Each module or function is programmed once. A module and a integration test is done in the test phase. The main-tainance phase can contains tasks to keep the product up and running.

A lot of FLOSS components are not unique like the Linux kernel is. Different needs of different communities are resulting in diverse FLOSS packages from competing project groups. In general FLOSS software has a continuous development cycle, software will be designed, tested and new iteration of design will be initiated.

This behaviour can be a problem for an industrial DLC, but with a risk assessment during the design phase and a FLOSS supervision task during maintainance this can make an integration possible. This supervision task should give engineers the possiblity to monitor (evaluate change logs when a new version is released and monitor the corresbonding bug system) used FLOSS packages and also participate in the design phase where needed.

9.2 Impacts

Going to FLOSS can provide an extrem wide scoped perspectives to companies in the railway market. This perspectives can deliver a great resource for companies to persist in a globalised world and changing market demands. Some companies provided conservative solutions for many years, but embedded technologies has progressed and so modern solutions can not be provided without transition to modern platforms. Sure for modernisation selecting a modern proprietary OS is a possibility, but using FLOSS has a lot more adavantages. FLOSS can bring a wide range of technologies into a company. It is possible to increase know how in the company unit, give engineers the possibility to understand systems in a deeper way. Know how and innovation is nowadays one of the only ways to ensure jobs unless you are a pure production company. Costs, project expectations, complexity and time to market are not an argument against FLOSS, which was shown by the TCD analysis. To minimize risks a Linux vendor can be choosen or a service contract with one of the several embedded Linux support companies can be done. This helps to be in time and speed up know how transfer to the company. For application developers daily work keeps the same. A wide range of reliable source editing tools, debuggers and compilers is available. For embedded platform engineers changes

can be substantial, they have the possibility to go deep into the system, no stone should be left unturned and no dark corner should be left unilluminated. Engineers have to keep up to date what is changing and try to get a feeling what is going on in the community and where the decisions are moving project to. Core developers should stay at the front lines where technology is evolving to bring it into the company - The Linux kernel and FLOSS is a great place to get into innovations.

References

- [1] <http://www.gnu.org/philosophy/free-sw.html>
- [2] <http://www.opensource.org/docs/definition.php>
- [3] <http://www.top500.org/stats/list/33/osfam>
- [4] <http://www.gartner.com/it/page.jsp?id=910112>
- [5] <http://www.eetindia.co.in/SEARCH/ART/embedded+Linux.HTM>, 2007-10-08 Linux a favourable choice for embedded market
- [6] <http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8>
- [7] http://remword.com/kps_result/2_6_27_lt.html
- [8] http://embeddedforecast.com/images/Embedded_Linux_TCD_Analyzee

A New Real-time Network protocol-Node Order Protocol

Li chanjuan, Nicholas Mc Guire, Zhou Qingguo

Distributed and Embedded System Lab

School of Information Science and Engineering

Lanzhou University

Tianshui South Road 222, Lanzhou, P.R.China

lichanjuan04@gmail.com, mcguire@lzu.edu.cn, zhouqg@lzu.edu.cn

Abstract

It is natural for real-time network protocols to start at the top level with the temporal behavior, so do many of the existing protocols. However, time-triggered protocols not only add extra complexity because related to maintenance of global time but also incur a penalty in network utilisation due to temporal padding. To relieve those issues, we introduce a node ordering concept in which a strict node order is defined and the order of communication events is established by this pre-configured order instead of global time. Thus what is required is maintaining the node order only. Based on such a concept, we propose Node Ordering Protocol (NOP) in which medium access is controlled by strict node ordering. In NOP, there is no concept of global time and communication activities are event-triggered. So NOP eliminates the interval reserved for keeping temporal order and can saturate the network in the general case.

In this paper, we first describe communication semantics of NOP. Then the fault hypothesis is analysed and error detection and handling is introduced in detail. Finally, we supplement consideration in practical implementation of NOP.

1 Introduction

In distributed embedded real-time system, Time Division Multiple Access (TDMA) is a widely used method to access shared medium. In TDMA, the point in time when a node obtains the right to transmit a frame is determined by the progression of real time. This requires that a global time-base is available at all nodes. In a TDMA-based system, the total channel capacity is statically divided into fixed number of slots and unique sending slots is assigned to every node. Thus the protocols based on TDMA have the obvious disadvantage that the slot is not used if a node does not want to transmit a message. This property of TDMA is called temporal padding. Moreover, the intrinsic limitation of global time [1] must be considered, when establishing total order of events in a time-triggered protocol. The determination of the occurrence of the communication event has to be based on sparse time base, instead of dense real-time base. Therefore the network has to be idle

between two active intervals[2].

Finally, the physical clock has negative effect on the precision of the global time due to its physical properties (e.g. sensitiveness to the environment, aging, drift...). Thus communication protocols relying on a well synchronized clock encounter heavy performance penalties and unexpected errors caused by clock synchronization failures[3].

To get rid of these issues, we propose a new protocol, the Node Order Protocol (NOP), which determines the order of accessing nodes to the shared medium by specifying the node order a priori (statically configured order in the simplest case). This results in a communication protocol which is no longer time-triggered, but event triggered. The advantages of NOP are, that the global time is removed from the protocol layer, and thus the temporal padding is eliminated, further failures in time synchronisation do not necessarily lead to a node or system failure. This leads to a tremendous improvement of the network utilisation and also simplifies the protocol de-

mands. However, global time is very important for many applications of distributed systems. Since we eliminated the global time from the underlying communication protocol, we relocate it and put it on top of the communication protocol. This is the contrary approach of most existing protocols as shown in Figure 1. In this paper, we will introduce the concept behind NOP and its communication semantics. After that we analyze the reliability of NOP by stating error detection and containment in detail.

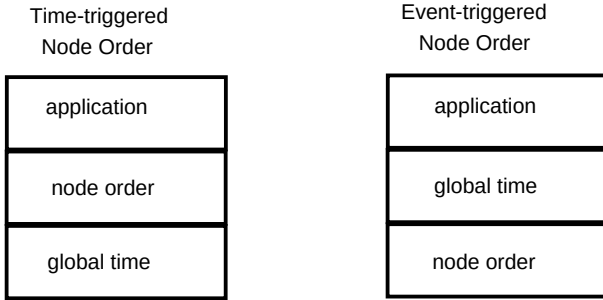


FIGURE 1: *Logical Layer of TTP and NOP*

2 Related work

There are other protocols with the goal to improve the utility of bandwidth in TDMA. One example are TDMA protocols with slot skipping (TDMA/SS)[4], which try to identify unused slots and skip them. TDMA/SS uses an `access_counter` which is set, if after the start of a slot a period of time has passed, and no message has been sent. Now the unused slot can be skipped and the next node is allowed to send his message earlier. TDMA/SS effectively avoids wasted slot in TDMA, but it is still a time-triggered protocol and depends on a global time base. Further this reclamation mechanism has a fairly high latency - which reduces the effectiveness of this measure. POWERLINK [5] proposes another solution to this issue. POWERLINK maintains fixed length cycle. Instead of dividing the cycle into slots, it uses a center node to assign bandwidth in a request/response way. The node can access the medium only, when the center node is polling it. Time slot is based on communication events instead of fixed length. In addition, POWERLINK allows several nodes multiplex a time slot if the period of message is longer than POWERLINK cycle which prevents from wasting network bandwidth. Though POWERLINK integrates event-driven model to improve utility of network, it also relied on well synchronized clocks.

3 Node Ordering Concept

The objective of global time in communication protocols is to maintain the temporal order of occurrences of events and construct a causal order of event chains. While most available communication protocols try to achieve this objective, with a global time base which can be set into relation with real-time, we suggest a solution, which (in contrast to the Lamport algorithm[6]), is based a node ordering concept, in which all nodes follow a strict order to issue communication events. Based on this ordering, the temporal order of events can be derived and form an totally ordered set. From this totally ordered set, even causal order can be reconstructed. From the perspective of node ordering, network communication protocols in distributed embedded systems can be classified into two categories:

- ordering protocols
- non ordering protocols

TCP/IP, UDP/IP and CAN are obviously non ordering protocols, because the nodes transmit in a random order. Profibus seem to be a node ordering protocol, but the ordering can be reset during run-time. Examples for ordering protocols are TTP and TDM, since TDM provides a round-robin media arbitration, while communication based on a message delivery list (MEDL) is used in TTP [7]. The problem of ordering and non-ordering protocols is in principle not related to the temporal behavior of the connections, though in the two mentioned ordering protocols, the ordering is bounded to global time. We claim that the node order is the main objective, and that temporal order is one way to achieve the goal. In NOP, it is necessary for each node to monitor the correctness of ordering during runtime, in order to detect a potentially faulty node. Hence, multicast or broadcast is required in the LAN.

4 Design of NOP

Though there are many protocols around, there is none designed from the start point of node ordering. The NOP protocol we propose in this paper is designed entirely around node ordering concept without temporal ordering at all. In the following section, the design details will be introduced and the potential advantage of the protocol will be illustrated.

4.1 Basic Communication Semantics

For NOP to work, the minimum requirement is that all nodes agree on a sending order. In principle, this is a sufficient criteria for NOP. In a practical environment, at least one further requirement has to be added: a timeout for the receipt of packets has to be introduced. Timeouts in NOP need not to be based on a global time, as long as the drift of all clocks is within an upper bound (which can be monitored by exchanging time stamps). The implementation details of timeout will be introduced in section 5.2.

Thus the configuration parameters of NOP are a MEDL, which assigns an order of the nodes and a TIMEOUT value (the maximum waiting time for a packet). Based on these two preassigned parameters, we begin to introduce the basic communication semantics of NOP.

- Message receive verification
 - Keep on listening to the network for at most TIMEOUT interval.
 - If a frame is received before the end of TIMEOUT, the receiver performs a series of fault tests to the received frame. If the frame passes the verification, then mark the state of NOP to NOP_OPERATION, else set it to NOP_SUSPEND.
- Message sending
 - If the NOP state is NOP_OPERATION, a message in message buffer is sent out. If no message to send, KEEPALIVE is copied to message buffer and is transmitted.
 - When the state is NOP_SUSPEND(timeout occurs in this node), a timeout message is sent.

As we mentioned above, there is no global time in NOP, all communication activities are event-driven. The big advantage of NOP is, that it keeps the network saturated all the time, as long as no error occurs. Figure 2 shows a comparison of the medium access assignment in TDMA and NOP. As you can see, the length of each slot is decided by the length of the transmitted frame in NOP, the only limitation is the MTU of underlying network. If the network is in light load, only KEEPALIVE messages are transmitted and the length of cycle is quite shorter compared to TDMA. Although there is no fixed cycle in NOP, the end-to-end delay of a message is bounded by the

number of node and the maximum frame size (MTU - Maximum Transmission Unit).

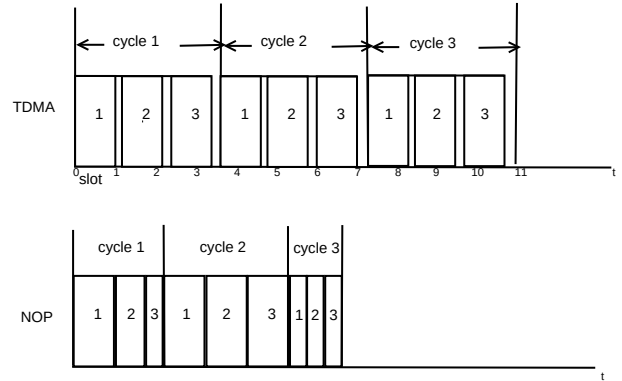


FIGURE 2: Comparison of TDMA with NOP

4.2 Failure Mode in NOP

So far, we have only considered the error free case. Error detection and handling have not been included in the communication semantics presented so far. The simple setup presented above, leads to a number of potential faults. In this section we will describe what strategies NOP uses, to make the protocol robust. The fault hypothesis of NOP is defined as follows:

- omission failures
 - at sender side
 - at receiver side
- crash failure
 - sender node crashes
 - receiver node crashes
- clock performance failure
 - timeout too early
 - timeout too late
- receipt of out-of-order frames
- babbling idiot failure

The failures produced by the physical medium are not list above, but they will be considered during the discussion on omission failures.

In NOP, there are basically only two possible error cases:

- a timeout occurs, while waiting for a frames
- an out of order frame is received

these are the erroneous behaviour, which can detect all the failures listed above.

4.3 Error Detection and Containment in NOP

In this section, we will discuss the error detection mechanism in detail. At the same time, a recovery strategy for omission failures is introduced.

Obviously, the node failure and loss of frame at sender side will result in a timeout event, and is detected by other nodes by strict node ordering. In most case, however, clock failure in NOP can be tolerated and has no negative effect on NOP, because communication activities in NOP is event-triggered. Only when the failed clock issues incorrect timeout events, it can be identified. I will take clock failure as an example to illustrate the mechanism of NOP to detect the failure. The process is shown in figure 3. In early timeout case, C timeouts before A's message arrival. Thus C will assume A fails. Until it receives the frame from A, it knows that it timeouts too early and exits quietly because it is not its turn to access medium. So, A and B will timeout when they wait for C's message and identify C failed.

From the figure 3, we can see, that the error detection mechanism in NOP is simple and easy to implement. Each node only has to maintain the state informations (NOP state and expected sender number), which is easy accessible.

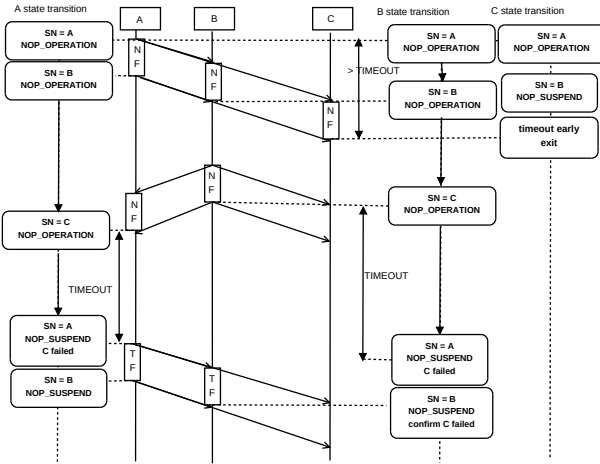


FIGURE 3: Early Timeout Failure

It is fairly simple to detect and handle the out-of-order frame by ignoring it and going on waiting for a correct frame. If a node is sending out messages which conflict with the MEDL (or the MEDL at the sending side is corrupted), then this will be detected by TIMEOUT handler (that is the expected message from the node does not arrive).

In NOP, there is no bus guard to prevent an babbling idiot node from monopolizing the network as TTP implements it in hardware. However, babbling

idiot node will issue out-of-order message. Thus this failure can be detected by adding the out-of-order counter on a per node basis at receiver side. If the node clogs the network, the counter will exceed the threshold value and triggers an action to shut down the network. Otherwise, the timeout condition will be hitted and the node will be removed from the alive node list.

Errors in a NOP based LAN are basically system level errors. It is not necessary to provide membership protocols at this point. In addition, it would not make much sense to do this at the protocol level as it is very application domain dependent to react adequate to the loss of communication.

Considering that a lost frame can render a network unusable, the availability of the protocol is not very satisfying. So a simple recovery strategy is proposed to tolerate the loss of a frame at receiver side. To recover the lost frame, retransmission is obviously required. The only usable way to do this is that every node retransmits the Nth-1 message along with its current message. This allows recovery without an extra delay of communication traffic in theory. Though this strategy may half the effective bandwidth available in the system, it more or less eliminates the ability of transient errors. However, this strategy is not able to trivially recover from dual omission faults.

5 Implementation Consideration

5.1 Physical Layer

NOP itself does not depend on a specific physical layer. However, for we decided to base NOP on a 100BASE-X Ethernet for our practical implementation. To reduce the transmission jitter and delay, all nodes in the NOP network are connected through hubs instead of switches.

5.2 Timeout Implementation

The TIMEOUT interval is an important parameter in NOP. As mentioned above, timeout management is simply provided if the clock drifts are known. In practice, this can not be assumed. We considered two possible strategies to tackle this problem. One is, to assume a reasonable boundary on clock drift for example 10^{-4} for a quartz crystal. During run time the drift is measured and a violation of the assumption is detected. This strategy is easy to implement

and has no effect on the communication semantics. Obviously, the maximum drift will impact the performance and availability of NOP. Another way is to continuously measure the maximum clock drift of the nodes and adjust the timeout value. To implement this strategy, it is necessary to extend the communication semantics to include a time stamp difference of each node in its message header, thus allowing all nodes to monitor the clock drift and correct the maximum drift in the system. This strategy does not work well at present and needs further investigation.

6 Conclusions

In this paper, we proposed a new event-triggered protocol called NOP, which is based on a node ordering concept. The paper points out the main advantages of such a protocol. To sum it up, these advantages are:

- NOP is based on strict node ordering concept, thus removing dependence on global time at the protocol level. This makes the protocol fairly simple and easy to validate. At the same time, the network utilization is more economic than a TDMA based protocol. Moreover, the TIMEOUT mechanism makes sure that the protocol is robust to the impact of the drift effect of the physical clocks (i.e. no byzantine clock problem).
- Communication activities in NOP are event-triggered and the waste of bandwidth caused by the temporal padding in time-triggered protocols is eliminated while retaining the advantage of a handshake-free protocol like TTP.
- Error detection in NOP is easy to implement and requires very limited state information, which also implies that the implementation in temporal and spatial redundancy is fairly easy in NOP.
- Time of error detection is within a well defined time window bounded by the TIMEOUT value.

However, strict node ordering is not a general requirement of distributed systems. So NOP can solve and relieve the issues of a particular problem domain, most notably those that are now using TDMA or TTP, but it does not provide a general solution for communication. The target domain for NOP currently is tightly coupled systems in the domain of safety related systems (i.e. TMR).

7 Acknowledgment

Special thanks to Nicholas Mc Guire, my supervisor who gives me a such interesting topic and provides plenty of important instructions. Thanks to DSLab for offering the necessary resources and appropriate environment to do my research work.

References

- [1] *Real-Time Systems Design Principles for Distributed Embedded Applications*, Hermann Kopetz, 1997.
- [2] *Sparse Time versus Dense Time in Distributed Real-Time systems*, H.Kopetz, 12th Int. Conf. on Distributed Computing Systems, 1992
- [3] *The Effect of Large Clock Drift on Performance of Event and Time Triggered Network Interfaces*, Dawid Trawczynski, Januaz Sosnowski, Junusz Zalewski, 2007, DepCoS-RELCOMEX'07
- [4] *Analyzing TDMA with Slot Skipping*, B.Andersson, E.Tovar, N.Pereira, 2005, RTSS'05
- [5] *Ethernet POWERLINK Communication Profile Specification version 1.1.0*
- [6] *Time, Clocks and the Ordering of Events in a Distributed System*, Leslie Lamport, 1978, Communication of the ACM.
- [7] *TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems*, H.Kopetz, G. Grunsteidl, 1993

SIL4Linux: An attempt to explore Linux satisfying SIL4 in some restrictive conditions

Lijuan Wang, Chuande Zhang, Zhangjin Wu, Nicolas Mc Guire and Qingguo Zhou

Distributed and Embedded System Lab

School of Information Science and Engineering Lanzhou University

Tianshui South Road 222,Lanzhou,P.R.C

zhangchd.lzu.edu@gmail.com

Abstract

Linux is an existing widely-used operating system in lots of fields, including desktop applications, server solutions, embedded systems and even some real time controlling environment with the rt-preempt extensions. And it works well without any big problem currently, but for it is a complex and large system, some potential uncertain factors may influence its stability, so there is no guarantee to use it in some safety-critical environment.

In this paper, we will try to explore the possibility of Linux satisfying SIL 4 in some restrictive conditions. To achieve such a goal, a sil4linux system have been designed and implemented via integrating some kernel tracing/profiling tools, two formal analyzing methods, and with the support of a DBMS.

1 Introduction

In the modern world mankind depends on quite a number of safety-critical systems. Sometimes this dependability is obvious as on the subject of aircraft construction and sometimes it is hidden in tiny embedded systems like ones that trigger a fire alarm. Reference[1] states requirements to safe operating systems assessing Linux for safety related systems. Reference[2] point out some main issue of using COTS/OSS software in the context of 61508 compliant safety related systems, and also it is the initial idea of the whole SIL4linux project.

The SIL4linux project is attempt to find some available methods to explore the possibility that Linux is suitable for use in many safety related applications with SIL 4 integrity requirements.

In this article, we outline an architecture of the primary step of the sil4linux system, and its implementation, usage,related tools and try to outline the possible method to progress on the whole project. It should be noted that this paper just gives the first step of the SIL4Linux project to explore the possibility of using Linux or modified versions of Linux

for SIL 4 applications.

2 Background

Before showing what and how we have done on the SIL4Linux project, we should interpret some basic concepts. The following sections will introduce SIL, SIL4, FMEA and FTA.

2.1 SIL

A SIL[3] is a measure of safety system performance, or probability of failure on demand (PFD) for a SIF or SIS. There are four discrete integrity levels associated with SIL. Both ISA and IEC have agreed that there are three categories: SIL 1, 2 and 3. IEC also includes an additional level SIL 4 that ISA does not. The higher the SIL level, the lower the probability of failure on demand for the safety system and the better the system performance. SIL 4[3] is the highest level of risk reduction that can be obtained through a Safety Instrumented System.

2.2 FMEA

FMEA[4] is a procedure for analysis of potential failure modes within a system for classification by severity or determination of the effect of failures on the system. Its functions as following:

- Recognize and evaluate the potential failure of a product/process and the effects of the failure.
- Identify actions that could eliminate or reduce the chance of the potential failure occurring.
- Complementary to the process of defining what a design or process must do to satisfy the customers.

2.3 FTA

FTA is

- an analysis method with Fault Tree
- an analysis to display the relationship of different fault
- a deductive method from top-level event not inductive

3 Put Linux in some restrictive conditions

For Linux is very complex and large, we need put it in some specific conditions: specific applications (including specific configuration, arguments and environments) with specific kernel executing paths in kernel internal. For describing it more conveniently, the basic GNU/Linux architecture is shown in Figure1.

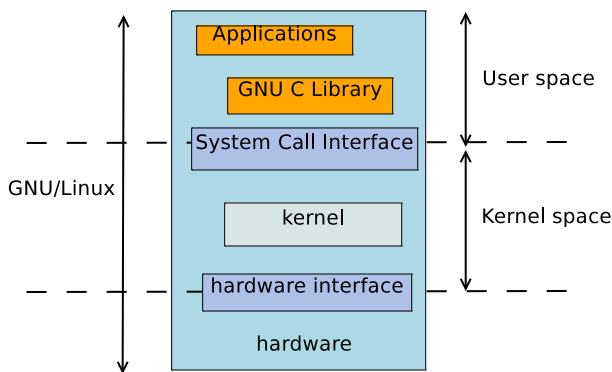


FIGURE 1: Basic GNU/Linux architecture

When running the specific applications, which with relative arguments and configuration in a given environment, the system calls triggered directly by the applications or indirectly via libs will be fixed. And the kernel functions called by the system calls will also be fixed, and the same as the hardware-interfaces. The system calls triggered by the applications can be traced by strace, and the calling paths from the system calls to the hardware-interfaces can be traced by some kernel tracing tools such as kft[5], ftrace[6]. And if we map the system calls and kernel functions to the relative source code files via some tools like ctags, we can analyze them via some formal tools. And further, if we running the application enough times and profile the kernel in source code line level, we will trace the most-often executed lines or blocks(several lines together) of the source code in the kernel functions, and then the hotspots to evaluate will be shorted. There is really such a tool for profiling kernel, it is kgcov[7], and the user-space tool gcov[8] can help us map the profiling result to the source code files for further analyzing like the autotags does. The tools and their functions as following:

- autostrace: find out the system calls called by the applications
- autokft: trace all of kernel functions called by the system calls
- autotags: find out the source code files who defines the relative kernel functions in the path of the Linux kernel directory
- autokgcov: test the coverage of the code in the kernel functions(kernel space)
- autogcov: test the coverage of the code in the kernel functions(user space)

4 Come to whole sil4linux system

The formal two parts have introduced the principle of our sil4linux project, now a whole sil4linux will be discussed, including the design architecture, implementation and usage.

4.1 System architecture

The whole system architecture is given in figure 2 to emphasize the main components in the system.

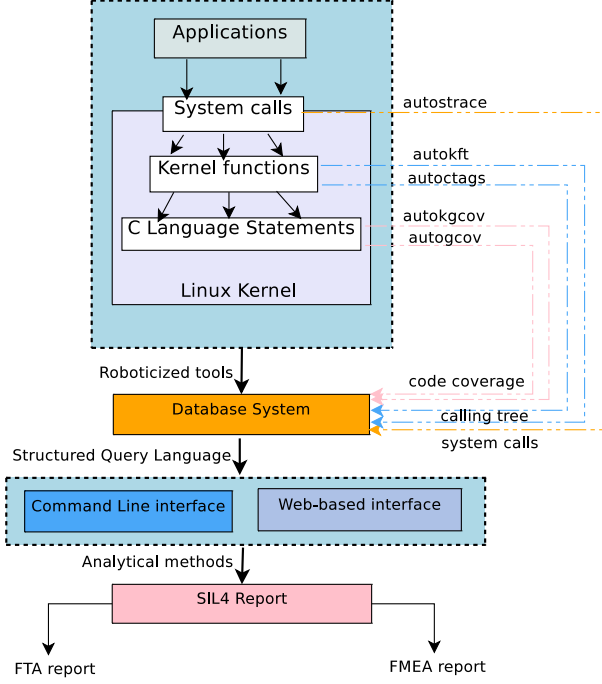


FIGURE 2: *sil4linux architecture*

In the above architecture, there are four parts:

- Roboticized tools
- Database system
- Interface
- Result report

The roboticized tools are autostrace, autokft, autotags, autokgcov, autogcov. All the tools and their functions have presented in section 3.

As the goal of the SIL4Linux project is trying to provide information about the possibility of using Linux for SIL4 application. But as we know, Linux kernel is too complex to be analyzed directly. So we should try to find out a representative sub-collection of it and then analyze it. For the whole structure of Linux like a “big” tree, and it’s too big to be analyzed directly. However, if some methods can find out the representative trees appear frequently (perhaps relative with the special application area), then just analyze some of these trees. So if using some special test-suites which cover most of the system calls, and do enough numbers of tracing with strace, kft and gcov as many as possible, then may get the representative trees. And there is really such a test-suites named POSIX test-suite. The Open POSIX

Test Suite is an open source test suite with the goal of performing conformance, functional, stress, and performance testing of the functions described in the IEEE Std 1003.1-2001 System Interfaces specification[9].

4.2 System design

4.2.1 Database design

To save the result of all the tools, by selecting a database management system (this project use PostgreSQL), and using SQL to do some relative statistic and analyzing.

what need save including the experiment environment, the system calls, the kernel functions, the calling trees of the system calls, the program parts (code blocks) of the kernel functions, and some relative information, such as the the files who define the relative functions. So we design a database named sil4 with different tables to store the above information.

4.2.2 Interfaces design

For simplifying the operation of the SIL evaluating procedure, two different interfaces have been designed, one is the command line interface, another is the web-based interface.

1. CLI interface. In CLI interface, we have two main script kft_query and gcov_query. kft_query is a shell script which designed to query the kft relative tables in the sil4 database, it including

- list all of the system calls
- list the the source code files (in Linux kernel) who define the system calls
- list the calling trees called by the system calls

gcov_query is a tool for searching the Gcov relative information, mainly used to search three different types of source code of the kernel functions.

2. Web Interface. We can access the data which selected from sil4 project via web-based interface by provide a user name and password to login.

4.3 Implementation

The implementation includes several parts:

- the implementation of automatic tools in the system architectures: autostrace, autokft, autotags, autokgcov, autogcov. These have been discussed in the above parts.
- the implementation of the database system: here using PostgreSQL DBMS to manage the database. PostgreSQL[10] is an object-relational database system that has the features of traditional commercial database systems with enhancements to be found in next-generation DBMS systems. And PostgreSQL is free and the complete source code is available.
- the implementation of the interface which have shown in the above part.

And the following picture will show some results of our sil4linux project.

system calls of 2.6.28-rc3 :

System Calls	Trace Results	Call Trees	Max time	Min time	Avg time	Standard deviation time
sys_access	100	59	41635	59	1460.52	545.90
sys_alarm	100	26	1002458	42	10493.89	9877.25
sys_brk	100	22	124	3	5.78	1.22
sys_clock_getres	100	32	861230	6	20936.44	10368.27
sys_clock_gettime	100	11	879930	9	16834.29	10654.61
sys_clock_nanosleep	100	46	36650	3	417.39	364.71
sys_clock_settime	100	100	515605	707	8875.97	5149.04
sys_clone	100	100	41371	977	10138.87	654.52
sys_close	100	90	33957	7	793.71	385.52
sys_dup	100	37	46758	9	1246.89	633.70
sys_dup2	100	89	33770	4	4648.23	1037.64
sys_execve	100	100	48406	856	16716.40	819.32
sys_exit	100	100	1185300	2622	123840.90	20348.30
sys_exit_group	100	100	56127	2675	24724.77	925.36
sys_fcntl64	100	77	41487	5	2856.36	877.45
sys_fstat64	100	20	1185	10	33.11	11.92

FIGURE 3: a part of system calls of Linux kernel 2.6.28-r3

From the above figure, you can get information about the system calls, trace result, call trees, Max time, Min time, Average time and standard deviation time. When click the call trees of sys_geteuid32, the information is shown in the following figure.

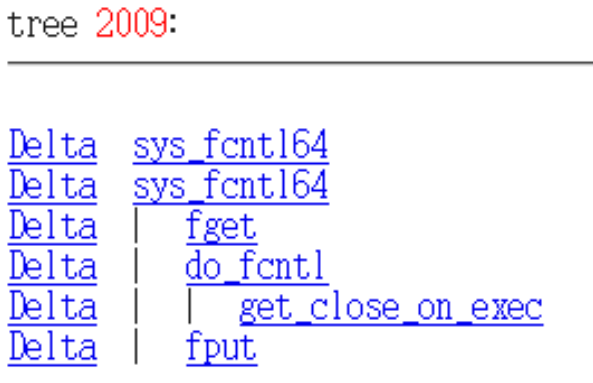


FIGURE 4: trees of sys_geteuid32

4.4 Usage

Before evaluating the kernel whether satisfying SIL 4, there are some procedure need do at first. First, need to install sil4linux system and basic tools, such as gcc, python, PostgreSQL, apache, ctags, posix-testsuite and so on. And some special tools such as FMEA, FTA, gcov, kft, strace and so on. Second, we need to run autostrace, autokft, autotags and autogcov. Then import the result to the database. Because we have create tables in the database, we only run the relative script to import different result files to the database. Third, we need script to access all results via command line interface and web-based interface. Finally, we can analyze results in the formal method.

5 The extra functions sil4linux can do

The extra functions the sil4linux can do including:

- it can help for Linux kernel study and development like LXR[11] does.
- guides for Linux kernel performance tuning in some specific application, for example, running Baidu or Google Spider in Linux, and run sil4linux for it, and then tuning Linux kernel for running Baidu & Google Spider quicker.
- some other potential database dig, such as compare the different system features when enabling/disabling kernel configuration, when migrating Linux kernel from a version to another and so forth.

6 Conclusions

In the modern world we live nowadays, mankind depends on a variety of technical, highly sophisticated systems. And the operating system Linux and its kernel is well known and used in thousands of desktop computers, servers and embedded systems all over the world. The thesis gave a possibility of assessing Linux satisfying SIL 4 applications in some restrictive conditions. It would be useful to provide a method that it is possible for Linux would meet a SIL 4 integrity requirement. And SIL4Linux is a project for finding out some available informations to use Linux for SIL4 applications under some restrictive conditions. At last, a archetypal method based on some formal methods, like FMEA, FTA have been designed.

7 Acknowledge

At first, it is very grateful of M. Sc Wu Zhangjin for his great contribution to sil4linux. His work makes all of us know more about SIL 4, KFT, GCOV, Linux kernel, shell script, debugging tools. Thanks a lot to Prof. Nicholas Mc Guire, Dr. Zhou Qingguo and all the other people of DSLab for their discussion on SIL 4 questions. This work was supported by Siemens.

8 Acronyms

- CLI - Command Line Interface
- COTS - Commercial Off The Shelf
- DBMS - DataBase Management System
- FMEA - Failure Modes and Effects Analysis
- FTA - Fault Tree Analysis
- GCOV - test COVerage program for Gnu cc
- IEC - International Electro-technical Commission
- ISA - Instrument Society of America
- KFT - Kernel Function Trace
- LXR - Linux Cross Reference
- OSS - Open Source Software
- PFD - Probability of Failure of Demand

- RRF - Risk Reduction Factor
- SIL - Safety Integrity Level
- SIF - Safety Instrumented Function
- SIS - Safety Instrumented Systems
- SQL - Structure Query Language

References

- [1] intr,H.R. Pierce, *Preliminary Assessment of Linux for safety related systems*,Research Report 011, 2002, HSE, ISBN: 0 7176 2538 9.
- [2] Nicholas Mc Guire,*Linux for Safety Critical Systems in IEC61508 Context*,2007
- [3] http://www.gmsystemsgroup.com/sil/sil_faqs.html
- [4] http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis
- [5] <http://tree.celinuxforum.org/CelfPubWiki/KernelFunctionTrace>
- [6] <http://linux.die.net/man/1/ftrace>
- [7] http://oss.lzu.edu.cn/blog/article.php?tid_2028.html
- [8] <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>
- [9] <http://posixtest.sourceforge.net/>
- [10] <http://en.wikipedia.org/wiki/PostgreSQL>
- [11] <http://lxr.linux.no/>

Finding Origins of Latencies Using Ftrace

Steven Rostedt

Red Hat, Inc.

1801 Varsity Drive, Raleigh, North Carolina 27606, USA

srostedt@redhat.com

Abstract

One of the difficult tasks analyzing Real-Time systems is finding a source/cause of an unexpected latency. Is the latency caused by the application or the kernel? Is it a wake up scheduling latency or a latency caused by interrupts being disabled, or is it a latency caused by preemption being disabled, or a combination of disabled interrupts and preemption.

Ftrace has its origins from the -rt patch [1] latency tracer, and still carries the capabilities to track down latencies. It can catch the maximum wake up latency for the highest priority task. This wake up latency can also be tuned to only trace real-time processes. There is a latency tracer to find the latency of how long interrupts and/or preemption are disabled. The maximum latency is captured and you can even see the functions that were called in the mean time. Ftrace also has a rich array of tracing features that can help determine if latencies are caused by the kernel, or simply are a bi-product of an application.

1 Introduction

Ftrace has its control files in the debugfs system. This is usually mounted in `/sys/kernel/debug`. If it is not already mounted, then you can mount it yourself with:

```
# mount -t debugfs nodev /sys/kernel/debug
# cd /sys/kernel/debug/tracing
# ls
available_events          set_ftrace_notrace
available_filter_functions set_ftrace_pid
available_tracers         set_graph_function
buffer_size_kb            stack_max_size
current_tracer            stack_trace
dyn_ftrace_total_info     sysprof_sample_period
events                   trace
failures                  trace_marker
function_profile_enabled  trace_options
options                   trace_pipe
per_cpu                   trace_stat
printk_formats            tracing_cpumask
README                   tracing_enabled
saved_cmdlines            tracing_max_latency
set_event                 tracing_on
set_ftrace_filter         tracing_thresh
```

As you can tell, there are a lot of files in this directory. We will only be concerning ourselves with those that will help us trace latencies in the system. Those are:

1. `available_tracers`
2. `current_tracer`
3. `events`
4. `trace`
5. `trace_marker`
6. `trace_max_latency`
7. `tracing_on`

The version of the kernel that I am using for this paper is 2.6.31-rc6-rt4.

2 Enabling Plugin Tracers

Looking in the file `available_tracers`, you will see the available tracers that have been configured.

We are most interested in `irqsoff`, `preemptoff`, `preemptirqsoff`, `wakeup_rt`, and `wakeup`. To enable a plugin, you simply echo the name into the `current_tracer` file:

```
# echo preemptoff > current_tracer
```

All examples in this document will assume that you have mounted the `debugfs` directory and changed directory into the `debugfs` tracing directory.

You can see the tracer that is activated by cat'ing the `current_tracer` file.

```
# cat current_tracer
preemptoff
```

To disable the plugin, echo the special `nop` tracer into the `current_tracer` file. The `nop` tracer is special in that it is not a plugin tracer, but lets the user disable all plugins.

```
# echo nop > current_tracer
# cat current_tracer
nop
```

3 Things That Might Cause Latencies

There are various events that can trigger a latency. First, let's define what a latency is. A latency is the time between an event is supposed to occur and when it actually does. The term **latency tracer** is really a misnomer, because the tracing tools do not actually trace latency, but instead it traces events that may cause a latency.

Within the kernel, there are basically four different events that can cause a latency.

1. Interrupts disabled - keeping interrupts from calling their handlers when a device triggers an interrupt to the CPU.
2. Preemption disabled - preventing a process that just woke up from running.
3. Scheduling latency - the time it takes a process to schedule in.
4. Interrupt inversion - the time that an interrupt handler is performing a task that is lower in priority than the task that it preempted.

Ftrace latency tracers can record the first three. But the interrupt inversion is not covered by the latency tracer but can be seen with other tracers.

4 Measuring Interrupts Disabled

Ftrace piggy backs on top of `lockdep` [2] to measure interrupts disabled. `Lockdep` is a tool made by Ingo Molnar that can detect possible deadlock scenarios within the kernel. It keeps track of locks that are taken and can check the order of locks to ensure that two locks are always taken in the proper order. `Lockdep` also makes sure that a `spinlock` that is used within an interrupt is not taken without disabling interrupts. If you do not understand `lockdep`, do not worry, it is beyond the scope of this paper. What is important, is that `lockdep` keeps track of all locations that interrupts are disabled as well as when they are enabled. Ftrace uses this implementation to record when interrupts are disabled and enabled.

Note: ftrace hooks into the `lockdep` infrastructure, but you do not need to enable `lockdep` to use the interrupt tracer. By enabling `lockdep` you will add even more overhead. If you are concerned about measuring latency and not debugging the locking of the kernel, then it is recommended to keep `lockdep` disabled (`CONFIG_PROVE_LOCKING` and `CONFIG_LOCKDEP`).

Measuring the time that interrupts are disabled in the system is key to for analyzing causes of latency. If interrupts are disabled when an event occurs, then that event must wait till interrupts are enabled to continue. The time it must wait is added latency on top of the overhead to get to the event. When interrupts are disabled, a device that sends an interrupt to the CPU will not be noticed until interrupts are re-enabled.

4.1 Irqsoff Latency Tracer

The plugin `irqsoff` is a way to measure times in the kernel that interrupts are disabled. Listings 1, 2, 3, and 4 show the output of running `irqsoff` tracer for just a little while. As you can see by the fact that I needed to break this up over 4 pages, it can get a bit verbose. This is due to the function tracer.

The function tracer (enabled by `CONFIG_FUNCTION_TRACER`) is a way to trace almost all functions in the kernel. When function tracing is enabled, the kernel is compiled with the gcc option `-pg`. This is a profiler that will make all functions call a special function named `mcount`. One would realize

that this could cause a very large overhead, but if the kernel is also configured with dynamic function tracing (`CONFIG_DYNAMIC_FTRACE`) then these calls, when not in use, are converted at run time to `nops`. This allows the function tracer to have zero overhead when not in use. If you do not understand this part, don't worry, you do not need to understand the implementation to use it. Just realize that enabling the dynamic function tracer gives you great power with no overhead.

4.2 The Heisenberg Principle

Any computer scientist (or any scientist for that matter) should be aware of the **Heisenberg Principle** [3]. Basically this means that the act of measuring something can and will modify the result. This is especially true with the interrupt tracer and even more so when the function tracer is enabled. The idea is to trace the time interrupts are disabled, but by adding a tracer to these core functions, it adds a little overhead. By running with the function tracer, it adds even more overhead to the time interrupts are disabled, because we are tracing every function that is called within the critical section.

You do not need to unconfigure the function tracer to keep it from running while tracing interrupt latency. There exists a proc file that lets you disable the function tracer from running at run time.

```
# echo 0 > /proc/sys/kernel/ftrace_enabled
```

This will allow you to find something a bit closer to the actual latency¹. Listing 5 shows the result of a latency trace with the function tracer disabled.

To get a good idea of the overhead, the benchmark test `hackbench` [4] can show the results well. Running `hackbench` with the function tracer enabled yields a test run time of 47.686 seconds and a max latency of 171 microseconds (way above the max that we allow for the real-time kernel). Running `hackbench` with the function tracing disabled, yields a test run of 34.361 seconds and a max latency of 30 microseconds². Note: running `hackbench` with both tracers disabled only took a running time of 9.774 seconds. I do not know the latency because it was not being traced.

Note: when enabling or disabling the function tracing for the latency tracers, it is best to reset the tracer or it may take effect. That is, echo in `nop` into the `current_tracer` file and `irqsoff` again.

¹Note: you must have a space between the 0 and the > otherwise the shell will interpret it as a redirection of standard I/O.

²hackbench did not even get on the radar in this run

³170 is small compared to thousands that the function tracer can do.

5 Reading the Trace

Before we continue to the other tracers, a description of how to read the output is in order. The lines in the Listings of 1, 2, 3 and 4 are numbered. We will go through some of the lines and explain their meanings.

Lines 001 through 018 is the latency tracer header, and is annotated with a '#' at the beginning of the line. Line 001 states the name of the current plugin tracer. Line 003 has the kernel version that is executing (ignore the trace version, that has not changed in a long time). Line 005 has a bit of information. Here we see that the latency trace recorded a 70 microsecond time that interrupts were disabled. This may be different than the last trace entry, but not by much, due to the tracer writing entries after it took the finishing time stamp. The #170/170 means that there was 170 entries printed out of 170 that were recorded. Since the latency trace ftrace plugins are usually small³ the two numbers should always match. But for other tracers, it is quite possible to have the first number smaller than the second due to the trace ring buffer overwriting older data.

The CPU#0 shows that this latency happened on CPU 0. Inside the parenthesis, the VP, KP, SP and HP will always be zero since they are not yet implemented. The M element shows what type of preemption the kernel was configured at. Here it is "preempt" but really should be "preempt-rt". Since the latency tracer has been replaced with the upstream ftrace, this field has not been updated. The other selections of preempt type are "desktop" for `CONFIG_PREEMPT_VOLUNTARY` (kernel preempts only at preemption points) or "server" for `CONFIG_PREEMPT_NONE` (no preemption inside the kernel). The #P:2 shows that there were 2 online CPUs active.

Line 007 shows information about the task that was executing when the latency was recorded. The task here was "sirqtimer/0" with process id 5. The policy shows that it was running under `SCHED_FIFO` (1) where as 0 would be a non real-time running the `SCHED_NORMAL` policy. `SCHED_RR` is represented with 2, `SCHED_BATCH` is 3, and `SCHED_IDLE` is 5. Because this is running under a real-time policy, the nice value can be ignored. The `rt_prio` field is the real-time prio as

maintained in the kernel. This can be a little confusing. Real time priorities for users range from 1 to 99, but these are represented in the kernel as 98 to 0, where the lower the number, the higher the priority. The trace shows the priority to be 49, but that is the kernel's representation. To convert the `rt_prio` to the user priority, subtract it from 99. The user priority of this task is actually 50.

The `trace_hardirqs_off_thunk` is a helper function called from assembly to trace when interrupts are disabled there, usually by entering of an interrupt. When an interrupt occurs, interrupts are disabled. Looking at the first function called on line 020 we can see the APIC timer interrupt went off.

Line 020 starts off with the `cmd` (the kernel name of the task) and the process id. The task at this recording is `bash` and its process id is 2724. Even though the trace header shows the task was `sirq-timer` a schedule switch happened inside this disabling of interrupts and ending task was `sirq-timer`. The next five items are labeled in the header. The first is the CPU number. The second is whether interrupts were disabled. A `d` means that interrupts are disabled. In the `irqs off` trace, all lines should show that interrupts are disabled. When interrupts are enabled a period (.) will be displayed.

The third item is for `need-resched`. When the kernel determines that a schedule should take place because a higher priority task woke up or the current running task is at the end of its time slice, it sets a `need-resched` flag to signal that a schedule should take place. The trace will annotate this with a `N` in that field. Line 093 shows this being set when we wake up the `sirq-timer` task that is of higher priority than the `bash` task. When the `need-resched` flag is not set, a period (.) is displayed.

The forth item denotes if we are in a hard interrupt or soft interrupt. The soft interrupt is a little misnomer because it really only denotes soft interrupts are disabled. Soft interrupts are disabled when ever the kernel is running a soft interrupt, as one soft interrupt can not preempt another. A `h` means that the trace was recorded in an interrupt. A `s` denotes that soft interrupts are disabled or the trace was recording inside a soft interrupt. A `H` denotes that the trace was recorded in an interrupt and soft interrupts are also disabled. Since the Real-Time Linux kernel runs the soft interrupts as threads, the soft interrupt disabling is not applicable. When hard and soft interrupts are enabled, a period (.) is displayed.

The fifth item denotes the preempt disable

depth. When a kernel disables preemption in critical sections⁴, it uses a preempt counter. The preempt count is recorded in all traces, and this field shows the value when it is greater than zero. When preemption is enabled, this field will contain a period (.). Line 041 shows a preempt depth of 1 that was caused by the `_atomic_spin_lock` just before it. Line 046 shows a preempt depth of 2 caused by the `_atomic_spin_lock_irqsave` before it.

Lines 181 and 182 shows the schedule switch that took place between the tasks `bash` and `sirq-timer`. Only the first 8 characters of the task name are printed, as can be seen by the truncated name of the task `sirq-timer`.

6 Preemption Disabled Tracing

When interrupts are disabled, events from devices and timers and even inter-processor communication is disabled. But the kernel can keep interrupts enabled but disable preemption. This allows devices and timers to be able to notify the CPU that an event has happened, but if a task should wake up because of it, it must wait till the kernel comes to a place it can preempt before it will schedule. The `preemptoff` plugin tracer will trace the maximum time that preemption is disabled.

Measuring the time preemption is disabled may be something used for academics, but it has really no practical meaning by itself. Being able to trace the time that both interrupts are disabled and/or preemption is disabled is much more informative. This is the total time that a task can not be scheduled. If interrupts are disabled, no event can occur to cause a preemption. The scheduler will not be called if preemption is enabled but interrupts are not. The `preemptirqsoff` plugin tracer shows this information.

```
[root@mx1 tracing]# echo preemptirqsoff > current_tracer
```

The output for this trace is not much different than the output of the `irqsoff` trace so I will omit it from this paper.

7 Using the Event Tracer

Since the function tracing can add a large overhead it is not always practical to use it. But without the

⁴In this case we can see spin locks disable preemption

function tracing enabled, the information may not be enough to see what is happening. Luckily, there is the **event tracer**. The event tracing is not a plugin. When events are enabled, they will be recorded in any plugin, including the special **nop** plugin.

There are two ways to enable events. One is with the `set_event` file and the other is with the `events` directory. The `set_event` file is a way to echo in events to enable them. The available events are:

```
[root@mx1 tracing]# cat available_events
skb:kfree_skb
block:block_rq_abort
block:block_rq_insert
block:block_rq_issue
block:block_rq_requeue
block:block_rq_complete
block:block_bio_bounce
block:block_bio_complete
block:block_bio_backmerge
block:block_bio_frontmerge
block:block_bio_queue
block:block_getrq
block:block_sleeprq
block:block_plug
block:block_unplug_timer
block:block_unplug_io
block:block_split
block:block_remap
kmem:kmallocc
kmem:kmem_cache_alloc
kmem:kmallocc_node
kmem:kmem_cache_alloc_node
kmem:kfree
kmem:kmem_cache_free
lockdep:lock_acquire
lockdep:lock_release
workqueue:workqueue_insertion
workqueue:workqueue_execution
workqueue:workqueue_creation
workqueue:workqueue_destruction
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
sched:sched_kthread_stop
sched:sched_kthread_stop_ret
sched:sched_wait_task
sched:sched_wakeup
sched:sched_wakeup_new
sched:sched_switch
sched:sched_migrate_task
sched:sched_process_free
sched:sched_process_exit
sched:sched_task_setprio
sched:sched_process_wait
sched:sched_process_fork
sched:sched_signal_send
```

The name before the colon is the system that the event is under. The event name is after the colon⁵. By echoing in the system name you will enable all the events in that system.

```
[root@mx1 tracing]# echo irq > set_event
[root@mx1 tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
```

Echoing in just the event name will enable the event as well. But if there are two event names under two systems that are identical, then both will be enabled. Currently no two event names are identical.

Adding new names follows shell concatenation rules. Using a `'>'` will truncate the file and disable the events that were previously enabled. Using a `'>>'` will add new events without disabling the ones that are currently enabled.

```
[root@mx1 tracing]# echo sched_switch >> set_event
[root@mx1 tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_entry
irq:softirq_exit
sched:sched_switch
```

The `'!'` character can be used to remove events. Note that this is also a bash command so it must be added in quotes.

```
[root@mx1 tracing]# echo '!softirq_entry' >> set_event
[root@mx1 tracing]# cat set_event
irq:irq_handler_entry
irq:irq_handler_exit
irq:softirq_exit
sched:sched_switch
```

The `events` directory is also useful. The directory structure is made of the event systems, and within each system directory is the events. Each level has an `enable` file.

```
[root@mx1 tracing]# ls events/
block      ftrace  header_page  kmem      sched
workqueue  enable  header_event  irq       lockdep  skb
[root@mx1 tracing]# ls events/irq
enable  irq_handler_entry  softirq_entry
filter  irq_handler_exit  softirq_exit
[root@mx1 tracing]# ls events/irq/softirq_exit/
enable  filter  format  id
```

⁵The kernel I have has `lockdep` enabled where you can see from the `lockdep` events

To enable all events echo 1 into `events/enable`. To enable all events within a system, echo 1 into the system enable file (`events/irq/enable`). To enable just a single event, echo 1 into the enable file for that event (`events/irq/softirq_entry/enable`). Echoing in a 0 will disable the same events that a 1 would enable.

Just enabling the scheduling events yields a nice useful output, as seen in Listing 6.

8 Tracing Scheduling Latencies

The time a task is awoken to the time it is scheduled is considered the scheduling latency. Two plugin tracers exist to measure this latency. The `wakeup` plugin will consider all tasks and the `wakeup_rt` will only consider real-time tasks to trace. Both of these plugins only trace the current highest priority task of the system. The trace records the max latency, and tracing anything but the highest priority task would lose the trace for the highest task, because the highest task may cause a lower priority task to take a long time to be scheduled.

If you are concerned about the wake up times of all tasks, simply enable all the scheduling events and examine the trace with the `nop` plugin.

Listing 7 shows the output of the `wakeup` plugin. Notice that the time is quite exaggerated. This is because of the way the wakeup tracer works. The tracer picks the highest priority task that has started. If it wakes up another task of equal priority it does not switch the trace to that task. Although we see that our wake up latency was 150 microseconds, the true wake up was only 7 microseconds. What happened was after the highest priority task `hald-addon-stor` with pid 2103 was woken up, we see that the task `hald` with pid 1952 was woken up afterwards. But since the two tasks have the same priority, the tracer did not switch over to test the wake up time of the second task. The scheduler chose the second task (`hald`) first, and the trace included the entire time that the task `hald` ran. Luckily it only ran for 142 microseconds.

If we chose not to enable the scheduling events we would not have seen the `hald` task wake up and we would assume that the true scheduling latency was 150 microseconds.

The `wakeup_rt` plugin only records real-time tasks. The tracer only records the maximum trace

which makes the `wakeup` plugin hide real-time tasks latencies. If it constantly records the long latency that is described above, then we will never see the latencies of real-time tasks that we care about. This is why there are two plugins to record scheduling latencies. Listing 8 shows the output of the `wakeup_rt` plugin.

The values in the scheduling events needs a little explanation. The first trace item (which is from the `wakeup_rt` tracer) shows that task with pid 6808 at priority 120⁶. The **R** means that it is in the running state⁷. The + denotes that it is waking up the task that follows. The task that is being woken up is the `migration/0` task with the pid of 3. It's priority is zero which is the highest priority Linux supports (99 - 0 = 99 user level priority). The `migration/0` task is in the sleep state denoted by the **S**. The number inside the brackets ([000]) is the CPU that the task that is being woken up on is assigned to. The task may migrate before it wakes up.

The third trace item is a scheduling event (denoted with the `sched_switch:`). This even is the scheduling context switch between the current running task (`bash`) and the task that was just woken up (`migration/0`). This time the first number in the bracket ([120]) is the priority of the current task with the state of the task in parenthesis ((**R**)). The “==>” also denotes a scheduling switch is occurring, followed by the task that is scheduling, its pid and in brackets, its priority ([0]).

The third and last events are pretty much identical. The third event came from enabling the scheduling events, and the last event is part of the `wakeup_rt` plugin tracer.

9 Adding Placeholders into the Trace

When you discover that there exists an unexpected latency in your system and none of the latency plugin tracers showed anything, then you may need to confirm that the issue may be with something in userspace. Assuming that you have access to the source code of the application, you can make have the application write into the tracer ring buffer.

Just enabling the events (for now we'll enable the `sched` and `irq` events) and running with the `nop` tracer, you can watch what is happening with your application.

⁶This is the kernel internal priority. Anything over 100 is a nice value. Here it is 120 - 20 = nice value of zero

⁷This field holds the same enumerations that `top` uses.

Two particular files are of importance.

1. **tracing_on** - Enabling and disabling the tracer.
2. **trace_marker** - Writing into the trace buffer.

Since the tracing facility uses a ring buffer that overwrites older data with newer data, it is not important to enable the trace, but instead let it constantly run. When you hit a point where you detect a latency is when you want to disable it. Having a file descriptor opened to both of the above files lets you see what is happening inside the application as well as stop tracing as soon as a latency is detected. Stopping the trace as soon as it happens is critical since you do not want to overwrite the trace that recorded the latency, as well as it will be easier to find the trouble area if it is relatively close to the end of the trace.

```
[root@mx1 tracing]# echo Hello Dresden > trace_marker
[root@mx1 tracing]# cat trace
# tracer: nop
#
#          TASK-PID      CPU#      TIMESTAMP  FUNCTION
#          | |          |          |          |
#          bash-2702   [001]    8934.777334: 0: Hello Dresden
```

Inside the application, you can add comments to the trace at particular points and use them as markers to what is happening in the kernel.

```
write(trace_mark_fd, "hit this point\n", 15);
```

When you detect a latency inside the application you can stop the trace by writing the ASCII character '0' (zero) to the **tracing_on** file.

```
/* Detected latency, stop the trace */
write(tracing_on_fd, "0", 1);
```

Using this in combination with the event tracers (or even the full function tracer) will help tremendously with finding latency problems in your application.

There is another plugin that is useful with the above: The **syscall** plugin. This is similar to the

strace tool but it traces all programs, not just one. The output ends up in the trace. Mixing the **syscall** plugin along with the event tracing will give lots of useful information to pin point trouble areas in your application.

10 Conclusions

During the early development of the -rt patch, some of our first testers were from audio users. People using the **jack** [5] utility to record music. They found the -rt patch gave the minimum latencies to record without defects. But every so often, they would come across something that would exceed the minimum latency, and would complain to us. Using the early **latency_tracer** we were able to find bugs in the kernel that caused their latencies and fixed them. But there were times that the **latency_tracer** proved that the latency was not in the kernel, and with further investigation, bugs in the **jack** utility were being discovered.

When your application fails to meet a deadline, it can happen due to several issues: hardware, latency in the kernel, or a bug in the application itself. When you discover something has gone wrong, the next step is to find what and where the problem arises. Having a good set of tracing utilities at your disposal will facilitate solving these issues.

References

- [1] *RT Patch*, <http://www.kernel.org/pub/linux/kernel/projects/rt/>, http://rt.wiki.kernel.org/index.php/Main_Page
- [2] *lockdep*, Ingo Molnar, See Documentation/lockdep-design.txt in the Linux kernel source.
- [3] *Heisenberg Principle*, http://en.wikipedia.org/wiki/Uncertainty_principle
- [4] *hackbench*, Rusty Russell, <http://devresources.linux-foundation.org/craiger/hackbench/src/hackbench.c>
- [5] *JackAudio*, <http://jackaudio.org/>

```

001: # tracer: irqsoff
002: #
003: # irqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
004: #
005: # latency: 70 us, #170/170, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
006: #
007: # | task: sirq-timer/0-5 (uid:0 nice:-5 policy:1 rt-prio:49)
008: #
009: #
010: #          _-=====> CPU#
011: #          / _-=====> irqsoff
012: #          | / _-=====> need-resched
013: #          || / _-=====> hardirq/softirq
014: #          ||| / _-=====> preempt-depth
015: #          |||| /
016: #          ||||| delay
017: # cmd      pid      time      caller
018: # \      /      \      /
019: bash-2724 0d... 1us : trace_hardirqs_off_thunk <-save_args
020: bash-2724 0d... 1us : smp_apic_timer_interrupt <-apic_timer_interrupt
021: bash-2724 0d... 2us : apic_write <-smp_apic_timer_interrupt
022: bash-2724 0d... 2us : native_apic_mem_write <-apic_write
023: bash-2724 0d... 2us : exit_idle <-smp_apic_timer_interrupt
024: bash-2724 0d... 3us : irq_enter <-smp_apic_timer_interrupt
025: bash-2724 0d... 3us : rcu_irq_enter <-irq_enter
026: bash-2724 0d... 4us : idle_cpu <-irq_enter
027: bash-2724 0d.h. 4us : hrtimer_interrupt <-smp_apic_timer_interrupt
028: bash-2724 0d.h. 5us : ktime_get <-hrtimer_interrupt
029: bash-2724 0d.h. 5us : clocksource_read <-ktime_get
030: bash-2724 0d.h. 5us : _atomic_spin_lock <-hrtimer_interrupt
031: bash-2724 0d.h1 6us : hrtimer_rt_defer <-hrtimer_interrupt
032: bash-2724 0d.h1 6us : __run_hrtimer <-hrtimer_interrupt
033: bash-2724 0d.h1 7us : __remove_hrtimer <-__run_hrtimer
034: bash-2724 0d.h1 7us : timer_stats_account_hrtimer <-__run_hrtimer
035: bash-2724 0d.h1 7us : _atomic_spin_unlock <-__run_hrtimer
036: bash-2724 0d.h. 8us : tick_sched_timer <-__run_hrtimer
037: bash-2724 0d.h. 8us : ktime_get <-tick_sched_timer
038: bash-2724 0d.h. 8us : clocksource_read <-ktime_get
039: bash-2724 0d.h. 9us : tick_do_update_jiffies64 <-tick_sched_timer
040: bash-2724 0d.h. 9us : _atomic_spin_lock <-tick_do_update_jiffies64
041: bash-2724 0d.h1 10us : do_timer <-tick_do_update_jiffies64
042: bash-2724 0d.h1 10us : update_wall_time <-do_timer
043: bash-2724 0d.h1 10us : clocksource_read <-update_wall_time
044: bash-2724 0d.h1 11us : clocksource_get_next <-update_wall_time
045: bash-2724 0d.h1 11us : _atomic_spin_lock_irqsave <-clocksource_get_next
046: bash-2724 0d.h2 12us : _atomic_spin_unlock_irqrestore <-clocksource_get_next
047: bash-2724 0d.h1 12us : update_vsyscall <-update_wall_time
048: bash-2724 0d.h1 13us : _atomic_spin_lock <-update_vsyscall
049: bash-2724 0d.h2 13us : _atomic_spin_unlock <-update_vsyscall
050: bash-2724 0d.h1 14us : calc_global_load <-do_timer
051: bash-2724 0d.h1 14us : _atomic_spin_unlock <-tick_do_update_jiffies64
052: bash-2724 0d.h. 14us : update_process_times <-tick_sched_timer
053: bash-2724 0d.h. 15us : account_process_tick <-update_process_times
054: bash-2724 0d.h. 15us : account_system_time <-account_process_tick
055: bash-2724 0d.h. 16us : cpuacct_update_stats <-account_system_time
056: bash-2724 0d.h. 16us : __rcu_read_lock <-cpuacct_update_stats
057: bash-2724 0d.h. 17us : __rcu_read_unlock <-cpuacct_update_stats
058: bash-2724 0d.h. 17us : acct_update_integrals <-account_system_time
059: bash-2724 0d.h. 18us : jiffies_to_timeval <-acct_update_integrals
060: bash-2724 0d.h. 18us : run_local_timers <-update_process_times
061: bash-2724 0d.h. 18us : hrtimer_run_queues <-run_local_timers

```

Listing 1: Interrupts Off Latency Trace (Part 1)

```

062:  bash-2724  0d.h.  19us : raise_softirq <-run_local_timers
063:  bash-2724  0d.h.  19us : raise_softirq_irqoff <-raise_softirq
064:  bash-2724  0d.h.  19us : wakeup_softirqd <-raise_softirq_irqoff
065:  bash-2724  0d.h.  20us : wake_up_process <-wakeup_softirqd
066:  bash-2724  0d.h.  20us : try_to_wake_up <-wake_up_process
067:  bash-2724  0d.h.  21us : task_rq_lock <-try_to_wake_up
068:  bash-2724  0d.h.  21us : _atomic_spin_lock <-task_rq_lock
069:  bash-2724  0d.h1  21us : update_rq_clock <-try_to_wake_up
070:  bash-2724  0d.h1  22us : select_task_rq_rt <-try_to_wake_up
071:  bash-2724  0d.h1  22us : activate_task <-try_to_wake_up
072:  bash-2724  0d.h1  23us : enqueue_task <-activate_task
073:  bash-2724  0d.h1  23us : enqueue_task_rt <-enqueue_task
074:  bash-2724  0d.h1  23us : enqueue_rt_entity <-enqueue_task_rt
075:  bash-2724  0d.h1  24us : __enqueue_rt_entity <-enqueue_rt_entity
076:  bash-2724  0d.h1  24us : cpupri_set <-__enqueue_rt_entity
077:  bash-2724  0d.h1  25us : _atomic_spin_lock_irqsave <-cpupri_set
078:  bash-2724  0d.h2  25us : _atomic_spin_unlock_irqrestore <-cpupri_set
079:  bash-2724  0d.h1  26us : _atomic_spin_lock_irqsave <-cpupri_set
080:  bash-2724  0d.h2  26us : _atomic_spin_unlock_irqrestore <-cpupri_set
081:  bash-2724  0d.h1  27us : update_rt_migration <-__enqueue_rt_entity
082:  bash-2724  0d.h1  27us : check_preempt_curr <-try_to_wake_up
083:  bash-2724  0d.h1  28us : check_preempt_wakeup <-check_preempt_curr
084:  bash-2724  0d.h1  28us : update_curr <-check_preempt_wakeup
085:  bash-2724  0d.h1  28us : calc_delta_fair <-update_curr
086:  bash-2724  0d.h1  29us : cpuacct_charge <-update_curr
087:  bash-2724  0d.h1  29us : _rcu_read_lock <-cpuacct_charge
088:  bash-2724  0d.h1  30us : _rcu_read_unlock <-cpuacct_charge
089:  bash-2724  0d.h1  30us : account_group_exec_runtime <-update_curr
090:  bash-2724  0d.h1  30us : resched_task <-check_preempt_wakeup
091:  bash-2724  0d.h1  31us : test_tsk_need_resched <-resched_task
092:  bash-2724  0d.h1  31us : test_ti_thread_flag <-test_tsk_need_resched
093:  bash-2724  0dNh1  32us : task_wake_up_rt <-try_to_wake_up
094:  bash-2724  0dNh1  32us : test_tsk_need_resched <-task_wake_up_rt
095:  bash-2724  0dNh1  32us : test_ti_thread_flag <-test_tsk_need_resched
096:  bash-2724  0dNh1  33us : task_rq_unlock <-try_to_wake_up
097:  bash-2724  0dNh1  33us : _atomic_spin_unlock_irqrestore <-task_rq_unlock
098:  bash-2724  0dNh.  33us : preempt_schedule <-_atomic_spin_unlock_irqrestore
099:  bash-2724  0dNh.  34us : softlockup_tick <-run_local_timers
100:  bash-2724  0dNh.  34us : _touch_softlockup_watchdog <-softlockup_tick
101:  bash-2724  0dNh.  35us : rcu_pending <-update_process_times
102:  bash-2724  0dNh.  35us : rcu_check_callbacks <-update_process_times
103:  bash-2724  0dNh.  36us : idle_cpu <-rcu_check_callbacks
104:  bash-2724  0dNh.  36us : rcu_try_flip <-rcu_check_callbacks
105:  bash-2724  0dNh.  36us : rcupreempt_trace_try_flip_1 <-rcu_try_flip
106:  bash-2724  0dNh.  37us : _atomic_spin_trylock <-rcu_try_flip
107:  bash-2724  0dNh1  37us : rcupreempt_trace_try_flip_a1 <-rcu_try_flip
108:  bash-2724  0dNh1  37us : cpumask_next <-rcu_try_flip
109:  bash-2724  0dNh1  38us : cpumask_next <-rcu_try_flip
110:  bash-2724  0dNh1  38us : cpumask_next <-rcu_try_flip
111:  bash-2724  0dNh1  39us : rcupreempt_trace_try_flip_a2 <-rcu_try_flip
112:  bash-2724  0dNh1  39us : _atomic_spin_unlock_irqrestore <-rcu_try_flip

```

Listing 2: Interrupts Off Latency Trace (Part 2)

```
113:    bash-2724    0dNh.    40us : preempt_schedule <-_atomic_spin_unlock_irqrestore
114:    bash-2724    0dNh.    40us : _atomic_spin_lock_irqsave <-rcu_check_callbacks
115:    bash-2724    0dNh1   40us : rcupreempt_trace_check_callbacks <-rcu_check_callbacks
116:    bash-2724    0dNh1   41us : __rcu_advance_callbacks <-rcu_check_callbacks
117:    bash-2724    0dNh1   41us : _atomic_spin_unlock_irqrestore <-rcu_check_callbacks
118:    bash-2724    0dNh.    42us : preempt_schedule <-_atomic_spin_unlock_irqrestore
119:    bash-2724    0dNh.    42us : scheduler_tick <-update_process_times
120:    bash-2724    0dNh.    42us : ktime_get <-sched_clock_tick
121:    bash-2724    0dNh.    43us : clocksource_read <-ktime_get
122:    bash-2724    0dNh.    43us : _atomic_spin_lock <-scheduler_tick
123:    bash-2724    0dNh1   44us : update_rq_clock <-scheduler_tick
124:    bash-2724    0dNh1   44us : task_tick_fair <-scheduler_tick
125:    bash-2724    0dNh1   44us : update_curr <-task_tick_fair
126:    bash-2724    0dNh1   45us : calc_delta_fair <-update_curr
127:    bash-2724    0dNh1   45us : cpuacct_charge <-update_curr
128:    bash-2724    0dNh1   45us : __rcu_read_lock <-cpuacct_charge
129:    bash-2724    0dNh1   46us : __rcu_read_unlock <-cpuacct_charge
130:    bash-2724    0dNh1   46us : account_group_exec_runtime <-update_curr
131:    bash-2724    0dNh1   47us : _atomic_spin_unlock <-scheduler_tick
132:    bash-2724    0dNh.    48us : preempt_schedule <-_atomic_spin_unlock
133:    bash-2724    0dNh.    48us : perf_counter_task_tick <-scheduler_tick
134:    bash-2724    0dNh.    48us : find_new_ilb <-scheduler_tick
135:    bash-2724    0dNh.    49us : cpumask_first <-find_new_ilb
136:    bash-2724    0dNh.    49us : resched_cpu <-scheduler_tick
137:    bash-2724    0dNh.    49us : _atomic_spin_trylock <-resched_cpu
138:    bash-2724    0dNh1   50us : resched_task <-resched_cpu
139:    bash-2724    0dNh1   50us : test_tsk_need_resched <-resched_task
140:    bash-2724    0dNh1   51us : test_tti_thread_flag <-test_tsk_need_resched
141:    bash-2724    0dNh1   51us : _atomic_spin_unlock_irqrestore <-resched_cpu
142:    bash-2724    0dNh.    52us : preempt_schedule <-_atomic_spin_unlock_irqrestore
143:    bash-2724    0dNh.    52us : run_posix_cpu_timers <-update_process_times
144:    bash-2724    0dNh.    52us : profile_tick <-tick_sched_timer
145:    bash-2724    0dNh.    53us : hrtimer_forward <-tick_sched_timer
146:    bash-2724    0dNh.    53us : _atomic_spin_lock <-__run_hrtimer
147:    bash-2724    0dNh1   54us : enqueue_hrtimer <-__run_hrtimer
148:    bash-2724    0dNh1   54us : _atomic_spin_unlock <-hrtimer_interrupt
149:    bash-2724    0dNh.    55us : preempt_schedule <-_atomic_spin_unlock
150:    bash-2724    0dNh.    55us : tick_program_event <-hrtimer_interrupt
151:    bash-2724    0dNh.    55us : tick_dev_program_event <-tick_program_event
152:    bash-2724    0dNh.    56us : ktime_get <-tick_dev_program_event
153:    bash-2724    0dNh.    56us : clocksource_read <-ktime_get
154:    bash-2724    0dNh.    56us : clockevents_program_event <-tick_dev_program_event
155:    bash-2724    0dNh.    57us : lapic_next_event <-clockevents_program_event
156:    bash-2724    0dNh.    57us : apic_write <-lapic_next_event
157:    bash-2724    0dNh.    57us : native_apic_mem_write <-apic_write
158:    bash-2724    0dNh.    58us : irq_exit <-smp_apic_timer_interrupt
159:    bash-2724    0dN.1   58us : do_softirq <-irq_exit
160:    bash-2724    0dN.1   59us : __do_softirq <-call_softirq
161:    bash-2724    0dN.1   59us : wakeup_softirqd <-__do_softirq
162:    bash-2724    0dN.1   59us : rcu_irq_exit <-irq_exit
163:    bash-2724    0dN.1   60us : idle_cpu <-irq_exit
164:    bash-2724    0dN..   60us : preempt_schedule_irq <-retint_kernel
```

Listing 3: Interrupts Off Latency Trace (Part 3)

```

165:    bash-2724    0dN..    61us : __schedule <-preempt_schedule_irq
166:    bash-2724    0dN..    61us : rcu_qsctr_inc <-__schedule
167:    bash-2724    0dN.1    62us : _atomic_spin_lock_irq <-__schedule
168:    bash-2724    0dN.2    62us : update_rq_clock <-__schedule
169:    bash-2724    0d..2    63us : put_prev_task_fair <-__schedule
170:    bash-2724    0d..2    63us : update_curr <-put_prev_task_fair
171:    bash-2724    0d..2    63us : calc_delta_fair <-update_curr
172:    bash-2724    0d..2    64us : cpuacct_charge <-update_curr
173:    bash-2724    0d..2    64us : __rcu_read_lock <-cpuacct_charge
174:    bash-2724    0d..2    65us : __rcu_read_unlock <-cpuacct_charge
175:    bash-2724    0d..2    65us : account_group_exec_runtime <-update_curr
176:    bash-2724    0d..2    65us : __enqueue_entity <-put_prev_task_fair
177:    bash-2724    0d..2    66us : pick_next_task <-__schedule
178:    bash-2724    0d..2    66us : pick_next_task_rt <-pick_next_task
179:    bash-2724    0d..2    66us : dequeue_pushable_task <-pick_next_task_rt
180:    bash-2724    0d..2    67us : perf_counter_task_sched_out <-__schedule
181:    bash-2724    0d..2    68us : __unlazy_fpu <-__switch_to
182:    sirq-tim-5    0d..2    68us : finish_task_switch <-thread_return
183:    sirq-tim-5    0d..2    68us : needs_post_schedule_rt <-finish_task_switch
184:    sirq-tim-5    0d..2    69us : perf_counter_task_sched_in <-finish_task_switch
185:    sirq-tim-5    0d..2    69us : _atomic_spin_unlock <-finish_task_switch
186:    sirq-tim-5    0d...    70us : trace_hardirqs_on <-schedule
187:    sirq-tim-5    0d...    70us : time_hardirqs_on <-schedule

```

Listing 4: Interrupts Off Latency Trace (Part 4)

```

[root@mx1 tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mx1 tracing]# echo irqsoff > current_tracer
[root@mx1 tracing]# cat tracing_max_latency
19
[root@mx1 tracing]# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
#
# latency: 19 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#
# | task: gnome-settings--2669 (uid:42 nice:0 policy:0 rt_prio:0)
#
#
#
#      _-----=> CPU#
#      / _-----=> irqsoff
#      | / _-----=> need-resched
#      || / _-----=> hardirq/softirq
#      ||| / _-----=> preempt-depth
#      |||| /
#      ||||| /
#      ||||| delay
#      ||||| time | caller
#      cmd \ pid
#      gnome-se-2669 0d... 2us+: trace_hardirqs_off_thunk <-save_args
#      gnome-se-2669 0dN.. 19us : trace_hardirqs_on_thunk <-retint_check
#      gnome-se-2669 0dN.. 20us : time_hardirqs_on <-retint_check

```

Listing 5: Interrupts Off Latency Trace (No Functions)

```
[root@mxl tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxl tracing]# echo 1 > events/sched/enable
[root@mxl tracing]# echo preemptirqsoff > current_tracer
[root@mxl tracing]# cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 2.6.31-rc6-rt4
#
# latency: 12 us, #6/6, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#
# | task: bash-2653 (uid:0 nice:0 policy:0 rt_prio:0)
#
#
#      _-----=> CPU#
#      / _-----=> irqsoff
#      | / _-----=> need-resched
#      || / _-----=> hardirq/softirq
#      ||| / _-----=> preempt-depth
#      |||| /
#      |||||
#      ||||| delay
# cmd      pid      time      caller
# \      /      \      /
bash-2653 1d... 1us+: trace_hardirqsoff <-task_rq_lock
bash-2653 1d..2 3us+: sched_wakeup: task sshd:2650 [120] success=1
bash-2653 1dNh3 9us+: sched_wakeup: task irq/21-uhci_hcd:580 [49] success=1
bash-2653 1dNh3 11us+: sched_wakeup: task irq/21-eth0:2333 [49] success=1
bash-2653 1.N.1 13us : _atomic_spin_unlock_irqrestore <-task_rq_unlock
bash-2653 1.N.1 13us : trace_preempt_on <-task_rq_unlock
```

Listing 6: Preemption and Interrupts Off Latency Trace

```
[root@mxl tracing]# echo 0 > /proc/sys/kernel/ftrace_enabled
[root@mxl tracing]# echo 1 > events/sched/enable
[root@mxl tracing]# echo wakeup > current_tracer
[root@mxl tracing]# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 2.6.31-rc6-rt4
# -----
# latency: 150 us, #7/7, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
# -----
# | task: hald-addon-stor-2103 (uid:0 nice:0 policy:0 rt-prio:0)
# -----
#
#          _-----=> CPU#
#          / _-----=> irqsoff
#          | / _-----=> need-resched
#          || / _-----=> hardirq/softirq
#          ||| / _-----=> preempt-depth
#          |||| /
#          ||||| delay
# cmd \ pid ||||| time | caller
# \ / ||||| \ | /
<idle>-0 1d.h3 0us : 0:140:R + [001] 2103:120:S hald-addon-stor
<idle>-0 1d.h3 1us+: wake_up_process <-hrtimer_wakeup
<idle>-0 1dNh3 3us+: sched_wakeup: task hald:1952 [120] success=1
<idle>-0 1d..3 7us!: sched_switch: task swapper:0 [140] (R) ==> hald:1952 [120]
      hald-1952 1d..3 149us : sched_switch: task hald:1952 [120] (D) ==> hald-addon-stor:2103
      hald-1952 1d..3 150us : __schedule <-schedule
      hald-1952 1d..3 151us : 1952:120:S ==> [001] 2103:120:R hald-addon-stor
```

Listing 7: Scheduling Latency Trace

```
[root@mx f tracing]# echo 0 > /proc/sys/kernel/ftrace-enabled
[root@mx f tracing]# echo 1 > events/sched/enable
[root@mx f tracing]# echo wakeup_rt > current_tracer
[root@mx f tracing]# cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 2.6.31-rc6
#
# latency: 5 us, #5/5, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#
# | task: migration/0-3 (uid:0 nice:-5 policy:1 rt_prio:99)
#
#
#      _-----=> CPU#
#      / _-----=> irq<del>off
#      | / _-----=> need<del>resched
#      || / _-----=> hardirq/softirq
#      ||| / _-----=> preempt<del>depth
#      |||| /
#      |||||
#      ||||| delay
# cmd   pid   time | caller
# \    /
bash-6808 0d..2 0us : 6808:120:R + [000] 3: 0:S migration/0
bash-6808 0d..2 1us+: wake_up_process <-sched_exec
bash-6808 0d..3 4us : sched_switch: task bash:6808 [120] (R) ==> migration/0:3 [0]
bash-6808 0d..3 5us : schedule <-preempt_schedule
bash-6808 0d..3 5us : 6808:120:R ==> [000] 3: 0:R migration/0
```

Listing 8: Scheduling Latency Trace

Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT_RT) on AMD's GEODE LX Platform

Kushal Koolwal

VersaLogic Corporation

3888 Stewart Road, Eugene, OR 97402 USA

kushalk@versalogic.com

Abstract

When it comes to embedded systems, real-time characteristics like low-latency, deterministic behavior and guaranteed calculations are of utmost importance and there has been an increasing market demand for real-time features on embedded computers.

This paper presents results of benchmarking a standard Linux kernel against a real-time Linux kernel (with PREEMPT_RT patch) using the Debian Linux operating system on AMD Geode LX platform board. As the PREEMPT_RT patch (RT patch) matures further and integrates into the mainline Linux kernel, we try to characterize the latency effects (average and worst-case) of this patch on LX-based platform.

The paper starts with a basic introduction of the RT patch and outlines the methodology and environment used for evaluation. Following that, we present our results with appropriate graphs (bar graphs, histograms and scatter plots) and discuss those results. We also look for any performance degradation due to the real-time patch.

The paper concludes with some future work that can be done to further improve our results and discusses some important issues that need to be considered when using the PREEMPT_RT patch.

1 Introduction

Linux has been regarded as an excellent General Purpose Operating System (GPOS) over the past decade. However, recently many projects have been started to modify the Linux kernel to transform it into a Real-time Operating System (RTOS) as well. One such project is PREEMPT_RT patch[1] (also known as RT patch) led by Ingo Molnar and his team. The goal of this patch is to make the Linux kernel more deterministic and reduce the average latency of the Linux operating system.

This paper builds upon “Myths & Realities of Real-Time Linux Software Systems” paper[2]. For basic real-time concepts (in Linux) we strongly recommend you to first read the FAQ paper before further reading this paper.

In this paper we are going to investigate the latency effects of the PREEMPT_RT patch on an

AMD Geode LX800 board. There have been many real-time benchmarking studies (using RT patch) based on Intel, IBM, and ARM platforms such as Pentium, Xeon, Opteron, OMAP, etc. [1,15], but no benchmarking has been done (as of this writing) with PREEMPT_RT patches on AMD's Geode LX platform. Moreover, the Geode platform has certain kind of “virtual” hardware built into it and it would be interesting to find out how does that affect the real-time latencies. The aim of this paper is to assess the RT patch using the 2.6.26 kernel series and discuss its effects on the latency of the Linux OS.

2 Test Environment

2.1 System details

Board Name	EBX-11
Board Revision	5.03
CPU (Processor)	AMD Geode LX800 (500 MHz)
Memory	Swiss-bit PC2700 512 MB
Storage Media	WDC WD200EB-00CPF0 (20.0 GB Hard Drive)
BIOS Version	General Software 5.3.102
Periodic SMI	Disabled
USB2.0/Legacy	Disabled

2.2 Operating System Details

OS Name	Debian 5.0 (Lenny/testing - i386) - Fresh Install
Linux Kernel Version	2.6.26 (without and with RT patch)
RT Patch Version	Patch-2.6.26-rt1
Boot mode	multi-user (Runlevel 2) with "quiet" parameter
Swap space	80MB (/dev/hda5)
ACPI	Off/Minimal

2.3 BIOS/System Settings to reduce large latencies

Following are some System/BIOS settings¹ that are known to induce large latencies (in 100's of msecs) which we need to take care of: - One of the things that make an OS a "good" RTOS is low latency interrupt handling. SMI (System Management Interrupts) interrupts are known to cause large latencies (in several 100's of μ s). Therefore we disable the "Periodic SMM IRQ" option² in the BIOS[3].

- Enable minimal ACPI functionality. Just enable the ACPI support option in kernel and uncheck/disable all the sub-modules. Features like on-

¹For more details about these issues please see: http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application#Latencies

²USB Legacy devices are known to cause large latencies so generally it is a good idea to disable the 'USB legacy option' (if it exists) in the BIOS and to also use PS/2 mouse and keyboard instead of USB. However, disabling the 'SMM IRQ option' in the BIOS takes care of this issue.

³Ideally we would generate the worst-case system load that our application might encounter. Since we are benchmarking in general, we generate an extremely high load which a real-time application is unlikely to encounter. Therefore in a real scenario we might (or might not) get slightly better real-time performance.

⁴Worst-case means system running under load

demand CPU scaling can cause high system latencies[4]. Since 2.6.18-rt6 patch we need the ACPI support to activate "pm_timer" since TSC timer is not suitable for high-resolution timer support.

- All tests were run through an SSH connection. It is not recommended to run the tests on a console as the printk (kernel's print command) can induce very high latencies[3].

3 Test Methodology

3.1 Test Selection

Based on our research there is no one "single" test that would test all the improvements/features of the PREEMPT_RT patch. Therefore, we selected various kinds of test for benchmarking, in order to cover all different metrics of real-time measurements such as interrupt latency, scheduling latency, worst-case latency, etc. A table comparing different tests that we have used follows.

3.2 Test Runs

All the tests were executed with (worst-case) and without (normal) "system load". By system load, we mean a program/script which generates sufficient amount of CPU activity and IO operations (example: reading/writing to disks) to keep the system busy 100% of the time. We wrote a simple shell script to generate this kind of load³. Please see **Appendix I** for the script code.

Normal (without load) run: Here we simply ran the tests without any explicit program (script) generating system load. This is equivalent of running your real-time application under normal circumstances that is an idle system with no explicit programs running.

Worst-case⁴ (with load) run: Here we ran the tests under a heavy system load. This is equivalent of running your real-time application under system load to determine the maximum time your application would take to complete the desired operation in case an unexpected event occurs.

Extended Worst-case (with load) run: In addition to running each of the above mentioned tests, under a Normal and Worst-case scenario for approximately 1 minute, we ran the tests for 24-hours also under a system load. Here, we do not show results of extended tests without any system load because the results were not significantly different from what we observed for running the test for 1 min without any system load (Normal Scenario). To get realistic and reliable results, especially for worst-case latencies, we need to run tests for many hours, preferably at least for 24 hours so that we have at least million readings (if possible) [5,6,11]. Running tests over short durations (for example 1 minute) may fail to reveal all the different paths of code that the kernel might take.

A table comparing the different real-time benchmarks, their features, and their basic principles of operation can be found in **Appendix II**.

3.3 Kernels Tested

For benchmarking purposes we tested four different kernels:

a) **2.6.26-1-486:** At the time we conducted the tests, this was the default Debian kernel that came with Debian Lenny (5.0)[7]. This kernel has no major real-time characteristics.

b) **2.6.26-vl-custom-ebx11:** This is a custom configured Debian Linux kernel that is derived from above kernel. We configure/optimize the kernel so that it runs efficiently on EBX-11 board. This kernel is partly real-time – the option “Preemptible Kernel” (CONFIG_PREEMPT) is selected under kernel configuration menu.

c) **2.6.26-1-rt1-ebx11:** We applied the PREEMPT_RT patch to the above kernel in order to make the Linux kernel completely real-time by selecting the option “Complete Preemption” (CONFIG_PREEMPT_RT).

d) **2.6.18-4-486:** Since kernel 2.6.18, some parts of the PREEMPT_RT patch have been incorporated into main-line kernel. We used this default Debian kernel then (April 2007) to see how the latency, in general, of a default Linux kernel has improved in newer releases like 2.6.26. For example, theoretically

we should see better results for 2.6.26-1-486 compared to 2.6.18-4-486.

4 Test Results

4.1 GTOD Test Results⁵

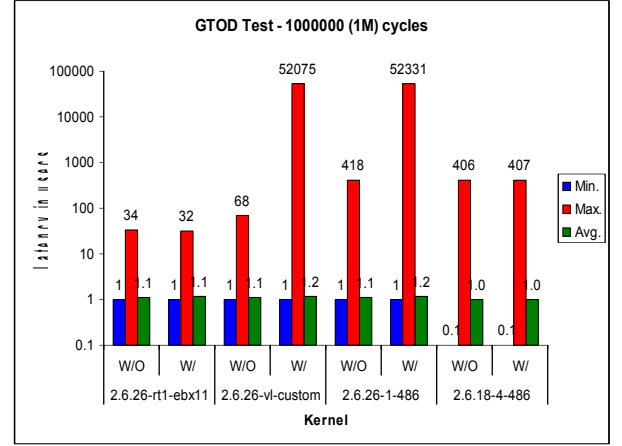


FIGURE 1: GTOD Results

For further details on the test results please refer to **Appendix III (a)**.

Scatter plots for GTOD Test under system load^{6,7}

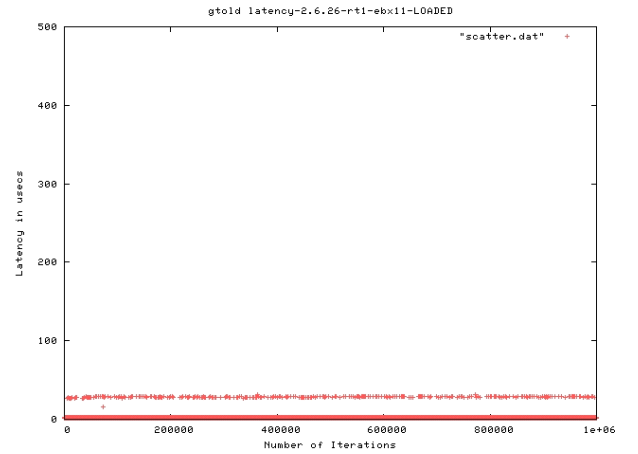


FIGURE 2: 2.6.26-rt1 LOADED

⁵1 second = 1000 milliseconds = 1000000 μs (secs)

W/O = Without system load

W/ = With system load

1M = 1 Million

⁶Wherever required, we have kept the scale of X and Y axes constant across all the graphs (histogram and scatter plot) of each test by converting them into logarithmic scale.

⁷The suffix “LOADED” at the top section of each graph, as in 2.6.26-rt1-ebx11-LOADED, means system was under load.

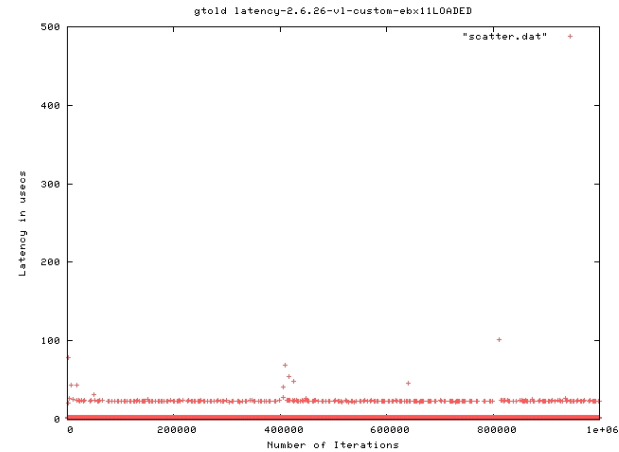


FIGURE 3: 2.6.26-custom LOADED

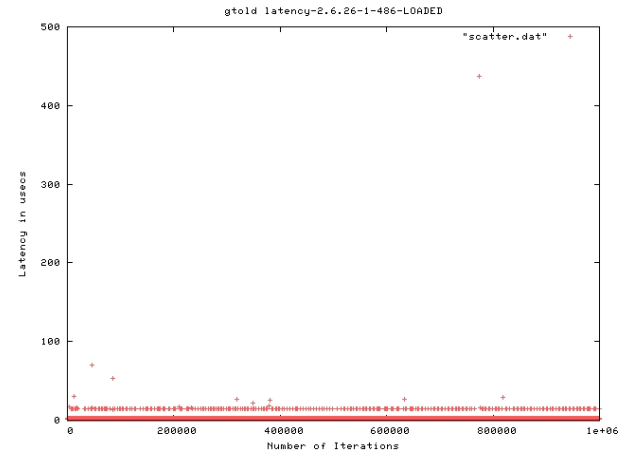


FIGURE 4: 2.6.26-1-486 LOADED

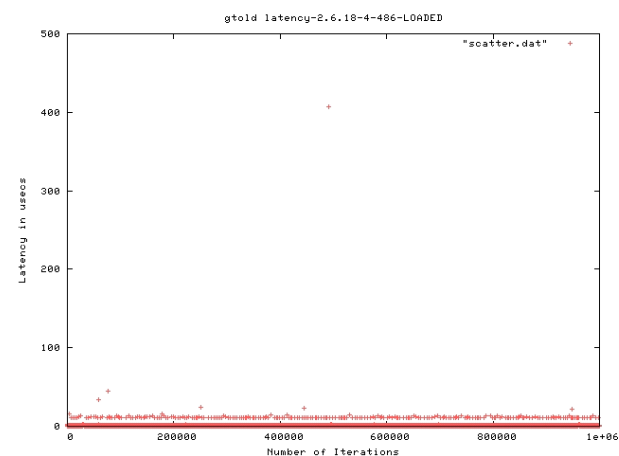


FIGURE 5: 2.6.18-1-486 LOADED

GTOD TEST: As we can see from figure 1 with the RT kernel version (2.6.26-rt1-ebx11), the maximum (red bar) latency is significantly reduced to 32 μ s even under system load. If we were to use a default Debian kernel (2.6.26-1-486), we would see max. latencies on the order of 52331 μ s (0.05 secs). Also even though the avg. latencies (green bar) is quite similar across all the kernels (around 1.1 μ s), we see significant differences with regards to max. latencies. When we talk about “hard”⁸ real-time systems we are more concerned with max. latency rather than avg. latency.

Also the 2.6.18 kernel performed better than non-RT 2.6.26 kernels. Usually we would expect the opposite of this - a recent kernel version should perform better than the older version.

4.2 CYCLICTEST Results

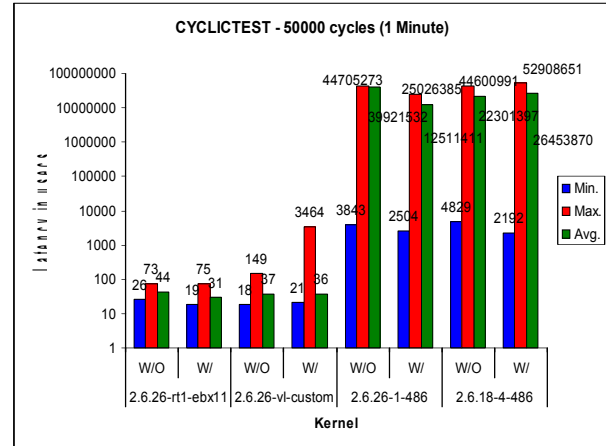


FIGURE 6: CYCLICTEST Results (1 min)

For further details on the test results please refer to Appendix III (b).

⁸For information about the difference between “hard” and “soft” real-time systems, please refer to the section “Hard vs Soft Real-time” in[2].

Histograms for CYCLICTEST (1 min) under system load⁹

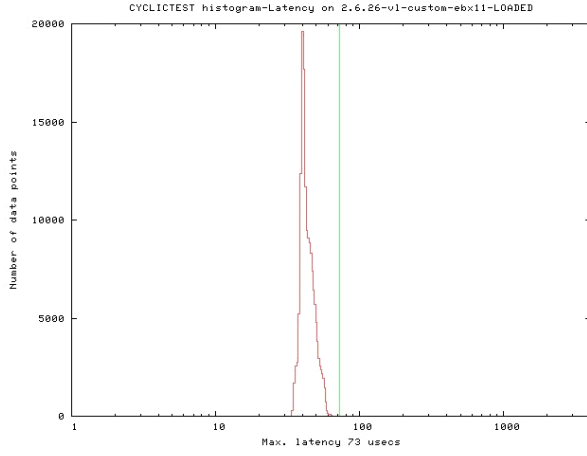


FIGURE 7: *2.6.26-rt1 LOADED*

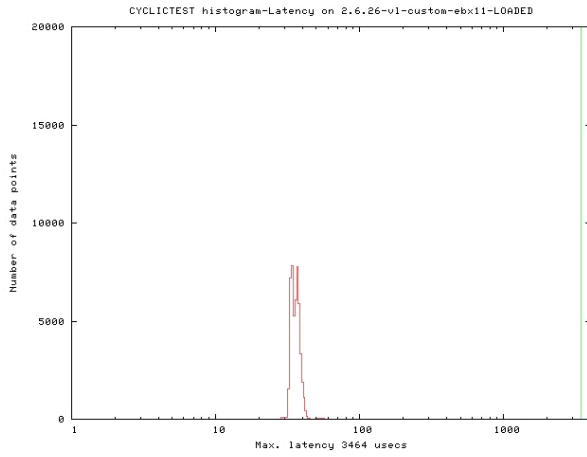


FIGURE 8: *2.6.26-custom LOADED*

Scatter plots for CYCLICTEST (1 min) under system load

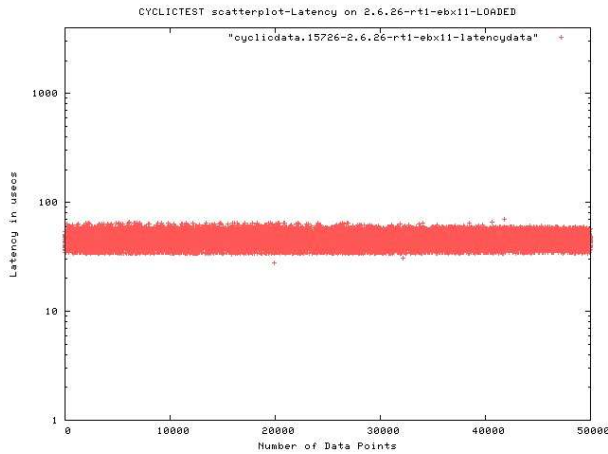


FIGURE 9: *2.6.26-rt1 LOADED*

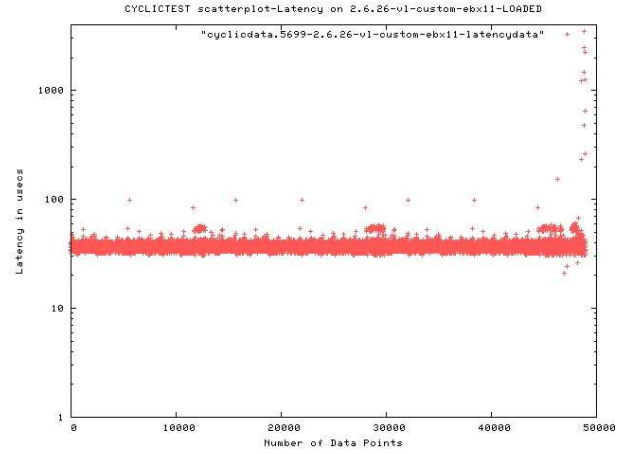


FIGURE 10: *2.6.26-custom LOADED*

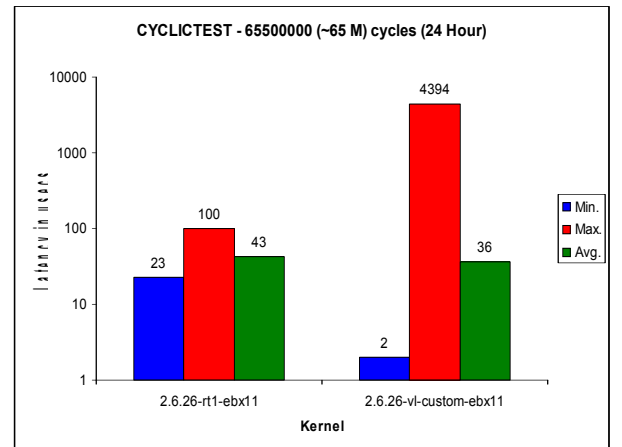


FIGURE 11: *CYCLICTEST Results (24 hr)*

For further details on the test results please refer to **Appendix III (c)**.

⁹The green line in histograms indicates the max. latency point

Histograms for CYCLICTEST (24 hour) under system load

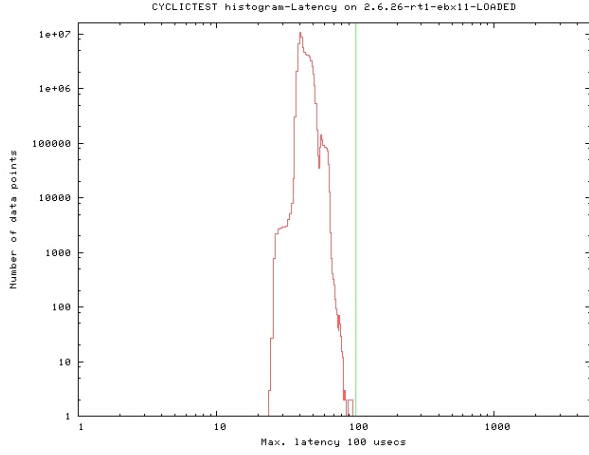


FIGURE 12: *2.6.26-rt1 LOADED*

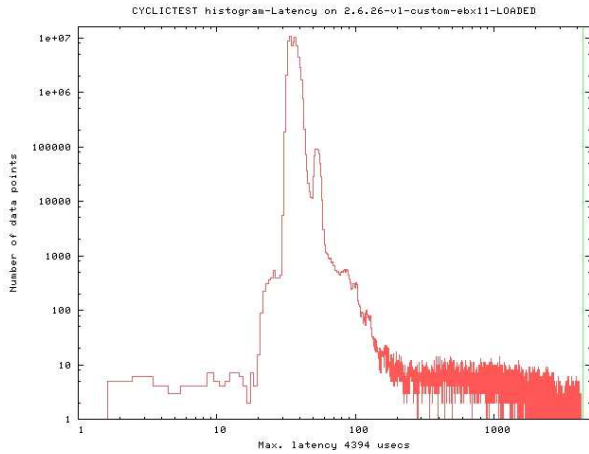


FIGURE 13: *2.6.26-custom LOADED*

Scatter plot for CYCLICTEST (24 hour) under system load

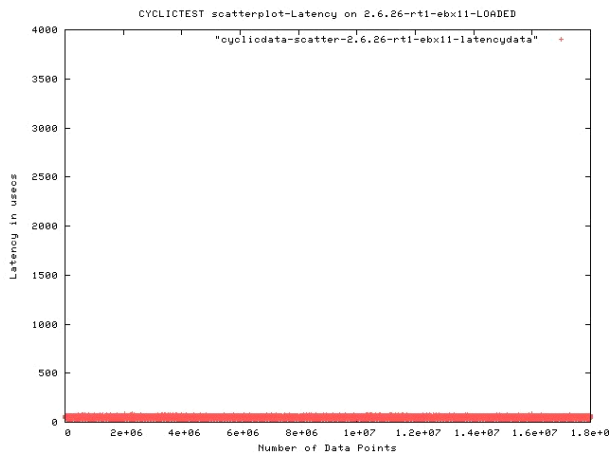


FIGURE 14: *2.6.26-rt1 LOADED*

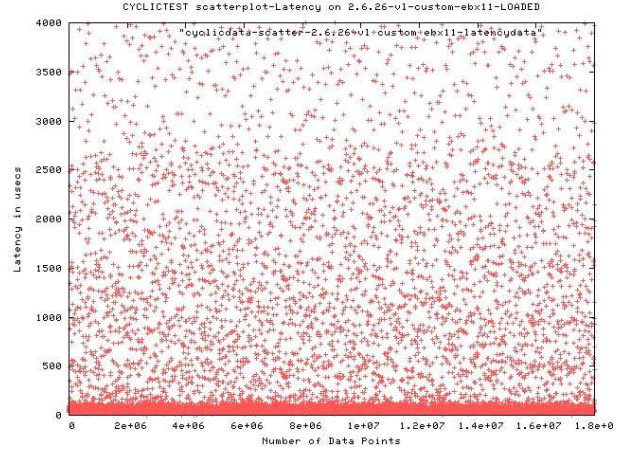


FIGURE 15: *2.6.26-custom LOADED*

CYCLICTEST: From figure 6, we can clearly see that the max. latency for RT kernel has significantly reduced to around $75 \mu\text{s}$ from $25026385 \mu\text{s}$ (25 secs). Also overall the avg. latency for RT kernel has also reduced. Also from figure 11, we can see that the max. latency for RT kernel increased to $100 \mu\text{s}$ (24 hour test) from $75 \mu\text{s}$ (1 min. test) but it is still less than max. latency of $4394 \mu\text{s}$ of the corresponding non-RT kernel (2.6.26-vl-custom-ebx11) under system load. This observation is consistent with the extended tests above - running tests for longer duration possibly makes the kernel to go to different code (or error) paths and hence we would expect increase in latencies.

Also from the above scatter plot (24 hour) figures 14,15 for cyclicttest, we can see that red dots are distributed all over the graph (bottom right) for non-RT kernel (2.6.26-vl-custom-ebx11) in contrast to RT kernel (bottom left) indicating that there are lots of instances in which latencies have shot well above $100 \mu\text{s}$ mark which is the max. latency of RT kernel (2.6.26-rt1-ebx11). One can see the nature of distribution of these latencies in the histogram plot (24 hour) also.

Furthermore, from figure 11 we can see that avg. latency of RT kernel ($23 \mu\text{s}$) is more than that of non-RT kernel ($2 \mu\text{s}$). This is quite surprising but instances like these are not uncommon[8] for people who have performed similar tests. Moreover, from a practical approach, we care more about maximum latencies rather than average latencies.

4.3 LPPTest Results

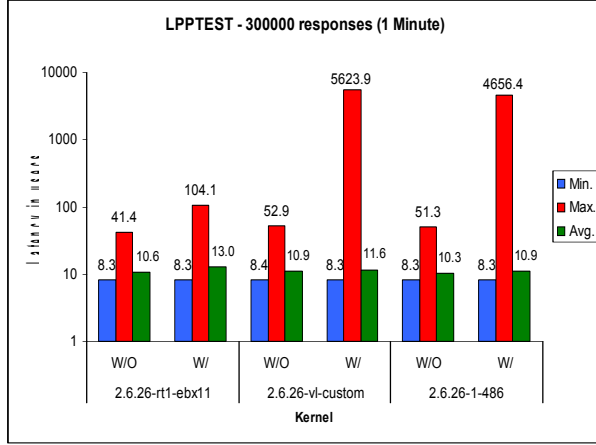


FIGURE 16: LPPTest Results (1 min)

For further details on the test results please refer to Appendix III (d)¹⁰.

Histograms for LPPTEST (1 min) under system load

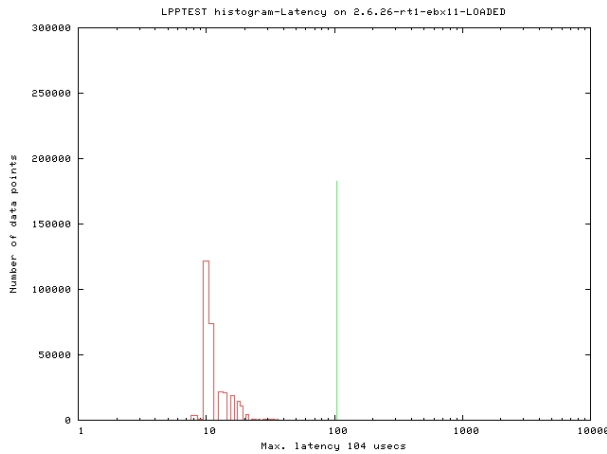


FIGURE 17: 2.6.26-rt1 LOADED

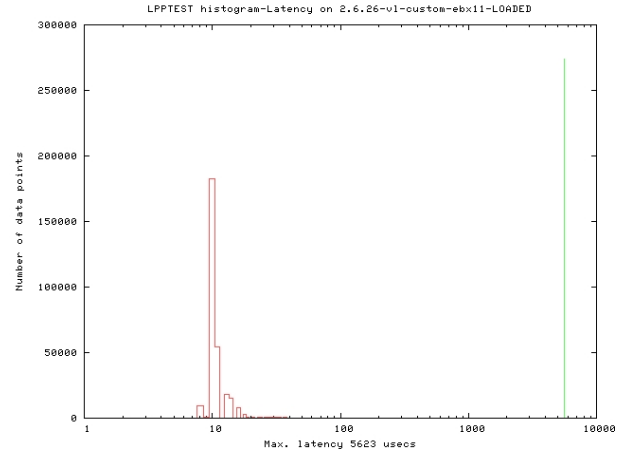


FIGURE 18: 2.6.26-custom LOADED

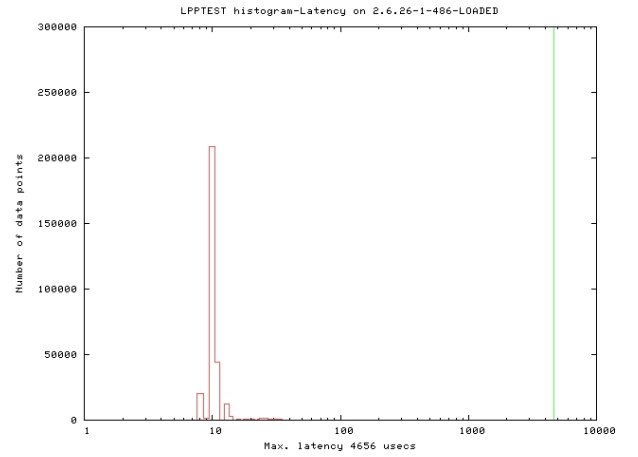
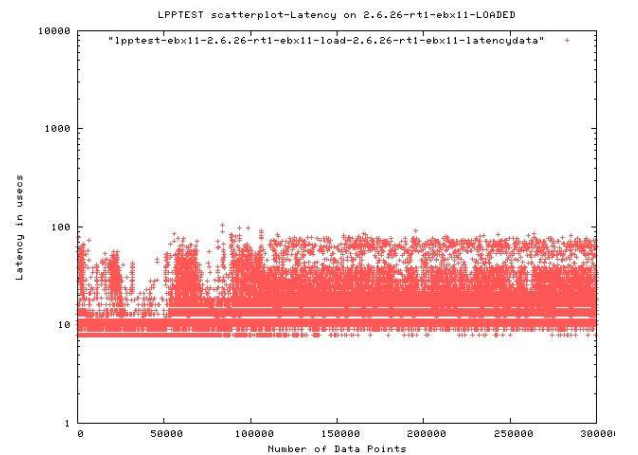
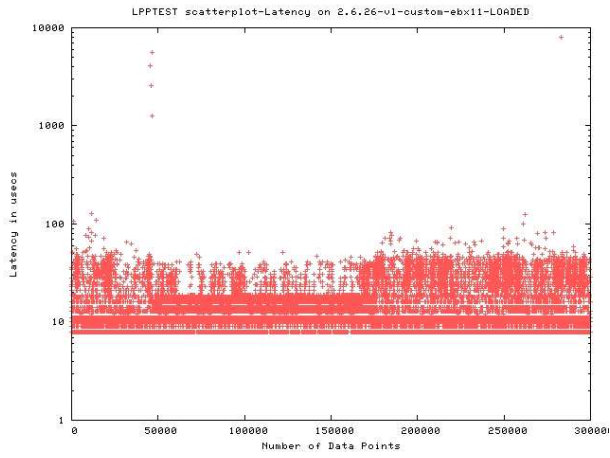
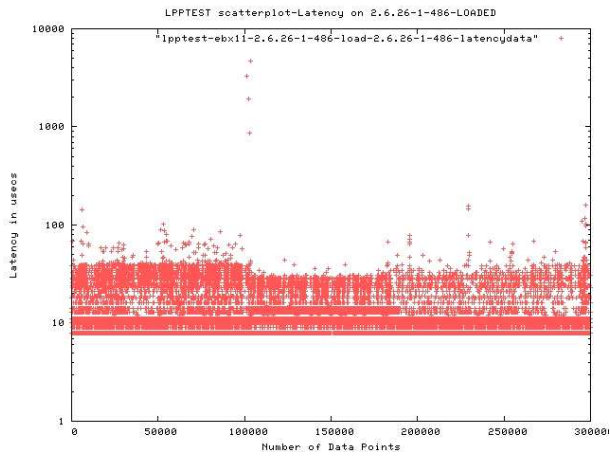
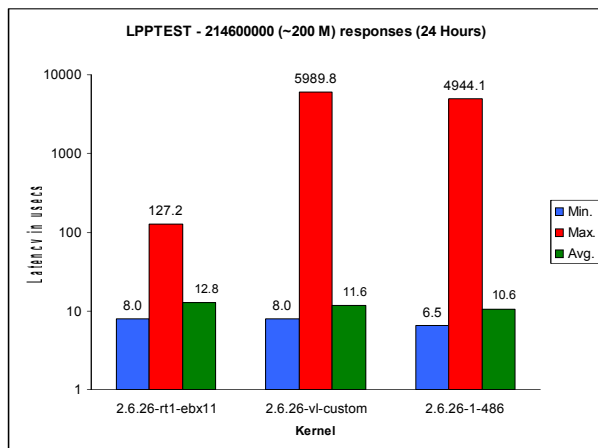


FIGURE 19: 2.6.26-1-486 LOADED

Scatter Plot for LPPTEST (1 min) under system load



¹⁰We were unable to test the 2.6.18-4-486 version under lpptest because of the difficulty in porting the lpptest program to 2.6.18 from 2.6.26 due to some major changes to the kernel code structure.

FIGURE 20: *2.6.26-rt1 LOADED*

FIGURE 21: *2.6.26-custom LOADED*

FIGURE 22: *2.6.26-1-486 LOADED*

FIGURE 23: *LPPTTest Results (24 hr)*

For further details on the test results please refer to **Appendix III (e)**.

Histograms for LPPTTEST (24 hour) under system load

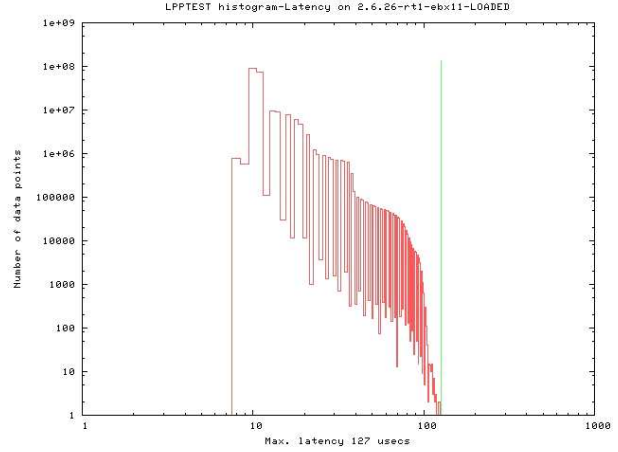
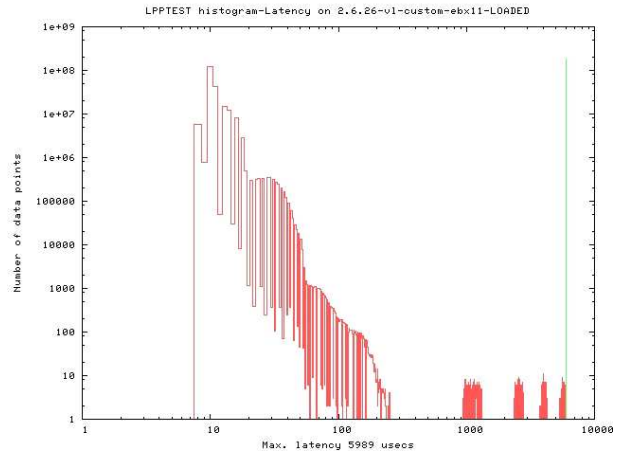
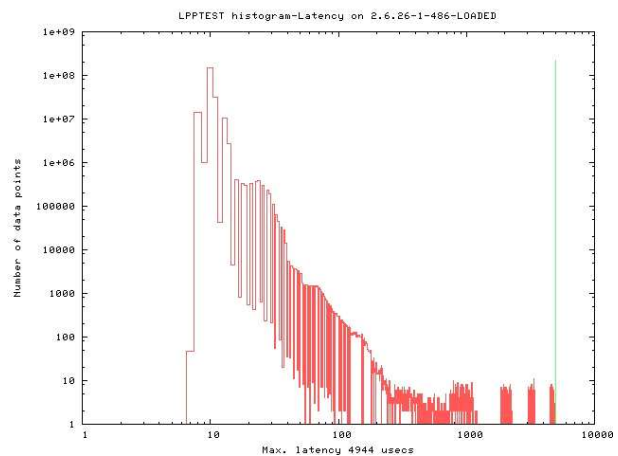
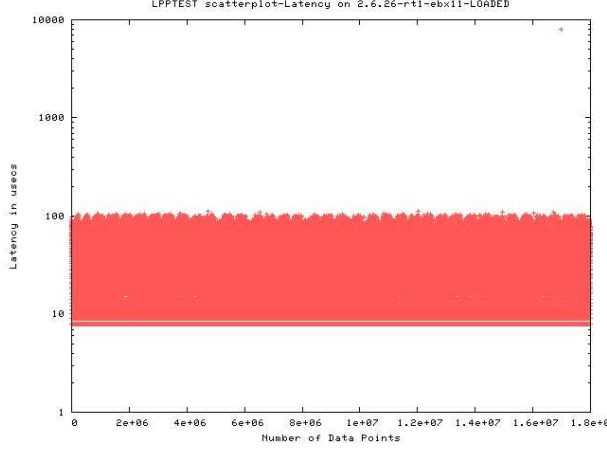
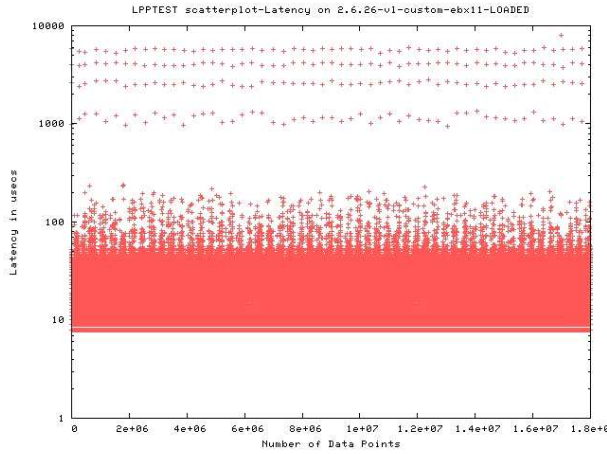
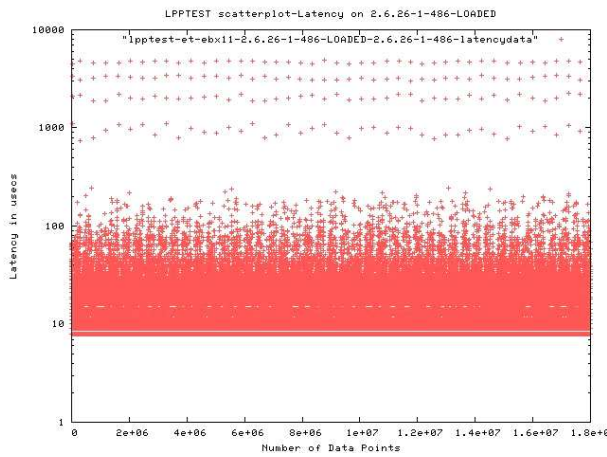

FIGURE 24: *2.6.26-rt1 LOADED*

FIGURE 25: *2.6.26-custom LOADED*


FIGURE 26: *2.6.26-1-486 LOADED*

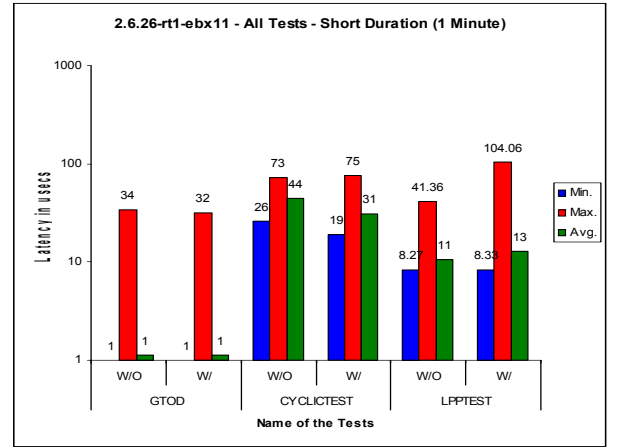
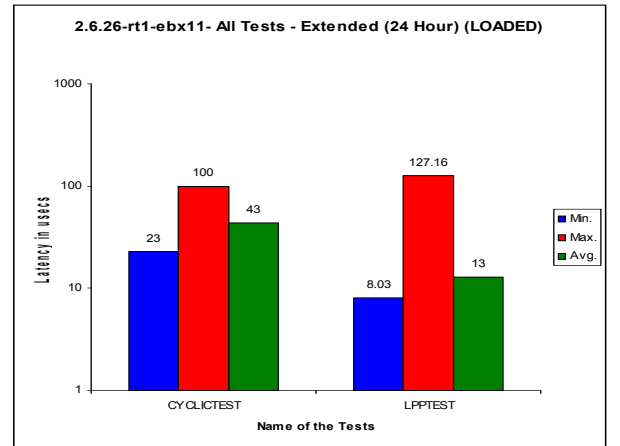
Scatter Plot for LPPTTEST (24 hour) under system load


FIGURE 27: *2.6.26-rt1 LOADED*

FIGURE 28: *2.6.26-custom LOADED*

FIGURE 29: *2.6.26-1-486 LOADED*

LPPTTEST: The results of lpptest, 1 min. and 24 hour, are consistent with results of the above two tests - max. latencies getting significantly reduced in the RT kernel as compared to non-RT kernels. See figure 4 and 5. Histograms and scatter plots (24 hour) of lpptest test show many latencies above the 127.2 μ s mark, the max. latency for the RT kernel.

4.4 Latency comparison of 2.6.26-rt1-ebx11 across all the tests

Here we compare the latencies of the RT kernel (2.6.26-rt1-ebx11) across all the tests.


FIGURE 30: *Latency Results (1 min)*

FIGURE 31: *Latency Results (24 hr)*

From the figures 30 and 31, we can see that latencies for each of the extended tests are more than their corresponding short duration tests. Also the latencies for LPPTTEST are greater than latencies of other

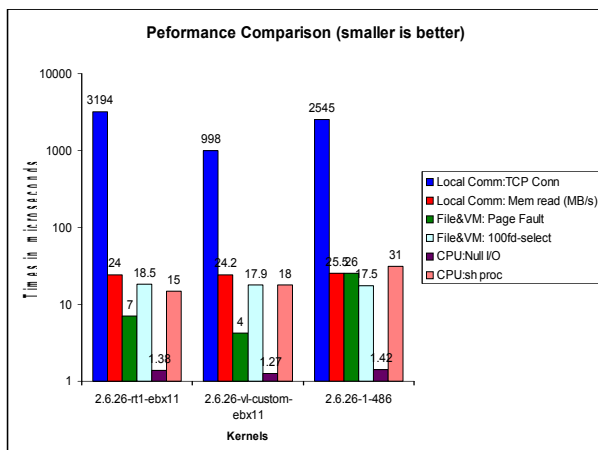
tests because of the external measurement method¹¹ involved.

Do these numbers make sense?

Now having summarized the results, what can we conclude from these numbers? Are these the right kind of numbers that we should have been expecting? Again, this is an open-ended question and the answer really depends upon your needs, the underlying application and the system[2]. However we can use the following guideline from OSADL[9] to give us some idea whether our numbers are in the ball park range: “As a rule of thumb, the maximum (worst-case) latency amounts to approximately (105 / clock frequency of an idle system).” Upon calculating based on the above guideline for the EBX-11’s 500 MHz LX processor, the maximum allowed latency comes to around 200 μ s¹². From figures 30 and 31, we can clearly see that during any of the tests, the max. latency has never exceeded more than 127.2 μ s. Therefore, based on the above guideline, the results look well under control, but then again, it really depends upon how much max. latency your application/system can tolerate.

4.5 Impact on performance

As we know[2], real-time improvements come at the cost of possible performance degradation¹³. An RTOS may (or may not) sacrifice performance in order to become more deterministic. Therefore we decided to benchmark performance of 2.6.26 kernels to see how PREEMPT_RT patch affects the performance (or throughput) in general.



¹¹We measured the latencies of the target board (EBX-11) from a host board (another EBX-11 board) using parallel port cable. For all other tests (gtod and cyclictst), the latencies are recorded on the board that is being benchmarked itself.

¹² $100000/500000000=0.0002$ secs = 200 μ s. The calculation was done assuming 1 GHz = 1 Billion cycles per second.

¹³Note that real-time does not mean better performance. For details refer to[2].

¹⁴For all parameters in the figure 32, smaller value is better except for Memread (red bar). In order to avoid creating a separate graph just for one parameter (Memread), we decided to add it with other parameters. Note that the unit of Memread is in MB/s and hence bigger value is better for Memread. For rest of the other parameters the unit is secs.

FIGURE 32: *Performance Comparison*

We arbitrarily chose few parameters to compare from various system categories such as Communication, File and Virtual Memory and CPU from the LMBench test suite[10], as shown in figure 32 above. From the above figure¹⁴, we can say the RT patch does affect the system performance negatively across most of the categories as compared to non-RT kernels, especially with TCP Conn., where real-time kernel (2.6.26-rt1-ebx11) takes 3194 μ s (tallest blue bar) and non real-time kernel (2.6.26-vl-custom-ebx11) takes only 998 μ s. In general, we can say that the negative impact of RT patch is very application dependent, although for majority of the parameters, the RT patch does not affect the system performance negatively by a “significant” margin. Please see **Appendix IV** for other system parameter results.

5 Future work and considerations

Although the above results give us a good idea about the usefulness and effectiveness of PREEMPT_RT patch, there are some additional factors to be considered:

a) BIOS/Firmware impact Real-time metrics can depend upon how well BIOS is configured and how well the BIOS code is written. Settings related to USB device configuration, SMI interrupts, System Management mode, etc. can make a huge difference on the latencies of a given system.

b) Kernel settings The above results could vary depending upon how the Linux kernel is configured. We carefully configured all relevant real-time kernel parameters to take maximum advantage of the PREEMPT_RT patch. However there is a possibility that our current kernel configuration can be tweaked to further improve the determinism and the worst-case latencies.

c) Trace large latencies To further improve deterministic behavior we can trace which kernel functions/threads/programs are responsible for causing large system latencies (in 100s of μ s). This can be done by enabling some of the debugging options in kernel[11,12]. We did not enable these debugging

options during our tests because doing so increases the system overhead and causes additional latencies to be introduced by the kernel itself, which would skew our results.

d) Different RT approach Lastly, as mentioned in the beginning of the paper, the PREEMPT_RT is just one of the several approaches to make Linux kernel real-time. It would be interesting to see if we can improve our current results by using some of the other real-time approaches like Xenomai, RTLinux, etc.

6 Conclusion

PREEMPT_RT patch significantly improves the preemptiveness of the Linux kernel. However we should keep in mind that these results are statistical in nature rather than being conclusive. Developers should consider the latency requirement of their application rather than relying on these latency numbers. For example, an embedded application developer who defines a worst-case latency requirement to be 200 μ s can probably use the above combination of EBX-11 and the RT kernel (2.6.26-rt1-ebx11). In general there is no such “magic” latency figure that one can simply assume.

There is still some debate in the community concerning the readiness of the PREEMPT_RT patch for “hard” real-time systems [14,17]. Therefore, we cannot yet draw broad conclusions about the suitability of the PREEMPT_RT patch for mission critical and life sustaining systems, where missing even a single deadline can lead to catastrophic failures and loss of life.

References

- [1] <http://www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.26-rt1.bz2>
- [2] http://versallogic.com/mediacenter/whitepapers/wp_linux_rt.asp
- [3] http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application#Latencies
- [4] http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO#Configuration_and_Compilation
- [5] <http://lkml.org/lkml/2005/6/30/9>
- [6] <http://lkml.org/lkml/2005/6/22/337>
- [7] <http://www.debian.org/CD/http-ftp/>
- [8] <http://kerneltrap.org/node/5466>
- [9] <http://www.osadl.org/Realtime-Preempt-Kernel.kernel-rt.0.html>
- [10] http://www.bitmover.com/lmbench/get_lmbench.html
- [11] *Read explanation of debugging options under “Kernel Hacking -> Kernel Debugging” option in the kernel configuration menu after applying the 2.6.26-rt1 RT patch from [1]*
- [12] <http://www.linuxdevices.com/news/NS8896249579.html>
- [13] <http://www.linuxdevices.com/articles/AT8906594941.html>
- [14] <http://dslab.lzu.edu.cn:8080/docs/publications/Siro.pdf>
- [15] <http://www.research.ibm.com/journal/sj/472/hart.html>
- [16] <http://www.uwsg.iu.edu/hypermail/linux/kernel/0507.2/0468.html>
- [17] *Building Embedded Linux Systems*, 2nd Edition, Karim Yaghmour, Jonathan Masters and Gilad Ben-Yossef, Aug. 2008, O'REILLY MEDIA

APPENDIX I – System Load Script Code

```
#!/bin/bash
defaultseconds=60
seconds="$1"
if test -z "$seconds" || test "$seconds" -lt 1
then
seconds=$defaultseconds
fi
cd /usr/src/linux
make clean
make -j8 &
cd;
echo "Started make -j8"
(while true; do dd if=/dev/zero of=/dev/null bs=1024000 count=1000; done) &
echo "Started DD"
ping -l 100 -q -s 10 -f localhost &
echo "Started ping flood"
(while true; do nice ./hackbench 30 >/dev/null; done) &
echo "Started hackbench..."
(while true; do du / 1>/dev/null 2>&1; done) &
echo "Started DU..."
while test $seconds -gt 0
do
echo -n -e "\rStill $seconds seconds to wait ..."
sleep 1
seconds='expr $seconds - 1'
done
echo
echo "Script Terminated..."
```

APPENDIX II – Tests Comparison

Test Name	GTOD_LATENCY	CYCLICTEST	LPPTTEST
Interrupt Generation	N/A	Clock clock_nanosleep	Data sent on Place- NameplaceParallel PlaceTypePort
Measures?	Time between pairs of consecutive calls to determine time	Interrupt and Scheduling Latency	End-to-End Re- sponse Latency
CityplaceNormal Iterations (1 Min. approx.)	1000000	50000	300000
WC¹ Iterations (1 Min. approx.)	1000000	50000	300000
Extended Iteration (24 Hr. approx.)	N/A ²	65500000 (65.5 M ³)	214600000 (214.6 M)
Driver/ Function	Gettimeofday()	N/A	lpptest
IRQ	N/A	N/A	7
Person/ Company	IBM	Thomas Gleixner TimeSys	Ingo Molar Red Hat
Comments	gtod = get_time_of_day. The test calls a func- tion to get system's time of day.	Uses High Resolu- tion timer support. Measures expected time vs. actual time for wakeup.	Built inside kernel. External measure- ment.

¹WC = Worst-case scenario²The test fails above 10000000 iterations and hence there are no worst-case results this test.³M = Million

APPENDIX III – Test results

a) GetTimeOfDay - gtod_latency (usecs) – 1000000 Iterations

Kernels	2.6.26-rt1-ebx11		2.6.26-vl-custom		2.6.26-1-486		2.6.18-4-486	
Load?	W/O	W/	W/O	W/	W/O	W/	W/O	W/
Min.	1	1	1	1	1	1	N/A	N/A
Max.	34	32	68	52075	418	52331	406	407
Avg.	1.1268	1.1422	1.1101	1.1621	1.1311	1.1817	0.9869	0.9951
Std Dev	0.5845	0.5720	0.4786	52.0764	0.7188	52.3333	0.4590	0.4509

b) CyclicTest (Interrupt and scheduling latency) (usecs) – 50000 cycles (1 Minute)

Kernels	2.6.26-rt1-ebx11		2.6.26-vl-custom		2.6.26-1-486		2.6.18-4-486	
Load?	W/O	W/	W/O	W/	W/O	W/	W/O	W/
Min.	26	19	18	21	3843	2504	4829	2192
Max.	73	75	149	3464	44705273	25026385	44600991	52908651
Avg.	44	31	37.47	36.33	39921532	12511411	22301397	26453870

c) CyclicTest (Interrupt and scheduling latency) (usecs) – 18000000 cycles (24 hour)

Kernels	2.6.26-rt1-ebx11	2.6.26-vl-custom
Load?	Yes	Yes
Min.	23	2
Max.	100	4394
Avg.	43.29363198	36.5
Samples	65512035	65543590

d) LPP Test (End to End response latency) (usecs) – 300000 responses (1 Minute)

Kernels	2.6.26-rt1-ebx11		2.6.26-vl-custom		2.6.26-1-486		2.6.18-4-486	
Load?	W/O	W/	W/O	W/	W/O	W/	W/O	W/
Min.	8.27	8.33	8.36	8.33	8.33	8.33	N/A	N/A
Max.	41.36	104.06	52.91	5623.92	51.26	4656.44	N/A	N/A
Avg.	10.59	12.98	10.90	11.58	10.33	10.94	N/A	N/A

e) LPP Test (End to End response latency) (usecs) – 214600000 responses (24 hours)

Kernels	2.6.26-rt1-ebx11	2.6.26-vl-custom	2.6.26-1-486
Min.	8.03	7.95	6.45
Max.	127.16	5989.77	4944.08
Avg.	12.79	11.59	10.63
Responses	214620000	214669636	214619724

APPENDIX IV

L M B E N C H 3 . 0 S U M M A R Y

(Alpha software, do not distribute)

Basic system parameters

Host	OS Description	Mhz	tlb pages	cache line bytes	mem par	scal load
ebx11	Linux 2.6.26-1-486	495		32		1
ebx11	Linux 2.6.26-vl-custom	494		32		1
ebx11	Linux 2.6.26-rt1-ebx11	493		32		1

Processor, Processes - times in microseconds - smaller is better

Host	OS	Mhz	null call	null I/O	stat	open clos	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
ebx11	Linux 2.6.26-	495	0.55	1.42	7.34	10.7	32.3	2.17	5.63	34.K	49.K	31.K
ebx11	Linux 2.6.26-	494	0.74	1.27	7.67	11.0	32.0	2.13	5.82	383.	4156	18.K
ebx11	Linux 2.6.26-	493	0.89	1.38	9.51	15.8	32.3	2.23	6.63	687.	2816	15.K

Context switching - times in microseconds - smaller is better

Host	OS	2p/0K ctxsw	2p/16K ctxsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctxsw	16p/16K ctxsw	16p/64K ctxsw
ebx11	Linux 2.6.26-	65.9	221.5	723.0		6.3800		
ebx11	Linux 2.6.26-	230.6	233.5	797.8				

Local Communication latencies in microseconds - smaller is better

Host	OS	2p/0K ctxsw	Pipe	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn
ebx11	Linux 2.6.26-		36.4	35.1	110.8		378.7		2545
ebx11	Linux 2.6.26-	65.9	49.2	107.	133.3	345.8	367.1		998.
ebx11	Linux 2.6.26-	230.6	117.1	173.	167.4	925.8	347.0		3194

File & VM system latencies in microseconds - smaller is better

Host	OS	0K File Create	10K File Delete	10K File Create	10K File Delete	Mmap Latency	Prot Fault	Page Fault	100fd selct
ebx11	Linux 2.6.26-					71.3K	0.985	25.5	17.5
ebx11	Linux 2.6.26-					20.7K	0.765	4.27620	17.9
ebx11	Linux 2.6.26-					33.2K		7.13720	18.5

Local Communication bandwidths in MB/s - bigger is better

Host	OS	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
ebx11	Linux 2.6.26-	17.6	19.5	15.3	13.1	24.5	20.7	20.0	25.5	82.8
ebx11	Linux 2.6.26-	15.9	16.8	11.9	12.4	24.4	19.8	19.1	24.2	83.9
ebx11	Linux 2.6.26-	15.1	16.2	1.46	9.2500	23.7	19.5	18.6	24.0	81.7

Timing Analysis of Linux CAN Drivers

Michal Sojka, Pavel Píša

Czech Technical University in Prague, Faculty of Electrical Engineering
Technická 2, 166 27 Prague, Czech Republic
sojkam1@fel.cvut.cz, pisa@cmp.felk.cvut.cz

Abstract

Controller-Area Network (CAN) is a communication bus widely used in industrial and automotive systems. There exists many CAN drivers for Linux and one of them – Socketcan – is being merged to the mainline kernel. Another often used driver is called LinCAN. The drivers differ in the internal architecture and in this paper we provide the results of several experiments that compare latencies introduced by this two drivers. The aim is to evaluate suitability of these drivers for use in real-time applications.

1 Introduction

CAN-bus (Controller-Area Network) is a communication bus widely used in industrial and automotive systems. Not only that it simplifies wirings, but CAN-bus offers deterministic medium access algorithm for which it is very suitable for certain type of time-critical applications. A typical CAN-based network consists of multiple microcontrollers and one or a few more complex nodes which implement higher-level control algorithms. These more complex nodes can run Linux OS so there is a need for CAN-bus drivers. There exist several projects which offer CAN drivers for Linux. Probably the most known driver is Socketcan, which is already included in the mainline kernel. Another option is LinCAN driver. The basic difference between these two drivers is that Socketcan is built around standard Linux networking infrastructure, whereas LinCAN is a character device driver with its own queuing infrastructure designed for specific needs of CAN communication.

The Linux networking infrastructure is designed with the goal of achieving highest possible throughput for IP and similar high-bandwidth traffic, which is almost opposite to the goals of CAN bus – low bandwidth (CAN messages can hold up to 8 bytes of data payload) but precise timing. One can naturally ask whether and how Socketcan's usage of Linux networking layer influences communication latencies.

We have developed a tool for measuring round-trip times (RTT) on CAN-bus and we have conducted several experiments to evaluate RTT under different system loads and other conditions with the two above mentioned drivers.

The results of this work are useful for the following reasons:

1. Users can approximately know timing properties of a particular CAN-on-Linux solutions.
2. Developed tools can be used for benchmarking other hardware to find suitability for intended applications.
3. Driver developers can profile and improve their drivers and find factors contributing to worst-case delays in RX/TX paths.
4. Overhead caused by the use of generic Linux networking infrastructure in Socket-CAN implementation was quantified. Our finding is that the difference is not negligible, but it is still suitable for many real-time applications.

1.1 CAN description

CAN is a serial bus introduced in 1986 [1] by Robert Bosch GmbH. It uses a multi-master protocol based on a non-destructive arbitration mechanism, which

grants bus access to the message with the highest priority without any delays. The message priority is determined by the message identifier, which identifies the content of the message rather than the address of the transmitter or the receiver.

Physical layer has to support transmission of two states – dominant and recessive. If no node transmits dominant state, the medium should remain in recessive state. Whenever any node transmits dominant state, the medium should be in dominant state. This property (sometimes called “wired OR”) is used by the medium access control (MAC) layer to accomplish the non-destructive arbitration mechanism. All nodes transmit bits synchronously and during transmission of the message arbitration phase the node checks for a difference between transmitted and medium states. If a difference is found, the node backs off. Since the arbitration requires synchronous bit transmission and the signal propagation speed is finite, the length of the bus is limited. For 125 kbit/s the maximum bus length is 500 m, for the maximum speed 1 Mbit/s, it is 30 m.

The message identifier, which is transmitted in message arbitration phase is either 11 or 29 bit long depending on whether extended message format is used. The length of the data payload can be up to 8 bytes and every message is protected by 15 bit CRC. Every message is acknowledged by an ACK bit. Transmitter transmits a receive bit and all receivers with correctly received message transmit that bit as dominant. This way, if no receiver is able to receive the message correctly, the transmitter can signal error to the upper layers.

2 Internal architecture of the drivers

In this section we briefly describe internal architecture of the tested drivers, especially the TX and RX paths, which mostly influence the time needed to process transmitted and received messages.

2.1 LinCAN

LinCAN driver presents CAN devices as character devices `/dev/canX`. Applications use the driver through the standard UN*X interface represented by the five functions: `open()`, `read()`, `write()`, `ioctl()` and `close()`. The heart of the driver is a flexible queuing infrastructure [2], which allows the driver to be used simultaneously from userspace, IRQ handlers and RTLinux programs. For every user of

the driver (open file descriptor in userspace or an RTLinux application), there one queue for message transmission and one for reception.

The queue has preallocated a fixed number of slots to hold received messages or messages waiting for transmission. By default every queue can hold up to 64 messages. The queuing infrastructure does not support message ordering according to their priority, but there is (currently unused) support for having multiple queues for different priorities.

LinCAN is distributed as a out-of-tree driver and there are no plans to push it to mainline.

2.1.1 Transmit path

To transmit a message, the application calls `write()` system call on a `/dev/canX` file descriptor. This operation executes `can_write()` which stores the message in the queue and wakes the chip driver to send the messages in the queue. The driver simply sends out the message from the queue. If the chip is busy by sending the previous messages, nothing happens and the queue is processed later when TX completion IRQ arrives.

2.1.2 Receive path

Interrupt handler reads the message and copy it directly from the IRQ context to all the queues connected to the particular device (`canque_filter_msg2edges()`). Then any readers waiting in `read()` system call are woken up.

2.2 Socketcan

Socketcan is a driver built around Linux networking infrastructure. As of 2.6.31, most of it is already merged into the mainline. Besides the drivers themselves, Socketcan implements several higher-level protocols (BCM, ISO-TP and raw) and provides useful userspace utilities. Applications access the socketcan functionality through BSD sockets API which is commonly used for network programming not only in the UN*X world.

2.2.1 Transmit path

After the message is sent through `sendmsg()` or similar system call, it is passed to `can_send()`, which queues it in the device queue by calling `dev_queue_xmit()`. This function uses a queuing discipline (qdisc) to queue the message in the driver

TX queue. The default qdisc is `pfifo_fast` with the queue length of 10. When a message is enqueued `qdisc_run()` is called to process the queue. If the driver is not busy transmitting another message, it transmits it immediately, otherwise the qdisc's queue is appended (see `__netif_reschedule()`) to a per-CPU queue and processed later in `NET_TX_SOFTIRQ`.

2.2.2 Receive path

Interrupt handler reads the message¹ and calls `netif_rx()` which queues the message in per CPU `softnet_data` queue and schedules `NET_RX_SOFTIRQ` for later processing. It is important to know that this queue is shared for all kinds of network traffic, not only for CAN.

The softirq runs `net_rx_action()` which polls the `softnet_data` queue in `process_backlog()`, and calls `netif_receive_skb()` for every message there. Note there is also rate limiting in this softirq, so if there are many packets from other sources, our CAN messages are delayed by additional delays.

Within `netif_receive_skb()` the list of registered packet types is traversed and the message is passed to the handler, which, in case of socketcan, is `can_rcv()`.

`can_rcv()` traverses the list of sockets and delivers the message to all sockets interested in receiving the message. For raw sockets this is accomplished by calling `raw_rcv()` which clones the socket buffer and calls `sock_queue_rcv_skb()`, which copies the data to the socket receive queue by `skb_queue_tail()` and userspace is woken up in `sock_def_readable()`.

3 Testing

As one can see from the description of the driver internals in the previous section, TX and especially RX path is considerably more complicated in case of Socketcan. Given the fact, that CAN message is up to 8 byte in length, maximum bitrate is 1 Mbit/s and deterministic timing is often required, the question which we try to answer in this paper is whether the Socketcan overhead is not too high for serious use.

¹This is true for `sja1000.c` driver. Other drivers, such as `mcan.c` read the messages entirely in softirq.

²<http://rttime.felk.cvut.cz/gitweb/canping.git>

³<http://ocera.cvs.sourceforge.net/viewvc/ocera/ocera/components/comm/can/canvca/libvca/>

3.1 Testbed description

To answer the question, we have set up a testbed to compare the drivers under the same conditions. The testbed was a Pentium 4 box with a Kvaser PCican-Q (quad-head PCI card based on SJA1000 CAN controller). The CPU is a single core and supports hyper-threading (HT).

In most experiments, we measured the round-trip time (RTT) i.e. the time it takes to send the message to the other end and back. To measure the RTT we used *canping* tool². It send the messages according to command line switches and measures the time elapsed until receiving a response. The next message is sent after a response to the previous one is received. The measured times are statistically processed and histograms can be generated from them. To access the CAN driver, canping uses VCA (Virtual CAN API) library³, which provides common API for different driver back-ends.

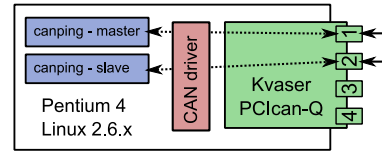


FIGURE 1: Testbed setup

We simply connected CAN card's output 1 with output 2 (see Fig. 1) and run two instances of canping on one computer – one in slave mode to only respond to messages generated by the second canping in master mode. Both instances was run with real-time priority and have locked all their memory (`mlockall()`).

In every experiment, we sent ten thousands messages. We have repeated all the experiments with several different kernels. For real-time application it is natural to measure the performance under rt-preempt kernels, but for other use, results for non-rt kernels can also be useful.

For all testes CAN baudrate was set to 1 Mbit/s and all messages carried 8 fixed bytes of data payload. The length of messages was measured on oscilloscope to be $112\mu s$, but every message is followed by 7 recessive end of frame bits and 3 bits of inter frame space. Therefore the shortest possible bus time needed for sending the message forth and back is $2 * (112 + 7) + 3 = 241\mu s$.

The computer didn't run unnecessary services such as cron. Also irqbalance daemon was not running so that the interrupts went always to the first CPU (hyper-threading). Since there were some differences in results depending on whether hyper-threading was used or not, we also run the tests on kernels with `maxcpus=1` command line parameter. Such test are marked by `:maxcpus=1` suffix to kernel version.

All tests were run at full CPU clock speed, which was 2.4 GHz, as well as at the lowest one (300 MHz) by which we try to simulate an embedded system, where CAN is used a lot.

All tests used socketcan from trunk at SVN revision 1009. The exception is the kernel 2.6.31-rc7, for which the version in mainline was used. For LinCAN we used a version from CVS dated 2009-06-11.

3.2 Results

To graph the results, we use so called latency profiles. It is a cumulative histogram with reversed vertical axis displayed in logarithmic scale. This way we can see the exact number of packets with worst latencies at the bottom right part of the graph. You can see it in Figure 2, where we measured RTT on an unloaded system. It can be seen that the LinCAN has lower overhead than Socketcan, which was an expected result. Another observation is that worst-case driver overhead is approximately the same as message transmission time. For LinCAN the overhead is $450 - 241 = 209 \mu\text{s}$, for Socketcan $525 - 241 = 284 \mu\text{s}$.

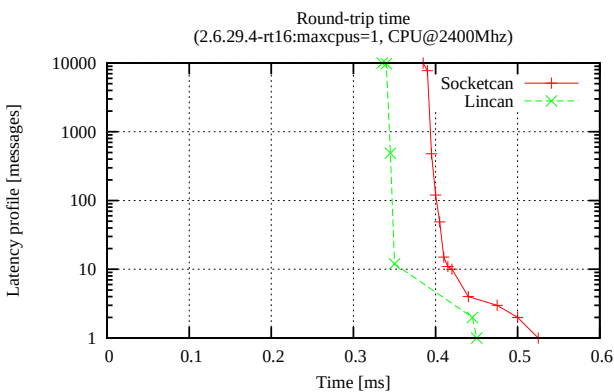


FIGURE 2: *Round-Trip Time on the unloaded system, 2.4 GHz*

If the CPU frequency is lowered to 300 MHz, then the overhead is much larger for both LinCAN and Socketcan (Fig. 3).

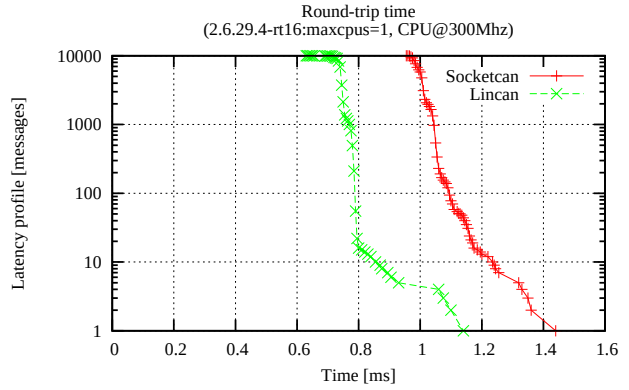


FIGURE 3: *Round-Trip Time on the unloaded system, 300 MHz*

Since Socketcan shares the RX path with the other parts of Linux networking stack, in the next experiment we tried to measure how other network traffic influences CAN latency. We generated Ethernet load to the CAN testbed from another computer. The used Ethernet card was Intel's e100 (100 Mbps). We tried the following load types: flood ping, flood ping with 64 kB datagrams and TCP load (high bandwidth SSH traffic). It turned out that the worst influence was caused by flood ping with 64 kB datagrams (see Fig. 4). With default system configuration both drivers exhibit additional latency of approximately 300 μs .

However, under `-rt` kernels, most interrupt handlers run in thread context and it is possible to set the priority of those IRQ threads. Therefore we increased the priority of the IRQ thread handling our CAN card interrupts (the interrupt line was not shared with any other peripheral). The results can be also seen in Fig. 4 – lines marked as “boosted IRQ prio”. As can be seen, LinCAN latency goes down to the same values as without any load. Socketcan is not influenced by this tuning at all.

Socketcan has another parameter which can be used to tune packet reception – `netdev_budget` sysctl. By this knob, one can tune how many received packets is processed in one run of `NET_RX_SOFTIRQ`. The default value of this parameter is 300 and as can be seen again in Fig. 4, changing this value has no effect on the Socketcan latency. Further investigation showed that with our tests there were rarely more than 10 pending packets and since softirqs are re-executed up to 10 times if they are pending at the time they finish execution, even lowering budget to 1 had no effect.

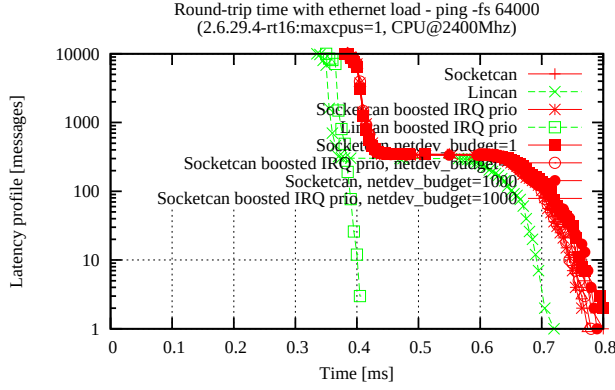


FIGURE 4: *Round-Trip Time with Ethernet load*

For the sake of completeness, we also tested whether Ethernet transmission influences CAN latency. As can be seen from Fig. 5, there is almost no influence by this type of load.

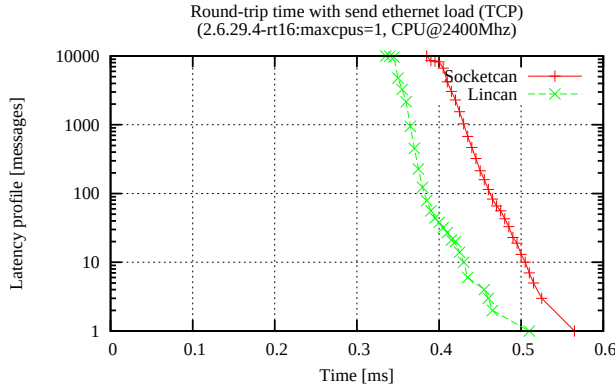


FIGURE 5: *Round-Trip Time with Ethernet transmission*

Also, there is almost no influence by non-realtime CPU load, which was generated by `hackbench` tool⁴ (Fig. 6).

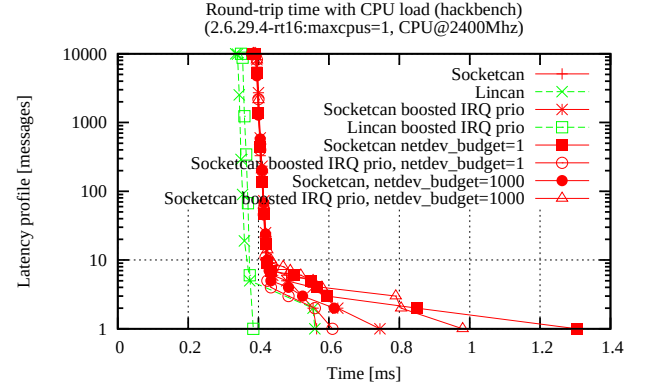


FIGURE 6: *Round-Trip Time with CPU loaded by `hackbench` -pipe 20*

While testing the drivers on non-rt kernels, the worst-case latency was usually very high as can be seen in Fig. 7. In this particular experiment, Socketcan experienced one 22 ms delay, but in other experiments this happened even to LinCAN. We didn't investigate the source of this latency in detail, but it was probably caused by `ath5k` driver because after unloading this driver, this latency "peaks" disappeared.

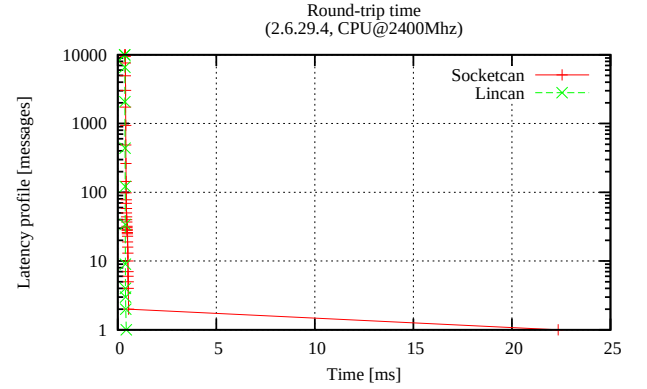


FIGURE 7: *Round-Trip Time on the unloaded system, 2.4 GHz, non-rt kernel*

When we unloaded all unnecessary modules from the kernel, the latencies went down for LinCAN (see Fig. 8). The reason is that LinCAN does all the RX processing in hard-IRQ context, whereas Socketcan employs a soft-IRQ which is shared by all network devices.

⁴<http://devresources.linux-foundation.org/craiger/hackbench/>

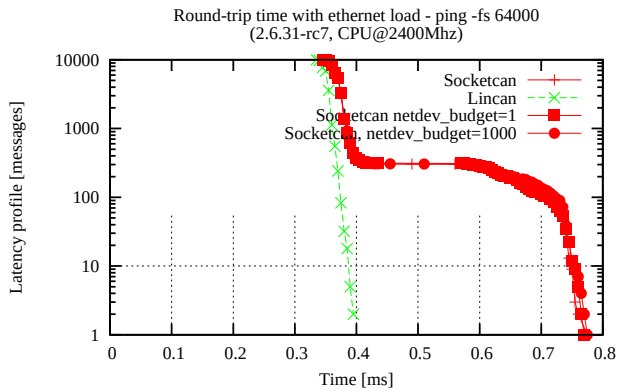


FIGURE 8: Round-Trip Time with Ethernet load, 2.4 GHz, non-rt kernel, no unnecessary modules

Another interesting results result is performance of virtual CAN drivers. This driver can be used to test CAN applications on places where real CAN hardware is not available. It seems there is some problem with how Socketcan implements this. Under Debian kernel 2.6.26-2-686, which is configured with `PREEMPT_NONE=y`, Socketcan latencies raised up to 550 ms (Fig. 9).

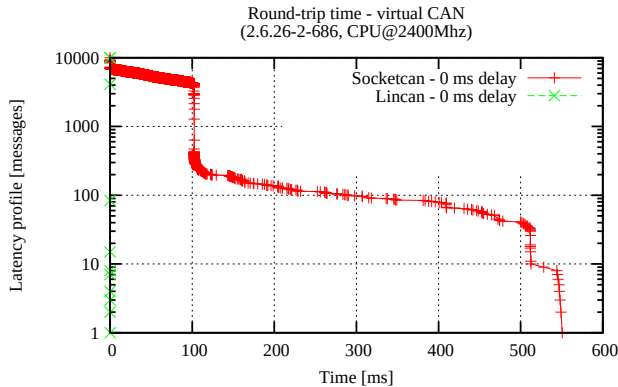


FIGURE 9: Round-Trip Time on virtual CAN driver

On kernel 2.6.31-rc7, the results seem also strange for Socketcan (Fig. 10). For all other kernels the results were as expected – worst-case latency around several hundreds microseconds.

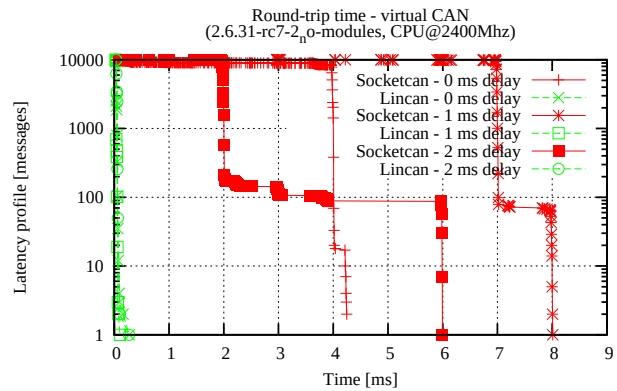


FIGURE 10: Round-Trip Time on virtual CAN driver

Last but not least, people are continuously improving Linux network stack and under kernel 2.6.31-rc7, the performance of Socketcan is only slightly worse than that of Lincan (see Fig. 11).

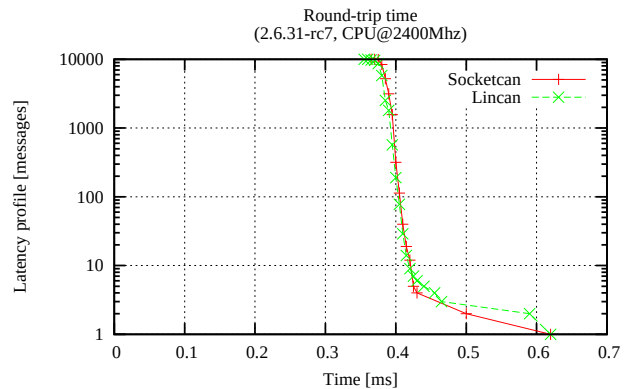


FIGURE 11: Round-Trip Time on recent kernel and unloaded system

The complete results of all tests for several different kernels can be found on our web page⁵. If somebody would like to reproduce our results all the code, configurations etc. is available from Git repository⁶.

4 Conclusion

The experiments described in this paper show the differences between two distinct approaches to writing Linux CAN driver – character device and network device. In most cases the character device approach

⁵<http://rtime.felk.cvut.cz/can/benchmark/1/>

⁶<http://rtime.felk.cvut.cz/gitweb/can-benchmark.git>

has lower overhead but it seems that recent improvements in Linux kernel decrease the overhead of the network device approach.

One area where character based drivers still win is when the system is loaded by receiving non-CAN traffic. Under preempt-rt kernels, one can increase the priority of the IRQ thread which handles the CAN hardware and latencies go down with character device approach, whereas network device approach exhibits higher latencies regardless of IRQ thread priority. When compared to the case with without non-CAN traffic, the worst-case latency is approximately doubled. This values can be still suitable for most real-time applications. If this latency is not acceptable, one would need to split the queue between hard- and soft-IRQ to several queues of different priorities with CAN messages going to the high-priority

queue and all other traffic to queues with lower priorities.

Finally, we found that timing of Socketcan's `vcan` driver is quite odd under some kernels. It seems, it has some connection to CPU scheduling and it needs to be further investigated and fixed.

References

- [1] CiA, "CAN history," 2001. [Online]. Available: <http://www.can-cia.org/index.php?id=161>
- [2] P. Píša, "Linux/RT-Linux CAN driver (LinCAN)," 2005. [Online]. Available: <http://cmp.felk.cvut.cz/pisa/can/doc/lincandoc-0.3.pdf>

Efficient embedded software development using QEMU

Pradyumna Sampath

ABB Corporate Research
Bhoruka Tech Park, 5th Floor, Block 1,
Whitefield Road, Mahadevapura,
Bangalore, Karnataka, India
pradyumna.sampath@in.abb.com

Rachana Rao

ABB Corporate Research
Bhoruka Tech Park, 5th Floor, Block 1,
Whitefield Road, Mahadevapura,
Bangalore, Karnataka, India
rachana.rao@in.abb.com

Abstract

Software based processor emulators have been around for a long time and predominantly used by developers of Operating Systems. With virtualization technology becoming more common and gaining in-processor support, these emulators have expanded to support several processor architectures and a richer feature set. Consequently, popular operating systems such as Linux have been adapted to run within virtual environments seamlessly. As part of the software development process, this virtualization technology has huge potential to increase product quality and developer efficiency.

This paper looks at employing virtualization for in-house real-time application software development through QEMU, a popular open source emulator. It also details the process of establishing such an environment and looks at some of the advantages of such a development model for embedded systems.

1 Introduction

Industrial Automation and Power technologies are characterized by extremely complex software systems which are in many cases very tightly coupled with mechanical and electrical sub-systems. These systems tend to be safety and mission critical. Development of such systems is considerably time consuming. However, they are also characterized by long if not very long product lifecycles sometimes in the order of fifteen years. The scope and complexity of software components in these industrial products has gone up significantly over the years. Continuous remodeling and development of software for the enduring industrial automation systems, places considerable challenges of efficiency on the development environment and processes followed. This paper attempts to look at acceleration of embedded software

development by decoupling it from the underlying hardware.

Traditionally, embedded software development is inherently dependent on hardware availability. Hardware development includes design, simulation and testing of hardware architecture, logic, circuit schematics and finally the PCB. This is a very time-consuming process, often iterative and unpredictable due to dependency on factors like component availability and vendor support. Software developers usually cannot afford to wait for the entire hardware design to be completed and the board to arrive on their desks before starting on application development. This would increase the product development time substantially which is unacceptable. Application development as well as verification(testing) should be well underway by the time the hardware

is finalized and delivered to the software developers. This is where system emulation and virtual environments step in and save the day. With the various advanced hardware emulators and virtual machines available today, embedded software developers can get a head start on their applications.

The ability to develop applications for a hardware without the physical device itself is a definitely a big advantage. However, there are other advantages provided by the emulator too. It can make a safe and secure environment for the testing of new and untried applications. Since modern emulators now support various kinds of hardware as well as different operating systems, the application can also be tested for scalability and reliability and not just functionality. This makes virtualized environments very powerful and feasible testing platforms. Emulators also prove valuable in cases of kernel and device driver development, where a small mistake can crash the entire operating system. Developing and debugging drivers on an emulator, makes it similar to user-space applications, which at the worst can lead to the emulator crash.

2 Virtualization and Emulators

Since the term virtualization[1] was coined, over four decades ago, it has been defined many times and in many different ways. Virtualization is basically abstraction of one or more computer resources to achieve the behaviour of the desired system. The desired system here could be a high end grid computer or a testing platform for a different architecture etc. In the beginning, the term emulation was usually used for virtualization using hardware. However, recently it has also become common to use the term emulation in the software context. Emulation[2] refers to the replication of functions of a system by another, so that the emulator acts similar to the emulated system.

There are various types of virtualization[3][4], depending on the system being virtualized and levels of abstraction used to achieve it. Hardware emulation refers to the virtualization of the required hardware on the host machine. It usually suffers from being slow, however the major advantage offered is the ability to simultaneously develop software through simulation, as the hardware is being developed. QEMU and Bochs are examples of hardware emulators. Native or Full virtualization uses a hypervisor which

acts as a mediator between the operating system and the underlying hardware. This method is faster than emulation, however the criteria is that the underlying hardware must be supported by operating system. KVM and VMware are examples of full virtualization solutions. Paravirtualization uses a hypervisor for shared access to the underlying hardware. This method offers the best performance. However, since the virtualization code is integrated into the operating system itself, the guest operating systems have to be modified for the hypervisor. Xen and User Mode Linux are examples of paravirtualization.

3 Our Setup

3.1 Hardware

The target for emulation is a custom board made by ABB. This board is based on the Freescale Lite5200 evaluation board[5]. The microprocessor on board is the Freescale MPC5200B[6], a PowerPC embedded processor based on the 400MHz MPC603e series e300 core. It is provided with 64 megabytes of RAM (DDR) and 32 megabytes of flash memory.

3.2 Software

The target board is a part of time critical control systems and hence there are real time constraints on the application response times[7]. To accomplish this, the target runs Linux kernel patched with the latest rt-preempt patch[8]. As of now, the version running on the hardware is Linux 2.6.29.4-rt17. However, since the real time performance is not a criteria as of now, we have reverted to the non rt version of the Linux kernel for this emulation. Hence, for the present time our requirements are limited to the latest mainline kernel and also support for ethernet ports. Ethernet is needed by the application to be run on the board emulator as well as for remote debugging of the application.

Traditionally, development of embedded software has been done on Windows host, cross-compiled and deployed on the target. In order to reduce the learning curve for embedded developers, it is important that the host platform for this emulation be retained as Windows.

4 Choices

A few virtualization methods and solutions were evaluated keeping the above needs in mind. Below is the list of the options considered.

4.1 Kernel Virtual Machine (KVM)

KVM[9] is a full virtualization solution for Linux on x86 hardware. It comes in the form of a loadable kernel module which provides the virtualization infrastructure as well as the processor specific implementations. KVM has been a part of the mainline kernel since 2.6.20 and is a very active open source project with a strong developer community. The PowerPC support however is still work in progress. Also there is no support from Windows as the host machine.

4.2 VMware Server

VMware server[10] is a virtualization platform which allows one to create and run virtual machines for different operating systems on a Linux or Windows x86 machine. It is partly open source. However, it does not support PowerPC emulation.

4.3 Xen

The Xen[11] virtual machine monitor is a paravirtualization solution for x86 and PowerPC architectures. The Xen hypervisor is the most privileged layer, above which a wide range of guest operating systems can run concurrently. The guest operating systems need to be modified to be able to run on Xen. Also the PowerPC support does not extend to MPC603e core processors.

4.4 PearPC

PearPC[12] is an architecture independent PowerPC platform emulator. This is an outdated project with no current developments ongoing.

4.5 QEMU

Qemu[13] is an open source machine emulator written by Fabrice Bellard. It has two modes of operation. In the User Mode emulation, it can execute binaries compiled for different architectures on a x86-Linux host. It can achieve near native performance in this mode. In the System Mode emulation, operating systems and applications compiled for different architectures can be run on the host system with a different Architecture-OS combination.

4.6 Why QEMU

Of the above options, a decision was made to use QEMU for the emulation of the ABB custom board. It was based on the following considerations.

- QEMU supports MPC603e core processor emulation

- QEMU supports both x86-Windows and x86-Linux as host machines
- QEMU is an active open source project with a good developer community and support

5 Programming Embedded systems using QEMU

An effective IDE which provides for efficient application development consists of two main components

5.1 QEMU

For the reasons mentioned above, QEMU is the system emulator that was selected to emulate the ABB custom board for which applications need to be developed. The applications can be run and debugged on the QEMU emulator just as they would on the real board. The compilers and linkers used to build the source code also remain the same, i.e. the application is compiled on the host machine for the Lite5200 target using cross-compilers (ELDK[14] on Linux and CodeSourcery Lite[15] on Windows) as usual. It is also possible to (remotely) debug the application running on QEMU from the host machine using GDB.

5.2 Eclipse

Eclipse is widely adapted as a development platform for embedded systems for many good reasons[16]. One of the main reasons is that it is an open framework with a plugin architecture that enables easy development and integration of a variety of software tools for embedded development. This is also the reason why it supports usage of a number of third party tools and interoperability between them, which is especially pertinent to embedded development. Most of the issues faced by embedded software developers like cross-platform development and debugging, application portability etc have been addressed and solved to a large extent by the vast open source community surrounding this project. A number of commercial embedded IDE vendors also employ Eclipse as their foundation to build upon[17].

Eclipse Ganymede[18] is used along with C/C++ Development Tools (CDT), Remote System Explorer (RSE) and Remote CDT as the development platform on a Windows host. The CDT perspective is used for application coding, compilation and linking. It provides a means of building the application for the Lite5200 platform thru the use of CodeSourcery cross tool chains for PowerPC. The RSE perspective

is used for establishing a SSH connection with the QEMU. The RSE provides a console to the QEMU and also enables one to browse through the remote QEMU filesystem. The SSH connection made by the RSE is also used by the Remote CDT debugger to remotely run and debug applications on the QEMU emulator.

6 QEMU on PowerPC

6.1 The right QEMU-PowerPC

Unfortunately, support for PowerPC platform in QEMU is unstable. The QEMU emulator version 0.9.1 works for the latest kernel versions on PowerPC platform [19]. The latest QEMU releases (up to 0.10.5) support Linux kernels only up to the version 2.6.26. Presently we have a 2.6.26 kernel running on PowerPC QEMU on Windows host that is primarily used for application development and debugging. We also have the 2.6.31 kernel running with the older QEMU on Linux host that is used as a test machine.

6.2 Kernel image for PowerPC

The Linux kernel image for QEMU-PowerPC is built in the usual manner, however arriving at a working kernel configuration and compiling a working system image requires effort. We use the standard Debian configuration (Lenny) with a few minor changes on Linux host. The DietPC[20] project has been a valuable resource in getting the QEMU PowerPC up on Windows host.

6.3 Getting QEMU-PowerPC up

The emulator is invoked using the following command

```
qemu-system-ppc -L . -M g3beige -m 128 \
-localtime -no-reboot -name P-Robo \
-hdc "P-Robo.img" -boot c \
-net nic -net user \
-redir tcp:22::22 tcp:2345::2345
```

Where `qemu-system-ppc` is QEMU PowerPC emulator.

- `-L .` points to the directory containing the openbios firmware.
- `-M g3beige` sets the emulation to beige G3 powermac which is compatible with PowerPC 603e cores.
- `-m 128` sets the virtual RAM size to 128M.

- `-localtime` sets the RTC to local time.
- `-no-reboot` sets up QEMU to exit instead of reboot.
- `-name P-Robo` sets the name of the guest to P-Robo.
- `-hdc "P-Robo.img"` directs QEMU to use `P-Robo.img` as the ide hard disk image.
- `-boot c` sets up the QEMU to boot from hard disk.
- `-net nice` creates a NIC card and connects it to a VLAN.
- `-net user` connects a user mode network stack to the VLAN.
- `-redir tcp:22::22` redirects SSH connections from host to guest.
- `-redir tcp:2345::2345` redirects default port for gdbserver connections from host to guest.

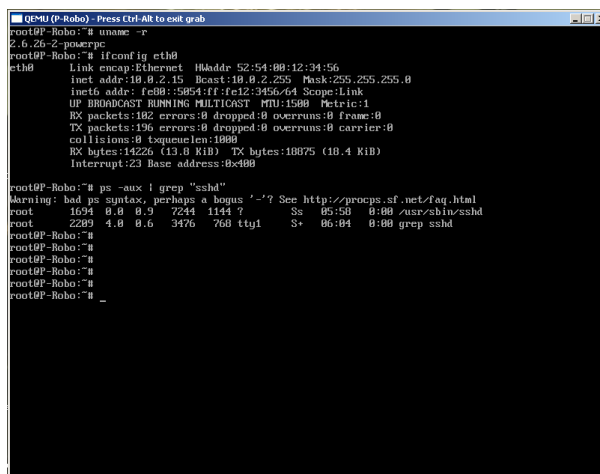


FIGURE 1: QEMU PowerPC on x86-Windows host

6.4 Eclipse

6.5 CDT and CodeSourcery

Eclipse Ganymede (with Eclipse CDT 5.0.1) is used as the IDE for developing and building the application. The CodeSourcery Lite cross tool chain for PowerPC is used to compile for mpc5200 target on Windows host. The cross compiler and linker are specified under the C/C++ build section in the project properties along with appropriate build flags.

6.6 SSH connection to QEMU

RSE (version 3.0.0 from DSDP) is used to create a SSH connection to the QEMU emulator. This can be done by connecting to the local host via SSH, since the SSH port of the host has been redirected to the SSH port of the emulator. The procedure to connect to the QEMU via SSH on root login is depicted by the snapshot below.

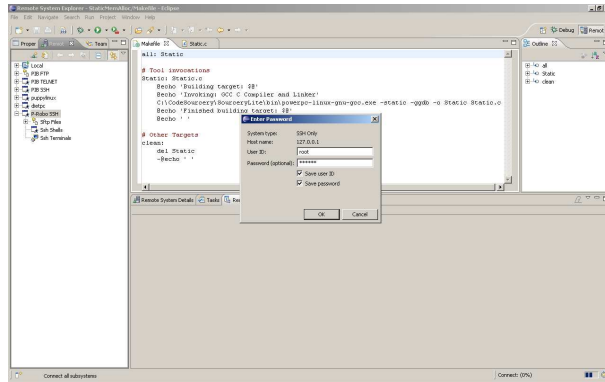


FIGURE 2: Creating a SSH connection from host to QEMU in Eclipse

Once this connection had been established a remote shell can be launched on the QEMU using the “Launch Shell” option under “SSH Shells” as shown below. This shell can be used to browse the emulator filesystem and run commands on the emulator.

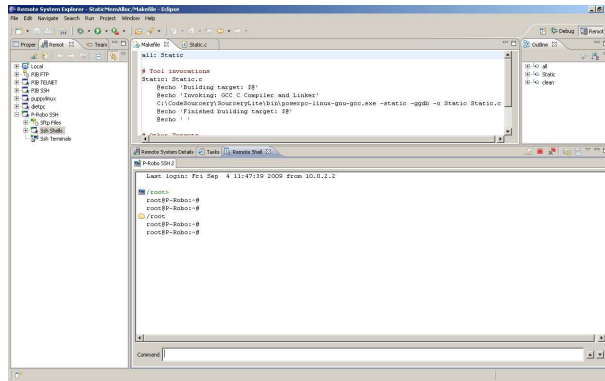


FIGURE 3: QEMU (remote) shell on Eclipse

6.7 Application Execution on QEMU

Once the application has been built for the target, it can be downloaded and run on the target by choosing the appropriate run configuration. The “C/C++ Remote Application” configuration has been selected and settings like “Connection”, “Remote Absolute File

Path” etc has to be set as shown below.

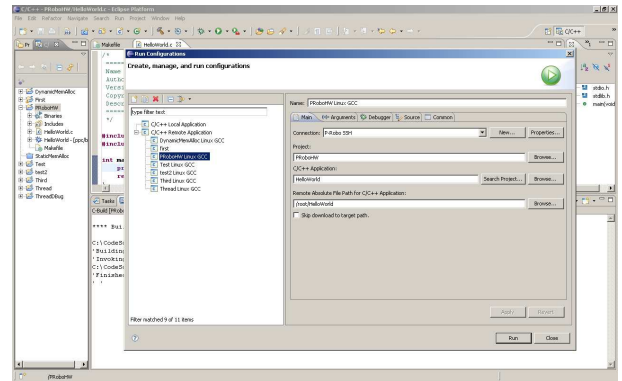


FIGURE 4: Setting up Eclipse to run the application remotely

On running the application, the output is displayed on a remote console on Eclipse as shown.

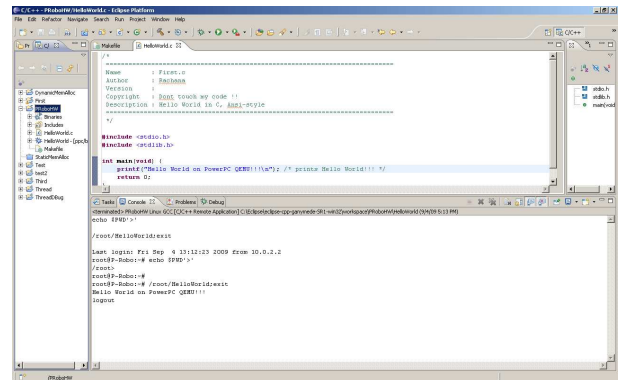


FIGURE 5: Remote QEMU console on application

6.8 Remote Debugging on QEMU

Once the application is running on the remote emulated board, the next step is to debug it. This is done by setting the debug configuration to “C/C++ Remote Application” and directing it to the right cross-tool gdb executable

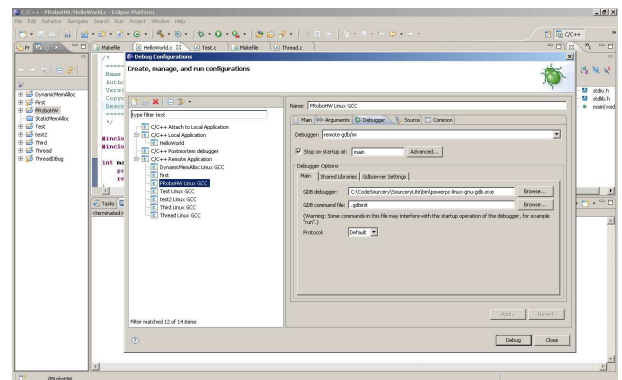


FIGURE 6: Setting up Eclipse to debug the application remotely

Debugging the application on the remote emulated host is done in a similar manner as with native applications. The snapshot below shows a very simple hello world program being debugged on the PowerPC QEMU.

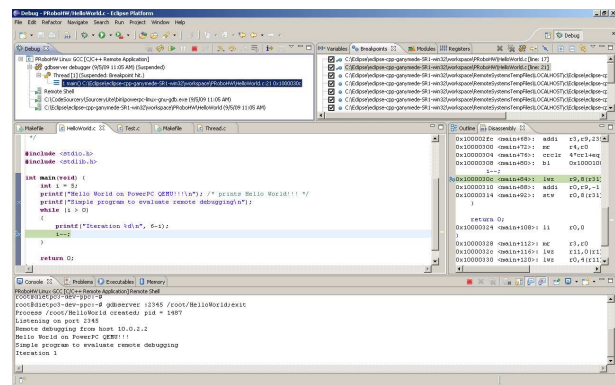


FIGURE 7: *Debugging application remotely on QEMU with Eclipse*

7 Conclusions

Using the above mentioned method, it is thus possible to create a development environment for embedded applications, that frees the developer team from the constraints imposed by hardware. It is also possible to further develop this into an secure and efficient testing platform. Virtualization enables this by providing the ability to simulate components like load generators, performance monitors and other resources beyond the scope of a usual test setup. QEMU proves to be a viable tool today to shorten development time for complex embedded system projects. When integrated with Eclipse it makes a fairly complete software development environment reducing the time to market for embedded products. Having a complete development and test emulation in place without having to depend on the underlying hardware, is a great advantage to embedded software developers attempting to optimize the product development in this fast changing market

8 Future Scope

QEMU for x86 based embedded platforms today is stable and effective. However, it has some distance to go when it comes to extensibility to allow users to easily customize peripherals and also support multi-core and latest CPU architectures other than x86.

The potential that simulation and emulation technologies offer in different stages of embedded software development is truly exciting. Be it easy and quick prototyping of research ideas for concept

validation or to simulate large and distributed systems in a virtual environment, these technologies save valuable time and effort. One interesting use of these technologies is in cross-platform development without the need for cross-compilers and linkers[21]. Cross compiling the BSP and user-space applications for another platform, is the most tedious and error prone stage of cross-platform development that requires repeated setting up of the environment variables and tool upgradation. Creating a virtualized native development environment for the applications is an efficient way to overcome these obstacles.

9 Glossary

QEMU	Quick EMulator
IDE	Integrated Development Environment
CDT	C/C++ Development Tools
RSE	Remote System Explorer
ELDK	Embedded Linux Development Kit
SSH	Secure Shell
GDB	GNU Debugger

References

- [1] <http://en.wikipedia.org/wiki/Virtualization>
- [2] <http://en.wikipedia.org/wiki/Emulator>
- [3] *Virtual Linux, An overview of virtualization methods, architectures, and implementations by M Tim Jones, IBM developerWorks*
- [4] *An Introduction to Virtualization by Amit Singh, Kernelthread.com*
- [5] http://www.freescale.com/files/microcontrollers/doc/prod_brief/MPC5200LITEPB.pdf?fsrch=1
- [6] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC5200B&fsrch=1
- [7] *Evaluation of Linux rt-preempt for embedded industrial devices for Automation and Power technologies - A case study by Morten Mossige, Pradyumna Sampath, Rachana Rao*
- [8] <http://rt.wiki.kernel.org>
- [9] http://www.linux-kvm.org/page/Main_Page
- [10] <http://www.vmware.com/products/server/>
- [11] <http://www.xen.org/>
- [12] <http://pearpc.sourceforge.net/>
- [13] <http://www.qemu.org/>
- [14] <http://www.denx.de/wiki/DULG/ELDK>

- [15] <http://www.codesourcery.com/sgpp/lite/power>
- [16] *Eclipse: Under The hood, Overview of the Eclipse open-source IDE framework* by Robert Day, *Embedded.com*
- [17] *Standardizing on a common embedded Eclipse IDE* by Martin Whitbread, *Embedded.com*
- [18] <http://www.eclipse.org/ganymede/>
- [19] Rob Landley's blog thing for 2009 - <http://www.landley.net/notes.html#25-06-2009>
- [20] *DIET-PC(Diskless Embedded Technology Personal Computer) project* by Paul A. Whittaker - <http://www.dietpc.org/>
- [21] *Firmware Linux - Embedded Linux build system* by Rob Landley - <http://www.landley.net/code/firmware/>

L4eRTL: Port of eRTL(PaRTiKle) to L4/Fiasco microkernel

Guanghui Cheng, Nicholas Mc Guire, Qingguo Zhou, Lian Li

Distributed and Embedded System Lab
School of Information Science and Engineering
Lanzhou University
Tianshui South Road 222, Lanzhou, P.R. China
cheng.guanghui@gmail.com

Abstract

L4/Fiasco is a real-time microkernel developed by Dresden Real-Time Operating System Group of Technical University Dresden and it is one of L4-family microkernels with the support of real-time capability. As said microkernel could only provide mechanisms not policy so based on Fiasco L4Env is developed to satisfy the different requirements as a available microkernel-based operating system. For a real-time kernel the posix real-time extension (pse-51) is widely accepted by industry and academic. However, now Fiasco can't support PSE-51 either. eRTL is developed by Real-Time Systems Group of the Universidad Politecnica de Valencia and it is aimed to embedded real-time application and as part of descendent version of RTLinux/GPL 3.0 it could support PSE-51 standard and some new extension only belonged to RTLinux such as pthread_delete_np. eRTL is well designed and implemented to be suitable for different environment such as the standard alone eRTL or as a domain in the XtratuM nanokernel from the same group as the eRTL. Therefore, it is reasonable to port eRTL to L4/Fiasco and provide an pse51 interface as the famous RTLinux.

1 Background

1.1 DROPS

The Dresden Real-Time Operating Systems Project shortly named DROPS is a research project in Technical University Dresden aiming at the support of applications with Quality of Service requirements in Technical University Dresden. This project is based on the second generation L4 microkernel named Fiasco written by C++. For the real-time capability Fiasco kernel is preemptible and it could only provide the basic mechanism like management of address space (task), thread, memory, and more importantly the implementation of synchronous fast ipc. The simple DROPS architecture is shown in Figure 1.

Around this microkernel lots of services totally named L4Env are developed aiming at a available real operating system like others. roottask is a basic resource manager including physical memory, interrupts, tasks and address spaces; DDE is a framework

used to port Linux device driver into DROPS; DOPE is a real-time window manager for GUI application. More, the creation, deletion of thread and semaphore is also provided, too. However, it is not posix compatible.

L4Linux plays a shining role in the DROPS and it could provide the same API/ABI as the widely-used Linux. Detailly L4Linux makes use of a pseudo "L4 CPU" as the physical CPU does and pure software "L4 CPU" is made up by lots of components from L4Env. From this feature it is possible to support more than 1 "L4 CPU" at the same time and so in fact they develop a paravirtualization technology based on L4 microkernel against Xen hypervisor.

Normally real-time application is divided into 2 parts: real-time part and non real-time part. Real-time part could run directly on the preemptible fiasco kernel and non real-time part could run an general purpose operating system like Linux. So in the real-time application L4Linux could give the environment for non real-time part with the use of abundant

of Linux libraries and applications. Hence, DROPS could provide the real-time solution as the parallel execution of the real-time task and non real-time task as described in Figure 1.

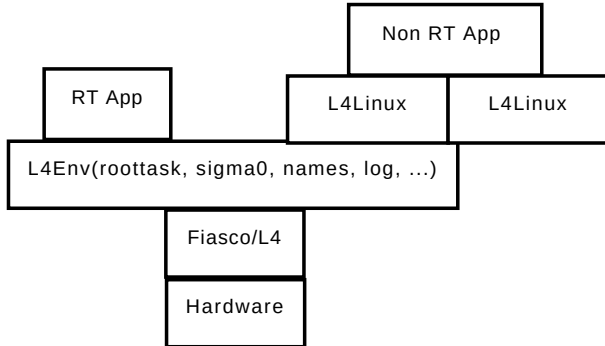


FIGURE 1: *Architecture of DROPS*

1.2 Real-Time in DROPS

IPC is one of the most famous attributions in the all the L4-family microkernels and since Professor Jochen Liedtke lots of researchers focus on fast IPC for many years. But in the real-time system IPC could be the potential cause of priority inversion when the high priority thread is trying to communicate with the low priority thread which is preempted by the middle priority. In the Fiasco the donation mechanism is invented to reduce the side effect of the IPC priority inversion. When the client tries to communicate with the server the client will donate its time quantum and priority to the server, vice versa.

About the resource management in the real-time community the separation of real-time part and non real-time part is still used not just in the calculation task. DOpE (Desktop Operating System Environment) separates the best effort screen updating into two parts: client redraw requests and server redraw operation in order to guarantee the QoS of real-time screen updating request.

1.3 eRTL

Officially eRTL could be considered as the successor of RTLinux/GPL which is too old to track the steps of mainstream linux kernel because of the problems in the architecture. But the interface of RTLinux is so friendly to use that eRTL only keeps the same interface but the architecture is redesigned. For many different kinds of applications with real-time requirements eRTL is designated to be portable and expandable.

PaRTiKle could provide the same real-time interface as RTLinux/GPL. But differently PaRTiKle could run on top of bare hardware as a single address space operating system aiming at the extreme requirements of real-time capability and a few functionality. At present this kind of PaRTiKle could run on top of ARM and x86 to be applied in the robotics system. But PaRTiKle could also be executed in the different domain separately from Linux kernel based on a nanokernel named XtratuM. This model of PaRTiKle is more robust and safer than the first single address space model but potentially with reduced real-time performance. In PaRTiKle the arch-dependent and arch-independent code is separated very well like in the Linux kernel. The directory architecture of PaRTiKle kernel are described below:

XtratuM [13], developed by the UPV (Universidad Politecnica de Valencia) has been rapidly evolving in the last several years. Currently it is at version 2.2. XtratuM is a nanokernel which means it does not include IPC mechanisms in the kernel core and thus it is smaller than a typical microkernel. Basically XtratuM provides interrupt and timer virtualization, minimum domain management and very simple memory management so it can support concurrent execution of many PaRTiKle. At first XtratuM adheres to Linux like as the traditional RTLinux does. But XM2 has been developed to be dependent from the Linux kernel. Similarly in the implementation of XtratuM the architecture-dependent part and architecture-independent part are separately very nice so it is so portable to run on X86, MIPS, PowerPC and Sparc.

TLSF(Two-Level Segregate Fit) allocator is a general purpose dynamic memory allocator designed to meet real-time requirements with bounded response time limitation, fast memory allocation and efficient memory usage. TLSF is O(1) cost for memory malloc, free, realloc and memalign with very low overhead. Now TLSF allocator is also used in many other projects.

2 Design and Implementation of L4eRTL

For L4eRTL, we have two basic solutions related to porting PaRTiKle to L4/Fiasco. The first method is to port XtratuM to Fiasco first. This seems easier because we only need to touch the very few XtratuM hypercalls (system calls). But if we port XtratuM on top of Fiasco and we try to support many instances of PaRTiKle we have two solutions like this:

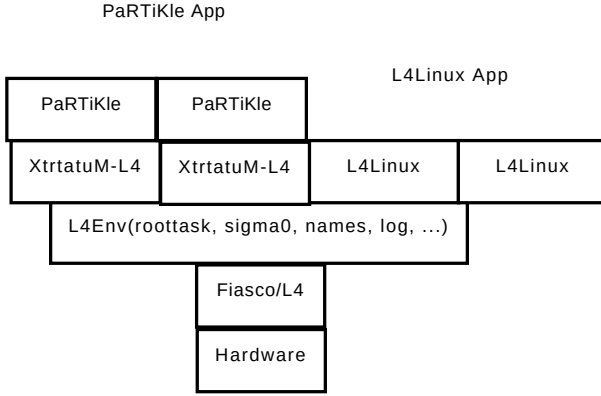


FIGURE 2: *Architecture of L4eRTL:1*

While this is straightforward porting, we can see that there are many XtratuM-L4 instances in the whole system. Of course what we need to do is little but performance loss can be expected.

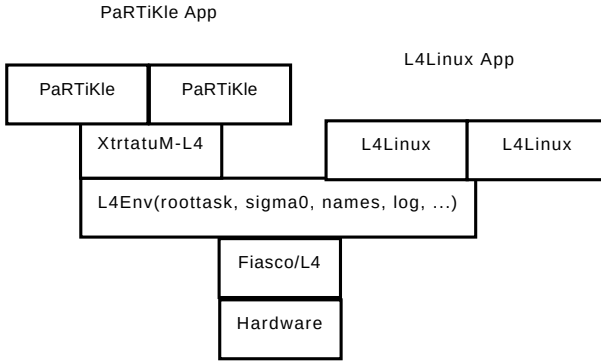


FIGURE 3: *Architecture of L4eRTL:2*

In this style if we design XtratuM-L4 as server, just like other L4 components, in which case we baically need to rewrite the XtratuM layer and generate different kinds of IDL files. In this mode there is the potential for substantial additional IPC related cost between PaRTiKle and XtratuM-L4. Further XtratuM has been moving on to XM2 which has seen a major redesigned of the core, So we can't keep our design bound to the XM 1 code.

Both of these is a departure from our actual target. Our first target is to port a POSIX interface for L4/Fiasco, but in the system the POSIX interface is provided by PaRTiKle not XtratuM.

Finally we choose to port PaRTiKle to L4/Fiasco directly and due to clean layering of the code, only a relatively small part of PaRTiKle source code needs to be reimplement with l4-specific APIs. So the architecture should be like this:

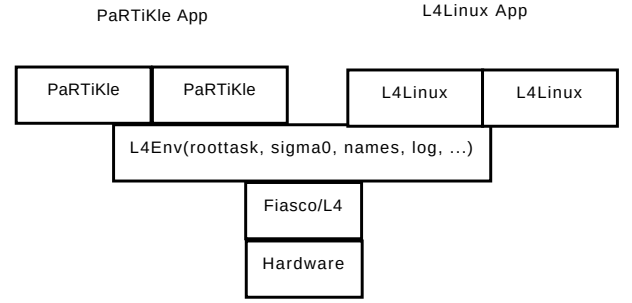


FIGURE 4: *Architecture of L4eRTL:3*

2.0.1 Hints in the Port

There are several aspects to be mentioned about the porting of eRTL to L4/Fiasco:

- **time**
Just as most operating systems, PaRTiKle provide two kinds of clocks a realtime and a monotonic clock source. The Real-time clock keeps track of current time which is a relative to an absolute (typically external) time based, on most systems 1970/1/1 is used as reference data, this clock can be adjusted and thus can stall or even go backwards. Monotonic clock on the other hand, means the clock value is monotonically increasing, it represents monotonic time since some unspecified starting point - the maching boot time in the most systems. In the X86 platform the real-time clock is derived from the RTC and monotonic clock is based on the TSC.

- **timer**

Fiasco doesn't provide the timer-virtualization or similar mechanism which could implement the user-space timer because Fiasco itself handles the timer interrupt. One solution about timer emulation in Fiasco is to (missuse) IPC timeout because Fiascos IPC is synchronous and thus when the timeout expires this IPC is cancelled which notifies the calling application. We could call this solution as the software timer, but it can't provide sufficiently accurate timer intervals compared with the hardware because in Fiasco timeouts are calculated by power expression $[\text{man} * 4^{(15-\text{exp})}]$ with an exponent and a mantissa in order to save some register bits, leading to a relatively low resolution. Never the less we are currently using the software timer to emulate the timer interrupt in this implementation. In the following code the `l4_rcv_timeout` converts the absolute

time to a timeout and `l4_ipc_receive` will return when the timeout expires.

```
pint=l4x_kinfo->clock;
//l4x_kinfo is the kernel info
//page which stores all the
//global information including clock.
while(1)
{
    pint += 10000;
    //about 10 milliseconds = 10000
    //microseconds
    l4_rcv_timeout(l4_timeout_abs(pint,
        L4_TIMEOUT_ABS_V1_ms), &to);
    //absolute time into relative timeout
    l4_ipc_receive(l4_myself(),
        L4_IPC_SHORT_MSG, &d1,
        &d2, to, &result);
    //wait for itself for the timeout
    l4_do_IRQ();
    //timer interrupt handler
}
```

Another option is to use real hardware to generate the timer interrupt, a normal PC-compatible machine has three devices available PIT (Programmable Interrupt Timer), RTC (Real-Time Clock) and APIC (Advanced Programmable Interrupt Controller). One of them is exclusively assigned to Fiasco and the two others are free to use for any other applications. In the Fiasco system it is allowed to use the user-space device driver. So the hardware clock for L4eRTL will be the choice in the future to remove the timer related issues found in the first implementation.

- hardware interrupt

In the Fiasco kernel all the hardware interrupts except the timer interrupt are translated into synchronous IPC processed then by the respective interrupt handler thread in user space. Normally `l4io`, which is a user-space resource handler, will manage the interrupts which it got passed on from the `omega0` (basic hardware interrupt multiplexer). And then application can do the actual processing of the interrupt it received from the `l4io` server. Thus hardware interrupts can be configured to be handled by L4Linux or L4eRTL or any user-space applications, though with some overhead due to the noted layered architecture.

- memory

L4eRTL executes in the user space on a L4/Fiasco system, so it only handles virtual

memory. For efficiency a continuous pinned memory block with predefined size is reserved for L4eRTLs main thread exclusively. And then this pool of memory is used to initialize by TLSF allocator. All application (thread) level memory operation in the L4eRTL are then handled via the TLSF allocator allowing dynamic memory resources in the bounded by the initial pool size.

- L4eRTL thread and L4 thread

A L4eRTL thread is different from L4 thread which is scheduled by L4 microkernel: the L4eRTL thread is scheduled by the internal L4eRTL scheduler which is one of the core PaRTiKle functions. For L4 kernel the L4eRTL appears only to be a single task which consists of L4eRTL timer thread, L4eRTL main thread and several L4eRTL interrupt handler thread if necessary. So the L4 kernel doesn't know anything about L4eRTL threads (or generally the internals of a task).

3 Conclusion and Discussion

Currently L4eRTL runs some basic tests-cases using a PSE51 compliant interfaces but the real-time performance of L4eRTL is not acceptable, most notably it is highly dependant on load produced in other tasks i.e. when a L4Linux instance is coexisting with the L4eRTL. Initial analysis suggests that the software timer is one of the main issues for the bad real-time performance so the next step is to fix it by utilizing a hardware timer. Generally the temporal isolation in L4/Fiasco does not seem to be as good as we had expected, though we are not yet sure if this must not be attributed to the current implementation of our L4eRTL. Improving the real-time performance of L4eRTL will thus be the main focus in the near future.

4 Acknowledge

Prof. Nichols Mc. Guire given us some important technology support. DROPS team from Technical University, Germany, gave us some help with this port. Most of work was done when the author was a guest student in the DROPS team, for which we would like to thank Prof Herman Haertig. DSLab offered resources and the appropriate surroundings for this development.

References

- [1] ,DROPS Overview,Hermann Haertig,
<http://os.inf.tu-dresden.de/drops/overview.html>
- [2] ,Ten Years of Research on L4-Based Real-Time,
Michael Roitzsch, Hermann Haertig, Proceedings of the Eighth Real-Time Linux Workshop, Lanzhou, China, 2006
- [3] ,Probabilistic Admission Control to Govern Real-Time Systems under Overload, Claude-J. Hamann, Michael Roitzsch, Lars Reuther, Jean Wolter, Hermann Haertig Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS 07), July 2007
- [4] ,Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video Roitzsch, Pohlack, Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), Rio de Janeiro, Brazil , 2006
- [5] ,Fast Component Interaction for Real-Time Systems, U. Steinberg, J. Wolter, H. Haertig Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05), Palma de Mallorca, Spain , July 2005
- [6] , Low-latency Hard Real-Time Communication over Switched Ethernet J. Loeser, H. Haertig Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004), Catania, Italy , June 2004
- [7] ,Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS) L. Reuther, M. Pohlack Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico , December 2003
- [8] , Demonstration of DOpE - a Window Server for Real-time and Embedded Systems N. Feske, H. Haertig Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico , December 2003
- [9] , OS-Controlled Cache Predictability for Real-Time Systems, J. Liedtke, H. Haertig, M. Hohmuth Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS'97), Montreal, Canada , June 1997
- [10] , The Performance of uKernel-based Systems, H. Haertig, M. Hohmuth, J. Liedtke, S. Schoenberg, J. Wolter Appeared at 16th SOSP, 1997
- [11] , TLSF: A New Dynamic Memory Allocator for Real-Time Systems ,Miguel Masmano,Ismael Ripoll,Alfons Crespo, ECRTS, 2004
- [12] , Quality-Assuring Scheduling in the Fiasco Microkernel, Udo Steinberg, Diploma Thesis Paper, 2004
- [13] *XTRATUM: AN OPEN SOURCE HYPERVISOR FOR TSP EMBEDDED SYSTEMS IN AEROSPACE*, A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J.J. Metge, 2009

Porting RT-preempt to Loongson2F

Wu Zhangjin, Nicholas Mc Guire

Distributed & Embedded System Lab, SISE, Lanzhou University, China
Tianshui South Road 222, Lanzhou, P.R. China
wuzhangjin@gmail.com, der.herr@hofr.at

Abstract

Loongsong2F, is a MIPS compatible single core CPU, it is a low-power, mid-range performance processor and therefore suitable for industrial usage. This paper focus on the effort of porting the RT-preempt extension of Linux to this processor based platform(FuLoong2F). we will briefly introduce the features of this platform, and outline the motivation of selecting RT-preempt, and then discuss the technical issues of porting RT-preempt to a new architecture, and describe the specifics of the platform. Further we cover the non-technical issues of how to integrate such work into mainstream development, and finally present some preliminary benchmark results based on community tool (notably cyclictst).

KeyWords: RT-preempt, Loongson, Real Time, Linux, industrial applications

1 Introduction

The 2nd generation of CPUs from Loongson[5] designed by ICT,CAS in China is currently at the 2F release, with the 2G to be expected this year. This MIPS compatible single core CPU is a low-power, mid-range performance CPU primarily targeting desktop and net-books. The features are listed in table 1.

CPU Core Frequency	800MHz-1GHz
Power Consumption	4 Watt.
Power Management	scalable cpu frequency
Built-in	North bridge,DDR2 Controller

TABLE 1: *Main features of Loongson2F*

The FuLoong(2F)[6] Mini PC made by Lemote, as a Loongson2F based machine, the total power consumption is only 10 watt which ensures quiet operation, and With small size (190x145x37mm) and light weight (0.85KG). It provides a large set of standard interfaces, including 4 USB2.0 Host High-Speed Interface, On-board dual LAN, 1 mini IDE interface, RS232 compliant serial port, 1 IR control receiver interface and it has reserved extended interfaces for wireless network card and LCD display, the form is shown in picture 1.



FIGURE 1: *FuLoong(2F) Mini PC*

Due to its low power-consumption, fan-less operation and integration in form-factors suitable for industrial usage, we have been working on porting suitable RTOS to it, the primary targets for this effort have been the XtratuM hypervisor and mainline RT-preempt[7]. We will not further cover XtratuM (which is in an early development stage) but rather focus on the mainline RT-preempt, to enhance its usability in industrial applications requiring reliable real-time services.

In the following sections, we will briefly outline the motivation of selecting RT-preempt and then focus on the technical issues of porting RT-preempt

to a new architecture and the specifics of the FuLoong(2F) platform. Further we cover the attempt of integrating such work into mainstream development, and finally present some preliminary benchmark results based on `cyclictst`[14]).

2 Why RT-preempt

The FuLoong(2F) machine, as introduced above, utilizes free software, thus is provided with source code for all system software and all standard services. Currently, it only runs Linux kernel based operating systems, including Lemote Loonux, Debian, Gentoo, Gnewsense (though Vxworks, Window CE reportedly can also run on it, but are not in the public market yet).

To meet the basic requirement of real-time services in industrial applications and also to make the RTOS porting project predictable, we selected RT-Preempt, after comparing several RTOSs for MIPS, including Vxworks, RTLinux/GPL, RTAI, Xtraum+PartiKle.

Vxworks is a commercial RTOS which is not easy to get, RTLinux/GPL is bound to the 2.4.X series of kernels and it's migration to XtratuM is not yet completed, RTAI for MIPS is too old, XtratuM+PartiKle is in its early development stage and targets very specific use-cases. But RT-preempt, as an Real Time Extension to mainline Linux, although the response time will likely always be a bit lower than RTLinux/GPL or a comparable dedicated RTOS, but is ready to "control an industrial welding laser"[7]. From the current development of RT-Preempt and its support by OSADL it can be expected that RT-Preempt will be able to cover a majority of the typical industrial real-time demands in the near future. Further one of the most attractive point is that there is an existing MIPS-specific support: 2.6.26.8-rt16, although it can't run on FuLoong(2F) directly.

And other than the realtime environment, RT-preempt is provided with the familiar Linux/POSIX programming environment, file systems, networking, and graphics which will help to expand the application area and also ease the Real Time Programming a lot. Although the current RT-Preempt source is a patch against Linux baseline, the active development of RT-preempt (the latest version is 2.6.31-rc4-rt1)[8] and the success of lots of upstream patches indicate that it can be expected that the whole RT-preempt will go mainstream within the next view mainstream releases. It should be noted that the upcoming 2.6.31 kernel already includes most of the essential RT-Preempt extensions and there are extensive efforts to unify core API (notably threaded interrupts).

3 Porting RT-preempt

The basic procedure of porting RT-preempt to Loongson2F includes,

1. download the original RT-preempt patches from its repository[8].
2. build or download cross-compile environment for MIPS/Loongson2F(only gcc>=4.4 has Loongson2F-specific support)
3. fix the grammar errors with the help of `cscope`, `objdump`, `nm`, `readelf`...
4. read, analyse the x86 & powerpc specific of RT-preempt
5. port the arch-dependent parts(including basic RT-preempt and debugging tools) to MIPS/Loongson2F platform
6. debug it on Qemu/Malta and FuLoong(2F)
7. optimize with KFT/Ftrace/Perf_counter/Oprofile, find out the hotspots and tune the different latencies(interrupt latency, schedule latency...)
8. benchmark with `cyclictst`
9. release & try to upstream by posting to the respective mailing lists and contacting the respective maintainers (MIPS and RT-Preempt).
10. fix the code once the first reviews come in and submit again

but this section will mainly focus on three parts, including how to get linux-2.6.26.8-rt16 running on FuLoong(2F) initially, migrate mips-specific support to linux-2.6.29-rc6 and later versions and real time debugging tools(`ftrace`, `perf_counter`) porting along with real-time performance optimization.

3.1 Getting linux-2.6.26.8-rt16 onto FuLoong(2F)

The original linux-2.6.26.8-rt16 includes basic MIPS support, but after fixing up a few compiling errors and running it on Qemu/Malta, it could not yet run on FuLoong(2F) directly. The problem is found out with the help of PMON(BIOS & Bootloader), `early_printk`, watch register of Loongson2F, and serial port, and at last locate the bug in `include/asm-mips/atomic.h`. The original RT-preempt tried to force all of the MIPS processors to use the `ll,sc` instruction to do the atomic operations if `PREEMPT_RT` was enabled, but it ignored the following two situations:

- No `ll,sc` instructions in the MIPS processor
- No macro `cpu_has_llsc` defined to 1

if any of the above condition occurs, the result will be: all of the atomic operations will be always empty or just before the `cpu_data[0].options` variable is initialized(if the macro is not defined to 1, it will be defined as `cpu_data[0].options & MIPS_CPU_LLSC`).

Unfortunately, although Loongson implemented the `ll,sc` instructions, but there is no macro `cpu_has_llsc` defined as 1, so, it failed when booting.

So, the corresponding fix is removing the buggy conditional compilation statement, and at the same time add a Loongson-specific `cpu-feature-overrides.h` to define the macro. To reduce the size and optimize the performance of Linux, other macros (related to describe the specific CPU features of the Loongson 2F) were also added in (i.e. `#define cpu_dcache_line_size() 32`, etc).

These macros will save about several hundred Kb of the kernel image, and will help the gcc compiler to remove the condition statement branches and at last optimize the performance.

3.2 Migrate mips-specific support to latest version of RT-preempt

As Linux is a rapidly developing code-base, there is little point in beginning a port with an out of date kernel (any kernel older than two releases is out-of-date) as this will break when trying to upstream any such patch-set. After making linux-2.6.26.8-rt16 working on FuLoong(2F) as a first step, We planned to do some optimization, but found out that `ftrace` does not works well on it and almost at the same time, found that linux-2.6.29-rcX-rtY was already released. Although 2.6.29-rcX-rtY was without mips-specific support, but the `ftrace` seems more mature, and there was a new `perf_counter` debugging tool introduced, so, We stopped to work on linux-2.6.26.8-rt16 and moved on to linux-2.6.29-rcX-rtY.

Since at that moment, the latest Linux for loongson is linux-2.6.27.1, the next step was to migrate Linux for loongson to linux-2.6.29 at first (mainstream not the rt-preempt variant), and then migrate the mips-specific RT-preempt. When doing this job, We have referred to these wonderful resources [9, 10, 2, 1] (one wonders where the claim that the Linux kernel is not well documented comes from).

To simplify the description of this migration, it's better to refer to the commit(70e06e) log of the latest mips-specific RT-preempt in the linux-loongson/2.6.29/rt-preempt branch of the rt4ls git repository[11], The main changes:

1. un-thread the timer and cascade interrupt

Add `IRQF_NODELAY` to the irq flags to un-thread the MIPS timer and the south bridge 8259 cascade interrupt. we must use the old spinlock if un-threading them, otherwise there will be deadlock. and although the i8253 timer is also un-threaded, but this MIPS timer is necessary instead of i8253. for it provides with higher precision clock for real time requirement.

2. using mutexed spinlock instead of old spinlock-/BKL in RT-preempt

Rename old `raw_spinlock/rw_t` to `_raw_spinlock/rw_t`, rename old `local_irq_disable/enable` to `raw_local_irq_disable/enable`, do real irq disable/enable operation on exception, system call.

3. using mutexed semaphore instead of old semaphore via adding a non-RT-preempt specific `rwsem.h`
4. reducing scheduling latency via adding a scheduling point in `do_signal`

Note that the FuLoong-specific part is very small, only need to un-thread the cascade interrupts.

3.3 Porting debugging tools & performance tuning

3.3.1 Porting ftrace

`ftrace`[13] comes from “function tracer”, but recently it becomes a tracing framework, various new tracers have been added via it, such as `irqsoff`, `preemptoff`, context switches tracers and so forth.

The original `ftrace`(static `ftrace`, dynamical `ftrace`, function graph tracer and system call tracer) is arch-dependent, which need to implement the arch-specific `_mcount` relative functions. And about the new tracers, although it is available, but has little value for real time tracing since they use the jiffies-based `trace_clock_local`(call `sched_clock()`) to record the local tracing times, so, they have very low timing precision(1/HZ, about 1ms when HZ is 1000), therefore the mips-specific `sched_clock()` needed to be implemented to get high timing precision.

- mips-specific `_mcount`

when enabling `HAVE_FUNCTION_TRACER`, the `-pg` option of gcc will be enabled to insert an `_mcount` function call to the entry

of any function without the “notrace” annotation or the functions in the files explicitly compiled without the -pg option (i.e. CFLAGS_REMOVE_ftrace.o = -pg), the function call looks as follows.

```
// save call_site's next pc to at
move    at,ra
// save next pc to ra and call _mcount
jal     _mcount
```

and the `_mcount` function itself works like a wrapper, if `ftrace` is disabled in user-space, it just return back to the caller.

```
jr      ra
move    ra,at
```

otherwise, it will call a real tracing function, and transfer the arguments in the `at` register and the `ra` register with an negative offset of `sizeof(jal _mcount)`.

When enabling `HAVE_DYNAMIC_FTRACE` and `HAVE_FTRACE_MCOUNT_RECORD`, the calling sites to the function `_mcount` are recorded to an ELF section (`_mcount.loc`), so, there is a possibility to replace this calling sites by a NOP operation (with more or less no impact) or changed back to a real call to `_mcount` and then we can indicate the functions to trace or filter.

and what about the `HAVE_FUNCTION_GRAPH_TRACER`? it is actually a simulation of the `-finstrument-functions` option of `gcc` which can generate instrumentation calls for entry and exit to functions. since the -pg only insert a call to `_mcount` at the entry to functions, so, need to simulate an exit to them, how? let's remove it's vail:

```
func:
    jal _mcount -> prepare_function_return
    ... (replace ra by return_to_handler)
    ...
    jr ra -> return_to_handler
    -> ftrace_return_to_handler
```

As the above description shows, when entering into the `_mcount` function, it call `prepare_function_return` and try to save the old return address, replace `ra` by `return_to_handler` and meanwhile, record the `calltime`, then when return from `func`, dont really return immediately rather call `return_to_handler` and then `ftrace_return_to_handler`, this function will record the `rettime`. Next calculates the function duration via `rettime-calltime`, and at last, does the real return to the saved return address.

- mips-specific `sched_clock()`

In x86, there is a `tsc`(64bit long) based high precision timing function `sched_clock()`, but this function of MIPS is jiffies based since although there is a similar register(\$9, clock counter) in MIPS, this counter is only 32bit long and thus incurs frequent roll-over.

But for real time tracing, the old millisecond level timing precision is not enough, and because the clock counter of MIPS has half of the CPU frequency, if the main frequency of CPU is 800MHz, the timing precision could reach $1/(400 \times 10^6 \text{Hz}) = 2.5\text{ns}$. So, obviously for RT we'd better implement a new `sched_clock()` based on this counter and the according roll-over handling needs to be consider. To avoid influencing the other parts of linux-mips, we only enable it for `ftrace`, here it is.

```
unsigned long long native_sched_clock(void)
{
    u64 current_cycles;
    static unsigned long old_jiffies;
    static u64 time, old_cycles;

    preempt_disable_notrace();
    /* update timestamp to avoid missing
       the timer interrupt */
    if (time_before(jiffies, old_jiffies)) {
        old_jiffies = jiffies;
        time = sched_clock();
        old_cycles = clock->cycle_last;
    }
    current_cycles = clock->read();

    time = (time +
            cyc2ns(clock, (current_cycles - old_cycles)
                    & clock->mask));

    old_cycles = current_cycles;
    preempt_enable_no_resched_notrace();

    return time;
}
```

3.3.2 Porting perf_counter

“The performance counters subsystem for Linux adds a new system call (`sys_perf_counter_open`) and provides a new tool(perf) that makes use of these new kernel capabilities. This subsystem and relative tool is new in that it tries a new approach at integrating all performance analysis related tools under one roof.”[12]

It has similar arch-specific APIs as Oprofile, and since Loongson has two performance counters and its Oprofile support is provided in the kernel already it's easy to migrate from Oprofile. At present, this Loongson-specific `perf_counter` is only implemented

in linux-2.6.29-rc6 and maybe out-of-date therefore not suitable to be introduced any more.

3.3.3 Real Time performance tuning

It's turn to tune the RT performance of mips-specific RT-preempt, so suitable tuning strategies need to be consider. When on the tuning procedure, we were very surprised to have found the paper [3], which gave us some very detail tuning methods, so not to repeat that paper, here is just list of the basic strategies:

1. compare compiler version
2. compare kernel versions and config options
3. tuning different latencies via kernel tracers
4. tuning via hotspots

Currently, the last step above is only a placeholder there, no deep try yet for that method is not that easy to enforce. The basic principle is catching the function level hotspots via ftrace or KFT and then getting the source code line level hotspots via Oprofile or kgcov, at last tuning the kernel via algorithm improvement, instruction sequence adjustment and other hardware specific low-level fixups.

4 How to upstream it

We have tried our best to push the whole MIPS/loongson-specific RT-preempt to mainline, but up to now, only a few parts were accepted While this sounds bad, the point is that getting patches into main-line (kernel/MIPS or RT-preempt) require adequate review by the respective subsystem maintainers, testing, coding-style related fixes, documentation and its not easy to get it right the first time. So the submissions to the respective mailing lists have resulted in changes to the patch both content wise and format wise, like moving it all to git - so it can be pulled into the respective tree, which is a standard method in the kernel development community.

This process is still on-going and has not yet resulted in the proposed patches being included (but they have not yet been kicked out either) and we hope to get the mainline support as well as the RT-preempt support to the upstream trees by the end of this year.

Currently, the MIPS/Loongson2F specific RT-preempt and ftrace,KFT,kgcov are maintained in the following git repository[11]. which also include linux-loongson kernel(>=2.6.29) for upstream: [git://dev.lemote.com/r4l.git](https://dev.lemote.com/r4l.git)

5 Benchmarking

For benchmarking a wide range of tools is available in the Linux kernel, we will not go into the details of each and every tool but rather focus on the most important RT benchmarking tool: cyclicttest.

“cyclicttest measures the delta from when it's scheduled to wake up from when it actually does wake up” [4].

to cover the mostly possible application environments when benchmarking, we add different load background as follows,

1. idle system
2. I/O load via about 50 “find /” background
3. Network load via “ping -f”(ping flood)
4. X window
5. the above combination
6. X window & Mplayer(Video)
7. the above combination

and the mostly possible interval(us) options of cyclicttest are used, including 100, 200, 500, 1000, 2000, 5000, 10000. to ensure the result is more credible, all of the tests are executed 100,000 times(loops). besides, The test thread runs in all cases with SCHED_FIFO, priority 80 and clock_nanosleep.

```
cyclicttest -p80 -t1 -n -i$interval -l100000 -v
> cyclicttest_$interval.log
```

and here lists the other testing environment for result reproducible.

1. gcc 4.4 with “-march=loongson2f”
2. binutils 2.19.1 with a patch[15] from Lemote
3. cyclicttest with a patch[16]

Table 2 and picture 2 shows the statistic result.

Load	Max	Avg.	Std.Dev.
idle	27	6.986	0.684
ping	32	8.144	1.548
x	34	7.312	1.500
find	68	15.553	5.888
ping,x,find	60	15.163	5.092
x,mplayer	99	13.132	7.678
ping,x,find,mplayer	82	15.902	5.764

TABLE 2: *RT-preempt testing report*

the first column is the load, the following three ones are event response times in microseconds, they are maximal, average, and standard deviation individually. the difference among different intervals are not shown in this picture to avoid the table too long, but they are coped with together(cyclictest_*.log).

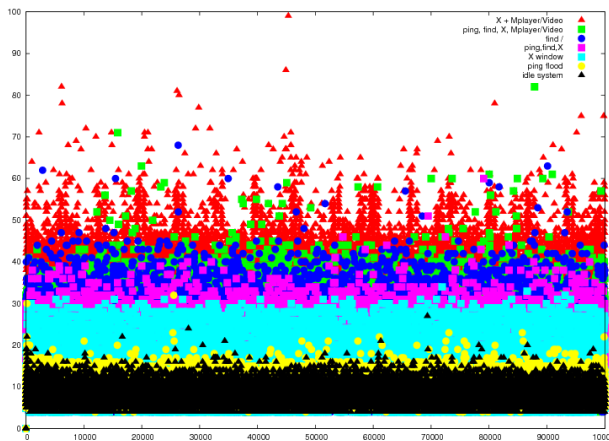


FIGURE 2: RT-preempt testing report

The X axes is the loops counts, the Y axes is the event response time in microseconds.

Currently there do not seem to be any data collections on RT-preempt values that allow an easy comparison of the Loongson platform to others, never the less we give the following references [14, 17].

6 Conclusion

The Loongson 2F has favorable hardware properties for the use in industrial applications, its GNU/Linux support made it well suitable for the integration into existing projects - what was missing though is a widely accepted and stable RTOS support, with RT-preempt this is now available for the Loongson 2F. Further it showed that joining an open community like the kernel development community is not as complicated as it might seem and that they are very supportive, allowing an efficient and swift port to a new architecture - which makes GNU/Linux an even more attractive target (RT)OS for hardware companies.

While the RT aspects initially were the main focus it quickly became evident that the tooling capabilities are at least of equal importance and here clearly the capabilities of the Linux kernel and the RT-preempt community proved to be mature and complete, allowing efficiency in the porting job and in the assessment of the results.

We hope that the next generation of Loongson CPUs (2G) will not only support RT-preempt but find its way into mainline Linux from the very start

- this will be our next major challenge after concluding the work of up-stream integration of the current Loongson 2F work.

7 Acknowledgment

We would like to thank the MIPS maintainer Ralf Bechle, and the RT-preempt maintainer, Thomas Gleixner, and also the other contributors from the FLOSS communities, for their support and critical review comments. Further we thank the folks at Lemote for supporting this effort, notably Dr. Zhang Fuxin. Last (but not least) the members of DSLab for discussions and support for this undertaking.

References

- [1] Steven Rostedt & Darren V. Hart, *Internals of the RT Patch*, P161-172 of Proceedings of the Linux Symposium, June 27-30th, 2007
- [2] Michael Opdenacker, Free Electrons, 2003, *Introduction to rtpreempt patches*
- [3] Frank Rowand, November 7, 2008, *Adventures In Real-Time Performance Tuning*
- [4] Clark Williams, *An Overview of Realtime Linux*, Redhat Summit, June 18-20th, 2008.
- [5] Loongson, <http://www.loongson.com>
- [6] FuLoong, <http://www.lemote.com/english/fuloong.html>
- [7] RT-preempt, <http://rt.wiki.kernel.org/>
- [8] <http://www.kernel.org/pub/linux/kernel/projects/rt/>
- [9] <http://www.ibm.com/developerworks/cn/linux/l-lrt/part1/>
- [10] <http://www.ibm.com/developerworks/cn/linux/l-lrt/part2/>
- [11] <http://dev.lemote.com/code/rt4ls>
- [12] perf_counter(v8), <http://lwn.net/Articles/336542/>
- [13] ftrace, Documentation/ftrace.txt
- [14] cyclictest, <http://rt.wiki.kernel.org/index.php/Cyclictest>
- [15] <http://groups.google.com/group/archlinux-for-loongson/web/binutils-2.19.1-loongson2f-3.patch>
- [16] <http://dev.lemote.com/code/rt4ls/raw-attachment/wiki/WikiStart/0001-cyclictest-ensure-it-work-with-glibc-2.7-in-linux-M.patch>
- [17] http://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT

Offline Scheduler

Raz Ben Yehuda

The Open University of Israel
Department Of Mathematics and Computer science
Halotem 8/1, Rishon Le Zion, Israel
raziebe@gmail.com

Abstract

OFFSCHED is a platform aimed to assign an assignment to an offloaded processor. An offloaded processor is a processor that is hot un-plugged from the operating system. In today's computer world, we find that most processors have several embedded cores and hyper-threading. Most programmers do not really use these powerful features and let the operating system do the work. At most, a programmer will bound an application to a certain processor or assign an interrupt to a different processor. At the end, we get a system busy in maintaining tasks across processors, balancing interrupts, flushing TLBs and DTLBs using atomic operations even when not needed and worst of all, spin locks across processors in vein; and the more processors the merrier. I argue that in some cases, part of this behavior is due to fact the multiple core operating system is not service oriented but a system oriented. There is no easy way to assign a processor to do a distinct service, undisturbed, accurate, and fast as long as the processor is an active part of an operating system and still be a part of most of the operating system address space.

The purpose of the OFFSCHED is to create a platform for services. For example, assume a system is being attacked; the Linux operating system will generate endless number of interrupts and/or softirqs to analyze the traffic and throw out bad packets. This is on the expense of good packets. Have you ever tried to ssh to an attacked machine? Who protects the operating system? What if we can simply do the packet analysis outside the operating system, without being interrupted? Why not assign a core to do only firewalling? Or just routing? Design a new type of Real Time system? Maybe assign it as an ultra accurate timer? Create a delaying service that does not just spin? Offload a TCP stack? perhaps a new type of a locking scheme? New type bottom-halves? Debug a running kernel through an offloaded processor? Maybe assign a GPU to do other things than just graphics? Amdahl Law teaches us that linear speed-up is not very feasible, so why not spare a processor to do certain tasks better? Technologically speaking, I am referring to the Linux kernel ability to virtually hot unplug a (SMT) processor; but instead of letting it wonder in endless halts, assign it a service.

1 Introduction

"With the advent of multithreaded/multicore CPUs, even embedded real time applications are starting to run on SMP systems, for example, both the Xbox 360 and PS/3 are multithreaded and there even have been SMP ARM processors! As this trend continues, there will be an interesting need for real time response from SMP systems. Because not all embedded systems vendors will be willing or able to create or purchase SMP real time operating system, we can expect that number of them will make use of

Linux". [1] Paul E. McKenney This paper argues that OFFSCHED fits to the niche of Multiprocessors real time systems by partitioning a system to two; the operating system and OFFSCHED. OFFSCHED is a hybrid system. It is hybrid because it is both real time and still a regular Linux server. Real time is mainly achieved by the NMI characteristic and the CPU isolation. It is a hybrid system because OFFSCHED scheduler interacts with the operating system.

2 OFFSCHED benefits

2.1 PROS

- **Accuracy**
A programmer may rely on the fact that only his code will be run without any disturbance because offloaded cores runs in NMI mode).
- **Security Sand-Box**
OFFSCHED is designed to assign a service to a single resource. This makes the load actually contained inside OFFSCHED. Example for that is a firewall contained inside a single core.
- **RAS - Remote Access Services**
OFFSCHED can be used as remote access server (such as a simple telnet server) inside the machine but outside the kernel. This can be useful in cases like getting operating statistics and monitoring an attacked system.
- **Better System Performance**
OFFSCHED flow is serial in its nature. Read-modify-write assembler instructions whose opcode is prefixed with the lock byte 0xf0 are less likely to appear in the code so memory bus contention is reduced. OFFSCHED waits in 'pause' assembler instruction, so pipe cache lines are not invalidated.
Branch prediction is one of the processor's ways to speed up code execution. Problem with branch prediction is that a branch might get flushed too early. Branch flushing happens mainly due to interrupts, exceptions, segments descriptor loads ([2] page a-51). OFFSCHED is running NMI so less pipeline flushes happen.
- **Speed-Up**
Designed correctly, Amdahl Law [3] for linear processors speedup truly is achievable. A good programmer that the real time code is very small compared to the rest of the software and will reduce the program footprint.
- **Consistency**
Offloading a processor has an additional property of being able to determine the amount of time it will take it to accomplish a task in a higher resolution than an active processor.

- **Hybrid System**
OFFSCHED joins two different system types in a single machine, a Real Time system and a server. This means that a data-base server and a F15 simulation may coexist. The benefit is that if the F15 simulation produces output to the data base server, it all can be done on the same machine. This is possible because OFFSCHED code is small and centric (actually, OFFSCHED is a driver with a small kernel patch) and as such it does not jeopardise the kernel stability. Other real time systems are spread all over the kernel.
- **Asymmetric Processing**
With the upcoming of Intel Nehalem processors, a machine with 4 physical cores may have 4 additional symmetric multi-threading processors. The processing power of a SMT processor is less than a physical core, so two processing units are asymmetric. Linux make this distinction by keeping CPU load accounts. But load accounts can only tell what happens only after it happens. OFFSCHED can utilize SMT processor by assigning this type of processors a distinct service suited to its processing power.

2.2 CONS

- **No User Space**
OFFSCHED cannot run user space processes.(A suggestion made Rik Van Riel was to create a scheduler just for user space tasks).
- **No Context Switch**
OFFSCHED does not perform context switch. OFFSCHED has a single context.It is up to the programmer.
- **No IPIs**
OFFSCHED does not receive IPI [4] (Inter-processor interrupts, no TLB invalidate, vectors 251-255).
- **No Direct access to a vmalloc memory**
OFFSCHED cannot use vmalloc'd memory. To access user space memory it must "walk on the pages".This is because whenever a processor accesses an empty pfn, it generates a page fault interrupt [4], but since OFFSCHED is NMI, this cannot be done and therefore this pfn remains invalid.
- **Development effort**
OFFSCHED is difficult to develop. Development should be done by a kernel developer.

3 Dynamic Data Management

This means a minimization of memory overhead delay when accessing data. Delay may improve by assigning tasks by a policy of a minimum memory delay. In OFFSCHED, memory access is a matter of a good scheduling design. Due to OFFSCHED serialized nature, memory low latency scheduling policy can be easily achieved. An example is that there is no consideration for the cost of stack switch. Another example is how messaging is handled. Process A assigns a message to process B. Process A schedules process B after himself and exits. Process B enters and the message is in the processor cache. There is no doubt that B will come after A, so the message in the processor's cache as long as its footprint is small enough.

4 RCU starvation

One may ask, why not create a busy loop like:

While (can I do my thing)

Do my thing

The reason is that this code will create a RCU starvation. RCU is a synchronization technique which enables multiple readers and multiple writers to access mostly accessed data structures from multiple processors. RCU starvation happens because each processor must walk through a quiescent state [?]. A quiescent state is when one of the below happens:

1. A processor performs a context switch.
2. A processor executes user mode.
3. A Processor executes the idle loop.

And neither of these will happen in a busy loop. OFFSCHED eliminates RCU starvation since the processor does not have to walk through a quiescent state as it is not part of the operating system.

5 Out-of-order execution

Consider the following sequence of operations:

INC RAX

ADD RCX,RBX

These instructions are independent. In regular pipelined processors these two instructions are not likely to run concurrently. Some commands might be associated with interrupts and exceptions. If the interrupt service routine is running in the initiating processor context, RCX and RBX content is lost

unless some additional logic is added to the processor. Either case, a processor has to do additional work that reduces performance. The NMI property of OFFSCHED actually makes program running in OFFSCHED context run faster than in a non-OFFSCHED processor.

6 The price of a context switch

When designing multi-threaded software, one has to understand the price of a context switch with respect to the system requirements. The below is an ftrace performed on a My Laptop, Lenovo with Intel Core 2 Duo 2GHz T7300, the same laptop where OFFSCHED performs a 1us timer. See how a context switch between the idle context to events costs at best 12us.

```
tracer: sched_switch
TASK-PID CPU TIMESTAMP FUNCTION
...
idle 01 225.481287: 0:140:R 10:115:R
events 01 225.481299: 10:115:S 0:140:R
...
```

OFFSCHED 1us timer shows that it is possible to work in high resolution scheduling. Though, the amount of work that can be achieved in 1us is very little. See OFFSCHED-Timer paper.

7 scalability

Moore Law presents us with a simple fact, if in 2008 we have 4 cores in a single die, in 2010 we will be having 8 cores, in 2012 16 cores, in 2014 32 cores etc. Does the current Linux operating system design fit the Multi core era? Should the operating system handle so many cores?

As this paper argues, the answer is NO. Having an operating system balancing tasks between 16 processors is not necessarily good performance. Linux kernel migrates pages and moves tasks between processors repeatedly to achieve balance in the system. OFFSCHED, on the other hand, does not care much for imbalance; it aims for responsiveness, accuracy and simplicity. OFFSCHED is designed to do serve an explicit task. OFFSCHED can produce linear speed-up as long as it is not bounded by BUS speed. Assignments can easily be spread in a balanced manner across the processors.

8 The concept of offlet

Offlet is the context of offline processor. When an offloaded processor is removed from the operating system, it is simply set to the halt assembler instruction while interrupts are disabled. Right before the halt instruction is where Offsched enters. From now on we say we are in offlet context. Yet, offlet context has some restrictions that a user must be aware of:

1. user may not cause any page fault[7], meaning he cannot access vmalloc'ed memory (only by "walking on the pages").
2. A user may not "STI".
3. A user may not invoke any code path that ends up in "schedule".
4. Allocation and de-allocation of kmalloc'd memory is still experimental.

This is why I refer to offsched context as an offlet; it is because it has more constraints than any other kernel context such as softirqs.

8.1 Offlet on the stack

Offlets may be allocated on the stack, schedule and released; very much like any other kernel entity but much faster, so fast that it will be faster than the initiating context.

A code example should be:

```
...
offlet_timer x
offlet_timer_init(x,my_func)
offlet_schedule(x,0)
...
```

8.2 offlet seize delay release scheme

Offsched is based on the notion that processors are not an expensive resource. In general, when task A is in kernel context and wishes to seize resource X all it does:

Step 0) ..

Step 1) Lock

Step 2) ..

Step 3) Unlock

Step 4) ..

If Step 4 is not serialized with respect to the previous steps then Offlet suggests an asynchronous service, instead of spinning in step 1, we ask the offsched processor to do steps 1 2 3 for us. Offlet scheduling is very light so the scheduling cost is meaningless in term of computation.

Another added value of offlets is that a system may choose to serialize access to a device through an offlet. A good example is OFFSCHED NAPI (see OFFSCHED NAPI paper). This serialization actually relieves the kernel from contention on slow devices.

9 conclusion

OFFSCHED can be used a real time system. OFFSCHED does not compete with other real time OS, but co-exists. It is aimed to the servers market and the embedded, as it relinquishes cores.

10 acknowledgements

I would like thank Mr. Jeff Roberson, who came with this idea (and the first draft) of offloading a whole core.

References

- [1] Paul E.Mckenney, "SMP and EMBEDDED REAL TIME", Linux Journal, issue 153, Jan 2007.
- [2] Intel Corporation, IA-32 Intel Architecture Software Developer Manual, Volume 3, System programming guide.
- [3] Michal J Flynn. "Computer Architecture PIPELINED AND PARALLEL PROCESSOR DESIGN" by Jones and Bartlett Publishers, Inc copyrights 1995
- [4] Daniel P.Bovet & Marco Cesati. "Understanding the Linux Kernel". O'Reilly Media Inc. Copyrights 2006

Reservation-Based Scheduling for IRQ Threads

Luca Abeni

University of Trento
Via Sommarive 14, Povo (TN)
luca.abeni@unitn.it

Nicola Manica

University of Trento
Via Sommarive 14 Povo (TN)
nicola.manica@gmail.com

Luigi Palopoli

University of Trento
Via Sommarive 14, Povo (TN)
palopoli@disi.unitn.it

Abstract

Scheduling applications that require multiple hardware resources (CPU, disk, network, etc...) is known as a challenging issue in real-time systems, due to the interference from device drivers (in particular, interrupt handlers) on high-priority real-time tasks. This paper proposes a possible solution for this problem, based on scheduling interrupt handlers through a reservation-based scheduler. The solution is based on a combination of the Realtime-Preempt patch for Linux (which transforms interrupt service routines in threads) and a reservation-based scheduler, the Constant Bandwidth Server - CBS (which is used for scheduling such threads).

1 Introduction

Real-time scheduling theory has traditionally dealt with the problem of scheduling the CPU so that the execution of a set of concurrent tasks can meet some timing constraints. The kind of real-time constraints considered range from hard real-time constraints (requiring strict and deterministic execution guarantees) to soft real-time constraints (for which occasional violations can be tolerated and probabilistic performance guarantees are required). However, scheduling real-time applications that simultaneously use multiple hardware resources (such as CPU, disk, network, ...) is still an open issue. As one of the possible problems that arise in this context,

let us focus on the execution of device drivers. In traditional non real-time operating systems, device drivers are executed as non-preemptable code inside the kernel. Therefore, they can block the execution of real-time threads, even if the latter have in that moment a very high priority. This way the real-time guarantees are at serious risk of being disrupted.

A preliminary step to solve this problem is recognised to be making the device driver schedulable entities alongside of real-time threads. One possible way to do this (e.g., pursued in μ kernel based systems [1, 2]) is to execute Interrupt Service Routines (ISR) inside IRQ threads. Recently a similar solution has been implemented also in the Real-time Preempt Linux kernel patch (by Ingo Molnar and oth-

ers). Some recent pieces of work have shed some light on a largely underestimated problem: **a real operating system kernel needs some CPU time to exchange data with hardware devices** [3, 5]. For instance, it is completely useless to precisely schedule a device (e.g., a disk) if the kernel is not able to find enough CPU time to manage the incoming data. And, the CPU time spent by the kernel for handling the device must not be accounted to real-time tasks that do not use such a device (causing deadline misses) This means that a really coordinated strategy for the scheduling of different resources [6] is needed.

In a previous preliminary work [7], we have shown that by selecting appropriate fixed scheduling priorities for the IRQ thread it is possible to avoid the interference of the device driver on the real-time threads, thus facilitating the enforcement of real-time constraints on their execution. In the same paper, we have shown that the price to be paid for this solution is an abrupt reduction in throughput of the IO operations. The problem is largely due to the inadequacy of a fixed priority scheduling, which for this type of applications reveals itself too coarse a tool. To get an intuition, by priorities we are only able to decide the relative importance of the tasks. This can be sufficient to say whether or not a very strictly characterised task set is able to execute concurrently meeting the time constraints [8]. If, on the contrary, we aim at a more complex goal, such as striking a good balance between real-time guarantees of a thread and throughput of an IO burst, it is not sufficient to allow or deny altogether the possibility for a device driver to steal execution time to a real-time thread. We need to control, in some sense, the amount of time that a device driver can take from a real-time thread. In other words, we need to complement the idea of making each device driver a schedulable entity with the adoption of a scheduling strategy (such as the resource reservations [10]) enabling a fine tuned control of the fraction of CPU allocated to each task.

This paper lies exactly in this line: we show how it is possible to combine the adoption of the RT patch (whereby each device driver can be execute in an IRQ thread) with a scheduling algorithm known as the Constant Bandwidth Server (CBS) [11] (which allows us to control the CPU fraction allocated to each thread). In the paper, we discuss how to carry out the integration between the RT patch and a Linux based implementation of the CBS [12]. In our experimental section we show how the combined use of

this technology offers substantial improvements over both the standard technology, and the simple use of the RT patch with fixed priorities.

2 The Problem

As discussed above, if some applications in the system use hardware resources other than the CPU they can “steal” execution time to high-priority real-time tasks. This happens because the OS kernel needs to consume CPU time in handling interrupts coming from the various hardware devices providing the resources, and traditional (non real-time) kernels tend to serve all the interrupt requests before scheduling user tasks (even real-time ones).

This time spent in serving interrupt requests can be a problem for real-time applications, because a low-priority, non real-time, task can affect the schedulability of a high-priority, real-time, task by performing heavy I/O (for example, a non real-time “find /” operation is known to be the source of high kernel latencies).

When a hardware interrupt fires (an Interrupt ReQuest - IRQ - arrives), it is served by a short Interrupt Service Routine (ISR, or hard IRQ handler) and by a “postponed interrupt handler”, called soft IRQ handler, which has a lower priority than the hard IRQ handler but still has a higher priority than regular threads and processes. As a result, hard and soft IRQ handlers can consume CPU time that should be devoted to more important real-time tasks. In other words, interrupt handlers can be the source of high kernel latencies¹. As a consequence, a low-priority task can jeopardise the schedulability of high-priority real-time tasks by causing the generation of a large number of hardware interrupts. The standard methodology to decide the schedulability of a set of real-time tasks is by some admission tests using the total utilisation of the tasks [8], or by response time analysis [4]. There are standard ways to enhance such admission tests to account for the blocking time introduced by non-preemptive code segments and/or for the interference introduced by ISRs (modelled as high priority tasks). This way, the time “stolen” by ISRs can be accounted in the schedulability analysis.

A very popular way to reduce the impact of interrupt handlers on real-time tasks is to make the interrupt handlers subject to the scheduling policy rather than a non-preemptive section (generating

¹The kernel latency is defined as the difference between the time when an event happens and the time when it was expected to happen.

blocking time for all tasks irrespective of their priority). For this reason, the RealTime Preemption patch (RT-preempt) [9] for the Linux kernel transforms hard and soft IRQ handlers in kernel threads (the hard IRQ thread and the soft IRQ thread), that are *schedulable entities* handled by the task scheduler in the same way as user tasks (so, IRQ threads can have lower priorities than real-time tasks, and can be preempted by them). A real-time application that does not need to interact with a specific device can schedule its tasks in foreground respect to the device's interrupt handlers, so that real-time tasks are not disturbed by the device's interrupts.

This solution can present a slightly higher overhead, and requires a more careful synchronisation, but also has the advantage of permitting to correctly account the handler code in a real-time system (that is, the CPU time required to execute the handler can be correctly accounted in order not to break the system's guarantees). The possibility to schedule interrupt handlers (provided by IRQ threads) permits to give user-space real-time tasks higher priorities than interrupts, reducing the interference from hardware devices.

Although a very important step, the possibility of scheduling interrupt threads is not a conclusive answer to the problem of providing real-time guarantees to applications with intense I/O operations. Indeed the problem remains open of how to properly assign scheduling priorities. While effective methodologies such as Rate-Monotonic or Earliest Deadline First [8] can be used for standard real-time task sets, when IRQ threads are present in the priority assignment is to be considered largely an open issue (as of today). As discussed next, an optimal priority assignment in this case could even be impossible to find.

Of course, it is easy to find priority assignments that provide good real-time performance in specific cases: for example, when real-time applications do not need to access a hardware device, the IRQ threads provided by RT-Preempt can simply be assigned a low priority reducing their interference on real-time tasks. The solution is much less obvious when some real-time activity use the data coming from the device. To expose the problem, consider a real-time application composed of periodic threads and of a thread receiving data from a hardware device. The simplest configuration revealing the problem is given by a periodic real-time thread and by a thread receiving high-bandwidth network data. In our experimental setup:

- a network card has been selected as an inter-

rupt generating device because it is easy to generate a controlled load on it, and to measure the **network** throughput;

- a periodic periodic real-time thread (scheduled using real-time - **SCHED_FIFO** - priorities) has been used to generate some time sensitive **CPU** load;
- real-time performance have been quantified by measuring the response time of the periodic thread. Such response time is strictly related to the kernel latency [12], which represents a good measure of the real-time performance of the OS kernel, because it must be accounted for in the admission test as a *blocking time* B , so high latency values risk to make unschedulable task sets that would be schedulable if kernel effects were not considered.

The periodic real-time task has been implemented by using a program that periodically executes a busy loop (this way the computation time of each job of the task can be approximatively considered as a constant one). This program is similar to **cyclictest** program [13], but it consumes a considerable amount of time at each period, and it measures the response time of each activation. When the periodic task is the highest priority task in the system, the response times are expected to be equal to the execution times; otherwise, the interference from higher priority tasks must be considered. In this experiment, a periodic task with period $P = 20ms$ and execution time between $6ms$ and $8.5ms$ has been used. The network throughput has been measured by using the **netperf** program [14].

When using the standard Fixed Priority Scheduler provided by Linux, the real-time task and the network IRQ threads have been scheduled through the **SCHED_FIFO** policy, and various priority assignments have been tested to check their impact on the latency and on the throughput. In particular, the goal of this set of experiments was to check how manipulating the priorities of the interrupt threads allows us to control the real-time tasks' latency and the network throughput. To reduce the impact of external factors, the experimental setup is composed of two computers connected by a cross network cable. One of the two computers runs the **netperf** server, while the other one generates the network traffic by using a **netperf** client and runs the periodic real-time task.

The priority of the periodic real-time task has been fixed to 80 and the priorities of the IRQ threads serving the network card (the hard IRQ thread, and

the `softirq-net-rx` thread - these two threads will be indicated as “networking threads”) have been varied from 1 (minimum priority) to 99 (maximum priority). To better expose the effects of these two threads, `netperf` has been configured to use small UDP packets

IRQ Priority	< 80	80	> 80
Response Time (μs)	8431	11284	1338573
Throughput (Mbps)	66.54	67.42	81.17

TABLE 1: Response time for the Real-Time task and Network Throughput with fixed priorities

A first set of experiments indicated that assigning the same priority to the hard IRQ thread and to the soft IRQ thread gives the best results, hence the results reported in this paper are relative to priorities assigned in this way. The experiments’ results showed three different possibilities for priority assignment:

1. the networking threads have the lowest priorities in the system. This includes all the priorities from 1 to 79;
2. the networking threads have the same priority of the periodic real-time thread: 50;
3. the networking threads have a priority higher than the periodic real-time thread: this include all the priorities ranging from 81 to 99.

Table 1 summarises the results obtained in the most relevant cases. In particular, it is possible to see that when the networking threads have priority from 1 to 79 (case 1), then the worst-case response time for the periodic task is $8.431ms$ (about the execution time of the task). On the other hand, the network throughput is low (around $66Mbps$).

The response times measured in case 2 (network threads priorities equal to the real-time threads) are a little bit larger than the ones measured in case one (the worst case is $11.284ms$), and the network throughput is basically unchanged.

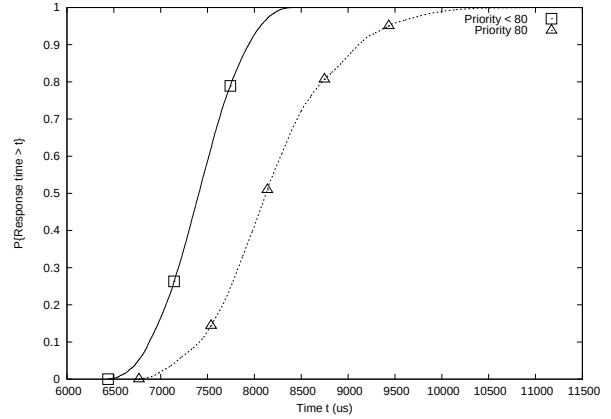


FIGURE 1: CDF of the finishing times for the real-time task, when IRQ threads have priority ≤ 80 .

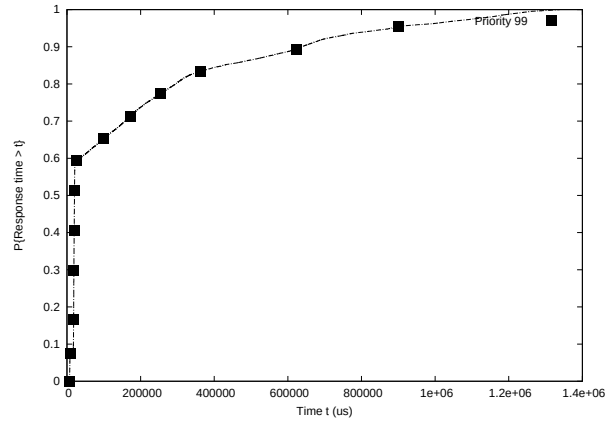


FIGURE 2: CDF of the finishing times for the real-time task, when IRQ threads have priority 99.

Further increasing the networking threads priority (case 3) increases the response times to more than $1s$, and allows to achieve a throughput of about $81Mbps$ ². The Cumulative Distribution Function (CDF) of the real-time response times in the three cases mentioned above are reported in Figure 1 and Figure 2. From the figures it is possible to notice that when the IRQ threads priority is set to 99 the kernel latencies are completely out of control.

Unfortunately, the increase in latency is not gradual, so it is not possible to assign tasks’ priorities to obtain a response time between $11ms$ and $1s$; in the same way, it is not possible to have a fine-grained control on the network throughput by only playing with priorities.

²Note that when using 300 bytes-long UDP packets this value is near to the maximum achievable throughput

As a concluding remark, while the introduction of the RT-patch leads to a significant improvement in real-time performance (see [7] for a comparison between the technology illustrated here and the mainstream technology in the Linux Kernel) the scheduling priorities prove unfit to find a good balance between the performance of different tasks. As discussed in the next section, this limitation can be overcome using a different scheduler).

3 A Possible Solution

In order to strike a good tradeoff between the performance of real-time tasks (in terms of missed deadlines) and the throughput of hardware devices, we propose reservation-based scheduling for the IRQ threads.

In particular, each IRQ thread is reserved an amount of CPU time Q^s in a reservation period T^s (this is often indicated by saying that the IRQ thread is attached to a CPU reservation (Q^s, T^s) , or is served by it); Q^s is called *maximum budget*, and T^s is called *reservation period*.

By attaching an IRQ thread to a reservation $RSV = (Q^s, T^s)$ it is possible to enforce that such thread will not demand a fraction of the CPU time larger than Q^s/T^s . In this way, it cannot increase too much the response time of real-time tasks. If a *hard reservation* behaviour is used [10] (see the CBS description below for more details about the hard behaviour), it is also possible to guarantee that the IRQ threads will not starve non real-time applications which are running in background. In plain terms, the hard reservations behaviour implements a non-work-conserving paradigm, in which the task never receives more than reserved time³. From a more formal point of view, resource reservations allow to guarantee that the worst case behaviour of a real-time task τ is not affected by the other tasks running in the system (thanks to the *temporal isolation* property).

Many different scheduling algorithms can be used to implement the resource reservations abstraction, and the reservation mechanism used in the paper is the Constant Bandwidth Server (CBS) [11]. A simplified description of the CBS algorithm follows: when a task τ is served, the CPU time consumed by it is accounted by decreasing a variable q called *budget*, and tasks are scheduled based on *scheduling deadlines* d^s assigned by the CBS (the client with

the earliest scheduling deadline is scheduled). When τ is created, q and d^s are initialised to 0. When a task τ is activated (enters the ready queue) at time t , the CBS has to assign a scheduling deadline to it. If $t > d^s - \frac{q}{Q^s}T^s$, then the latest scheduling deadline d^s (and the latest budget q) can be used; otherwise, a new scheduling deadline $d^s = t + T^s$ is generated and the budget q is replenished to Q^s . When q arrives to 0 τ is said to be depleted, and two different behaviours are possible:

- τ is not schedulable until time d^s , when the budget will be replenished to Q^s and the deadline will be postponed to $d^s + T^s$. This is known as *hard reservation behaviour*.
- the budget is immediately replenished and the scheduling deadline is immediately postponed as above; τ remains schedulable, but its priority is decreased (due to the deadline postponing). This behaviour is referred in this paper as the *soft reservation behaviour* (even if the term “firm” would be more appropriate than “soft” [10]).

For a further discussion on the CBS algorithm and on its properties, the reader is referred to the original paper [11].

Using a CBS (especially with the hard reservation behaviour) to serve IRQ threads allows to control the amount of time dedicated to device drivers, reducing their interference on real-time tasks without compromising the device throughput. To verify this property, the previous experiments have been repeated using some CBSs to schedule the IRQ threads and the periodic real-time task. In particular, the real-time task has been scheduled by a CBS with $Q^s = 8.5ms$ and $T^s = 20ms$, which guarantees that all the task instances finish in a reservation period (within $20ms$). The IRQ threads, instead, have been served using two (1, 10) CBSs. In this way, the maximum response time experienced by the periodic thread is controlled by the reservation period (and is always less than $20ms$), and the network throughput is close to the maximum value (a throughput of $79.7Mbps$ has been measured).

4 Additional Experiments

After verifying that reservation-based scheduling of the IRQ threads allows to find latency/throughput tradeoffs that are impossible with fixed priorities,

³If we care only about real-time tasks, and all of them are scheduled through appropriate reservations, then it is possible to use a work-conserving variant of the paradigm - see the “non hard” CBS algorithm.

some additional sets of experiments have been performed to test how the reservation's parameters Q^s and T^s can affect the network performance⁴.

Q^s	T^s	Throughput
940 μs	2000 μs	93.26Mbps
1410 μs	3000 μs	93.8Mbps
1880 μs	4000 μs	93.71Mbps
2820 μs	6000 μs	93.87Mbps
4700 μs	10000 μs	93.4Mbps
9400 μs	20000 μs	93.82Mbps

TABLE 2: Network throughput with different server parameters

For example, it has been verified that if the network IRQ threads are reserved enough CPU time, then the maximum network throughput can be achieved (note that a constraint on the reservation period T^s must also be respected... The correct dimensioning of the reservation's parameters depends on the number of interrupts that the network card can buffer [15]).

This set of experiments has been performed by first running `netperf` and measuring the fraction of CPU time consumed by the hard IRQ thread of the network card, which resulted to be 0.47. Then, the test has been repeated with the IRQ thread attached to a reservation (Q^s, T^s) with $Q^s/T^s = 0.47$. The results are shown in Table 2 and show that the measured throughput is always near to the maximum possible.

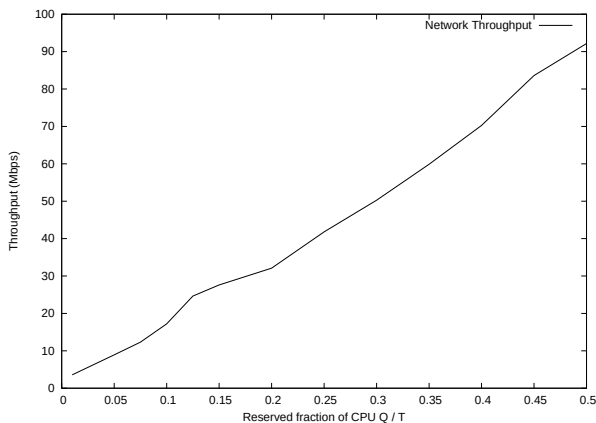


FIGURE 3: Network throughput as a function of Q^s/T^s .

By changing the ratio Q^s/T^s it is possible to control the network throughput, as verified in the next set

⁴Note that the network has been used as an example of additional hardware resource, but the presented techniques can be adapted to other devices.

of experiments, when the fraction of CPU time allocated to the network hard IRQ thread has been varied from 0.1 to 0.5. The results are displayed in Figure 3 and show that the achieved network throughput increases proportionally to Q^s/T^s until the reserved amount of CPU time is enough to properly serve the network driver.

T^s	Q^s 5ms	Q^s 10ms	Q^s 20ms	Q^s 40ms	Q^s 80ms
5ms	92,73	93,16	51,97	28,08	13,49
10ms	-	93,15	93,6	51,69	27,07
20ms	-	-	93,44	92,75	51,53
40ms	-	-	-	92,43	93,17
80ms	-	-	-	-	92,58

TABLE 3: Network throughput as a function of the reserved amount of time

Some similar results are shown in Table 3.

Finally, a last set of experiments has been performed to highlight the effects of the reservation period T^s on the network performance. Unless the reservation period is badly dimensioned causing dropped interrupts [15], it does not affect the network throughput (as shown, for example, in Table 3). However, it affects the network's responsiveness. This has been tested by adding a `ping` process in the experimental setup to measure the RTT (Round Trip Time) between the two test machines. The experimental results showed that increasing T^s can increase the maximum RTT (although the average value is not affected).

Q^s	T^s	min RTT	avg RTT	max RTT	mdev RTT
1ms	3ms	0.062	0.109	16.498	0.289
2ms	6ms	0.057	0.105	36.504	0.368
3ms	9ms	0.058	0.103	38.684	0.379
4ms	12ms	0.058	0.101	50.991	0.428
5ms	15ms	0.059	0.102	50.928	0.453
6ms	18ms	0.058	0.103	52.814	0.507
7ms	21ms	0.059	0.104	79.782	0.566
8ms	24ms	0.057	0.104	92.568	0.634
9ms	27ms	0.057	0.106	86.688	0.685
10ms	30ms	0.062	0.106	152.257	0.783

TABLE 4: Round Trip Time (in ms) as a function of the reservation's parameters

Some of the results of this set of experiments are shown in Table 4: note that while the minimum and average RTT are not affected by the server period T^s ,

the maximum RTT increases almost linearly with it. As a consequence, the standard deviation of the RTT also increases with the server period. From these experiments it is possible to see that if small network latencies are needed then the network IRQ threads should be attached to reservations with a small T^s . On the other hand, a small server period can increase the number of context switches, so using larger values is preferred (when possible).

5 Conclusions

This paper presented some results obtained when using a reservation-based scheduler for serving the IRQ threads in a real-time version of Linux. The experiments performed with periodic real-time tasks and some applications sending and receiving data over the network showed that reservation-based scheduling of the IRQ threads allows to control both the latency experienced by real-time applications and the network throughput.

More experiments are of course needed, and future tests will be extended to other hardware devices such as disks, etc...

References

- [1] *Cost and benefit of separate address spaces in real-time operating systems*, F. Mehnert, M. Hohmuth and H. Haertig, Dec 2002, Proceedings of the 23th IEEE Real-Time Systems Symposium
- [2] *Dresden Realtime Operating System*, R. Baumgartl, M. Borriß, H. Haertig, Cl.-J. Hamann, M. Hohmuth, L. Reuther, S. Schnberg and J. Wolter, Mar 1998, Proceedings of the Workshop of System-Designed Automation (SDA'98)
- [3] *Fitting Linux device drivers into an Analyzable Scheduling Framework*, T.P. Baker, A. Wang and M. J. Stanovich, Jul 2007, Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-time Applications
- [4] *Finding response times in a real-time system*, M. Joseph and P. Pandya, The Computer Journal, vol. 29, n. 5, 1986
- [5] *Modeling device driver effects in real-time schedulability analysis: Study of a network driver*, M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan and Wang, 2007, Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium
- [6] *Cooperative Scheduling of Multiple Resources*, S. Saewong and R. Rajkumar, 1999, Proceedings of the IEEE Real-Time Systems Symposium
- [7] *Providing QoS by Scheduling Interrupt Threads*, G. Modena, L. Abeni and L. Palopoli, 2008, 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008) Work in Progress.
- [8] *Scheduling algorithms for multiprogramming in a hard real-time environment*, C. L. Liu and J. Layland, 1973, Journal of the ACM
- [9] *Internals of the RT Patch*, S. Rostedt, 2007, Proceedings of the Linux Symposium.
- [10] *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems*, R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, Jan 1998, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking
- [11] *Integrating Multimedia Applications in Hard Real-Time Systems*, L. Abeni and G. Buttazzo, Dec 1998, Proceedings of the IEEE Real-Time Systems Symposium
- [12] *Implementing Resource Reservations in Linux*, L. Abeni and G. Lipari, Dec 2002, Proceedings of Fourth Real-Time Linux Workshop
- [13] <http://rt.wiki.kernel.org/index.php/Cyclicttest>
- [14] <http://www.netperf.org>
- [15] *Resource Reservations and Device Drivers: Modelling and Implementation*, N. Manica, L. Palopoli, and L. Abeni. <http://www.disi.unitn.it/~abenipubs/irqrsv.pdf>

A Flexible Scheduling Framework Supporting Multiple Programming Models with Arbitrary Semantics in Linux*

Noah Watkins

Systems Lab

UC Santa Cruz

jayhawk@soe.ucsc.edu

Jared Straub and Douglas Niehaus

Information and Telecommunication Technology Center

University of Kansas, Lawrence, Kansas 66045

iqrally@ittc.ku.edu, niehaus@ittc.ku.edu

Abstract

We present a hierarchic scheduling framework for Linux called Group Scheduling that facilitates the creation of arbitrary thread schedulers. Traditional approaches to developing new scheduling semantics require semantic mappings onto existing schedulers, such as static-priority. Group Scheduling allows for a direct implementation of semantics, allowing clear mappings at any level a developer desires. In order to effectively support scheduling semantics, integration with concurrency control is necessary (e.g. priority inheritance). However, when considering arbitrary scheduling semantics hard-wired solutions such as PI can't adapt. We present Proxy Execution as a general mechanism to resolving policy conflicts that arise as tasks from different scheduling domains interact through the RT-Mutex primitive.

1 Introduction

The recent and continuing evolution of computer systems and their applications exhibits a significant semantic explosion. The capabilities of computer systems continue to grow rapidly at the hardware level, and the range of application semantics is keeping pace. However, where hardware enhancements of the past have centered around increased chip frequencies, today we are seeing an increased degree of parallelism present in single systems. Effectively exploiting this parallelism in many instances requires that a system support diverse application semantics and reconfigurability.

Semantic diversity and reconfigurability of a single system means that the system supports multiple application semantics, that the set of semantics used by applications executing on a system can be recon-

figured, and that applications with different semantics can coexist. For example a single system should be capable of supporting relatively familiar application semantics such as deadline, rate, priority, and CPU share based semantics, in addition to experimental and emerging semantics that may include application progress or other semantics that do not map well onto traditional scheduling policies. Moreover, the set of applications on a system cannot only require a wide variety of execution semantics through reconfigurability, but in fact may require *many* or *all* of those semantics simultaneously. Single system support of multiple semantics can also reduce hardware costs by replacing several separate systems with a single physical platform that accurately supports the semantics of all applications.

In comparison to the huge amount of change in system hardware support, the semantics of the

*The work presented here was supported in part by NSF grants CNS-0716740 and CCF-0615035

programming models provided by modern operating systems has scarcely changed in the last 30 years. Threads are still typically scheduled with dynamic priority semantics¹, and concurrency is most commonly controlled using semaphores. Some expansion of semantics in scheduling and concurrency control has, of course, occurred in the past three decades. In particular, condition variables, reader-writer locks (including RCU), queuing semaphores, and monitors have expanded the concurrency control domain, while rate-monotonic analysis and EDF have expanded the availability of scheduling semantics in systems with static semantic configurations [1,2]. Nonetheless, it is fair to say that a software developer transported from 1980 would have far less difficulty recognizing and understanding the semantics of scheduling and concurrency control offered by modern operating systems than they would the hardware supporting today's mix of operating systems and applications.

1.1 The Growing Semantic Gap

The semantic gap between applications and operating systems is growing for two major reasons. First, the diversity of applications is expanding as cheap, powerful hardware begs to be used in new, imaginative ways. The second force fueling this semantic divide is the tremendous inertia the dynamic priority programming model has in modern operating systems due to widespread deployment, maturity, and simplicity. Decades of scheduling research and real-time system development have been done in environments implementing priority models, and this model is taught to all students of computer science.

While we do argue that a tremendous problem is arising with an increasingly large semantic gap, we in no way argue against the importance of priority-based programming models. On the contrary, priority programming models are by far the dominant models, and in an important sense a *victim of their own success*. Many applications exhibit priority semantics directly, and many other application semantics are easily mapped onto the priority model. Concurrency control mechanisms developed assuming priority-based scheduling are convenient and easier to develop. Finally, and perhaps most importantly, developers of new applications with new semantics often have no alternative to mapping their semantics onto priority. This lack of choice may be a result of having no access to OS source, or the prospect of developing a new scheduler may pose an

insurmountable barrier to developers because of time constraints, or lack of expertise. Bold developers sometimes use complex middleware solutions mapping application semantics onto traditional static-priority programming models. However, as application semantics become more sophisticated and diverse, the required mappings become more difficult and costly to create.

1.2 Bridging the Gap

A common pattern has emerged due to the great difficulty of changing the semantics of programming models exported by an operating system. Practitioners commonly assume that they will have to use one or more of the following techniques: (1) adaptation of their applications to use priority semantics, (2) manipulation of priorities to force the desired semantics at the user-level, or (3) using concurrency control mechanisms for their scheduling effects. These techniques have often proved to be *good enough* on dedicated systems where reasoning about a single semantics, or several simple semantics is a tractable problem. However, the explosive growth in application semantics, and the combination of semantics coexisting on a single system, is making such approaches increasingly difficult. Even worse, researchers are increasingly interested in programming models that make it *easier* to formally model and verify application behavior. Continuing to bridge the semantic gap by mixing and matching indirect methods will not scale as the resulting complexity will be too great. An alternative approach is to directly implement application semantics, allowing developers to choose the appropriate level at which semantic mappings occur.

Methods for *directly* implementing a wide range of application semantics, and for specifying how the conflicting demands of applications with differing semantics can be reconciled, would greatly simplify much of the *accidental* complexity incurred by techniques such as indirect semantic mappings. While direct implementations do not reduce the *inherent* complexity of application semantics, a direct implementation significantly reduces the complexity *added by indirect methods*, and facilitates modelability.

1.3 Group Scheduling

Group Scheduling²(GS) is a practical approach to hierarchic scheduling which emphasizes *direct repre-*

¹The introduction of the CFS scheduling class as a replacement for the SCHED_OTHER implementation is a step in the right direction, away from multiplexing many semantics into a priority-based scheduler

resentation of computation structure and *direct implementation* of application semantics [6]. For many years Group Scheduling has been used for a wide variety of projects, proving capable of describing a broad set of application semantics, and expressing system level policies such as how to balance conflicting demands from different applications. The use of GS reduces application implementation complexity compared to using indirect methods because developers are free to express semantics in ways appropriate for a given application.

This paper describes the most recent extension of Group Scheduling, which has integrated scheduling with the semantics of the RT-Mutex concurrency control primitive. This work builds upon, and significantly generalizes, the concurrency control approach used in CONFIG.PREEMPT_RT [4].

Our approach, which we call Proxy Execution, employs a general representation of mutex blocking relations, and provides hooks into which customized routines may be plugged to make decisions such as granting mutex ownership on release, and permission to steal³. The goal of Proxy Execution is to schedule owners of mutexes in such a way that desirable tasks blocked on mutexes are made runnable as soon as possible. A common solution, and the one used in CONFIG.PREEMPT_RT, is to implement the priority inheritance protocol. However, unlike PI, Proxy Execution works independently of scheduling semantics, and resolves policy conflicts that cross scheduling domains. While Proxy Execution is a simple idea, the implementation is complicated by a number of factors. These include the maintenance of blocking graphs, memory allocation requirements, and limitations on concurrency incurred during maintenance of the graph.

Proxy Execution does not yet integrate all forms of concurrency control, but we believe that the approach we have taken to the integration of scheduling with concurrency control already provides a platform within which a wide range of programming model semantics can be directly implemented, including a large number of scheduling algorithms popular in the research community that are currently implemented in specialized kernels, or in Linux using *ad-hoc* methods and evaluated with micro-benchmarks.

The availability of a common platform within which a wide range of algorithms and competing semantics could be implemented would be a significant advantage in facilitating accurate comparisons. Since each algorithm could be tested using identical

system configurations, system performance measurements can be taken to accurately reflect the characteristics of a particular algorithm or scheduling policy.

Many methods exist for collecting performance related information on Linux-based systems, both in user-space and within the kernel. Common frameworks include FTrace, LTTng, SystemTap, and Kprobes. In this paper we make reference to one such framework called DataStreams, developed and used by the KUSP research group.

The remainder of this paper is organized as follows. First we briefly explore related work. Next we provide a background and motivation for the development of Proxy Execution, and describe its implementation in Linux. Finally we provide a sampling of programming models implemented in Group Scheduling to illustrate its power and generality.

2 Related Work

The original Group Scheduling [5,6] framework was built in the 2.4 Linux kernel, and represented all computation types such as hard-IRQs, soft-IRQs, and tasklets, as members in the hierarchy. However, in this version integration of scheduling semantics with concurrency control was not addressed.

The work of [3] adapted the Group Scheduling framework to function on top of CONFIG.PREEMPT_RT, which in turn unified all computations as threads, allowing a simplification of the Group Scheduling framework's treatment of computation types. This work also introduced integrated concurrency control with a limited form of Proxy Execution that required an SDF to consider a thread blocked on a mutex. Knowledge of blocking relationships then allowed the framework to calculate the correct task to run to resolve the conflict. This form of Proxy Execution did not support scheduling algorithm optimizations such as removing a blocked task from its run-queue, nor did it address support for SMP systems.

The scheduling stack introduced at the same time as the Completely Fair Scheduler represents a first attempt at supporting multiple scheduling domains in Linux. However, the strict precedence order of scheduling domains and hard-coded assumptions of a priority scheduling model in the RT-Mutex framework are not flexible enough to support arbitrary semantics.

²Group Scheduling in this paper does not refer to the kernel facility *cgroups*, once named Group Scheduling

³stealing

Priority inheritance and priority ceiling are classic approaches to solving the problem of priority inversion [7]. The RT-Mutex in Linux implements the priority inheritance protocol [4], however neither solution effectively supports a general scheduling policy, and forces developers to map all policies onto priority, further complicating the situation when multiple policies are simultaneously active on a system.

3 Implementation

We describe our implementation of Proxy Execution, including the extensions to the RT-Mutex framework, and the integration of these extensions with Group Scheduling to support Proxy Execution. We begin with a brief overview of the Group Scheduling framework, and provide a detailed motivation for Proxy Execution. Then we describe Proxy Management, which is our set of extensions made to the RT-Mutex primitive that support Proxy Execution. Finally, we provide a detailed view of Proxy Execution and its use within the Group Scheduling framework.

3.1 Background

Group Scheduling is a hierarchic scheduling framework consisting of the following components: (1) scheduling decision functions (SDFs), (2) groups of computations, and (3) scheduling data. An SDF is an implementation of a specific scheduling semantics (e.g. static-priority or EDF), and controls the computations of a group, that is, a group represents a set of computations scheduled according to the scheduling semantics implemented by the controlling SDF. A computation is represented as a member of a group, and may be a thread or another group (allowing a hierarchic structure). And finally, arbitrary scheduling data may be associated with both groups, and members of groups.

Scheduling Decision Functions

The idea of an SDF is analogous to a *scheduling class* in the Linux kernel. Like a *scheduling class* an SDF is implemented by filling in generic function pointers. In fact, Group Scheduling has reused many of the scheduling hooks used in the implementation of the existing Linux scheduling framework.

An SDF itself is only an implementation of a scheduling semantics, and like a *scheduling class* which operates on a run-queue of tasks in a scheduling domain, an SDF too must operate with a set of

computations. Group Scheduling organizes computations into groups, where a computation may be a thread, or another group. A group must also be associated with exactly one scheduling decision function, which controls the semantics by which the group members are scheduled.

GS Hierarchy Evaluation

As discussed, groups are composed of computations, which may include other groups. This allows a set of groups to be organized in a hierarchic structure, with a single root group. When a scheduling decision is made the hierarchy is traversed starting with the root group. As the hierarchy is traversed, each group evaluates its associated scheduling decision function over its members, returning a group, a thread, or a control message such as *no decision*. This organization is very general, allowing multiple scheduling semantics to be active on a system simultaneously, but poses difficult problems for resolving conflicts between tasks as they interact through shared resources. In this paper we consider a shared resource to be represented by a semaphore, and a conflict to refer to a blocking relationship between a waiter and an owner of a semaphore.

An example hierarchy is shown in Figure 1. The root group is labeled *SEQ*, short for *sequential SDF*, and implements a semantics that chooses the first runnable task in a queue. When this hierarchy is evaluated each member is recursively evaluated until a runnable thread is found. Threads are depicted by the squares labeled *T1* through *T6*. The group labeled *Linux* represents a scheduling decision made by Linux through the evaluation of the standard Linux scheduling classes.

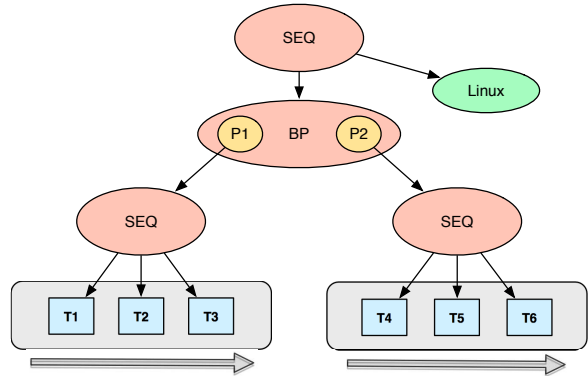


FIGURE 1: *Balanced Progress Hierarchy*

Resolving Policy Conflicts

Group Scheduling allows for a completely general organization of scheduling semantics to exist on a system. However, consider one common type of conflict that results from such a flexible configuration. Suppose a task scheduled under a static priority policy blocks on a mutex owned by a task scheduled under CPU reservation policy. Resolving this conflict in the general case is very difficult as no general decision function can be created to compare arbitrary policies. The solution that Group Scheduling provides is called Proxy Execution. Proxy Execution is a general mechanism for resolving such conflicts. However, first let us examine how Linux resolves conflicts between distinct scheduling domains.

Linux implements two main *scheduling classes*, the real-time static priority scheduler, and the completely fair scheduler. These two classes are organized in a stack with a strict ordering of importance. Specifically, the real-time class is always considered first, followed by the CFS class, and finally the class which always chooses the idle thread. In order for Linux to effectively support the overall, system scheduling policy implied by the strict ordering of scheduling domains a mechanism must be in place to resolve the conflict that result when the execution of a thread from one scheduling domain is blocked by a thread from another domain, the most important case in Linux being a real-time task blocking on a mutex owned by a task scheduled under the CFS policy.

The solution used in Linux is to implement the priority inheritance protocol, thereby integrating scheduling semantics with concurrency control. However, the priority inheritance protocol is only applicable to tasks scheduling under priority semantics, thus the PI mechanism alone cannot resolve such a policy conflict. The solution used is two-fold: (1) all tasks are capable of being moved into the real-time scheduling class (i.e. a priority field exists in the task struct), and (2) a strict separation of priority exists between threads in the real-time domain, and those in the CFS domain. Given these properties the clever solution is to simply observe alterations to a task's priority value and automatically move a task from the CFS domain to the real-time domain, and vice versa, depending on which range a priority falls into, either the real-time class or the CFS class. This mechanism utilizes the effects of the PI protocol to achieve a more complex system policy. In the general case this problem is very difficult to solve, and any solution is likely to fail if it builds on top of existing forms of integration that assume specific system

policies, such as priority.

One very important aspect of the solution implemented in Linux is worth considering, notably the *movement* of a task from one domain to another as a component of the solution to resolve policy conflicts between domains. In fact, Proxy Execution uses a similar method, but generalizes the implementation and addresses some short-comings with the existing Linux mechanism. Specifically, the movement of a task from one domain to another implies that each task be *capable* of moving, that is, have the proper data structures. However, in Group Scheduling arbitrary and dynamically allocated scheduling parameters can be associated with computations, thus the overhead of supporting a computation's direct movement between any task would require modification to all member computations as a result of updating a single group's configuration. Second, Group Scheduling places no restriction on the number of group memberships a computation may have. That is, a thread may belong to multiple scheduling domains. Obviously, the concerns are similar to that of direct task movement, but more importantly, the order of hierarchy traversal is non-deterministic, thus all memberships must be persistent.

3.2 Integrated Concurrency Control

Linux integrates priority scheduling semantics with concurrency control by hard-wiring an implementation of the priority inheritance protocol into the RT-Mutex primitive. Through priority boosting CONFIG_PREEMPT_RT implements a limited form of proxy execution, while scheduling semantics are further integrated into the RT-Mutex implementation during lock release and stealing operations by using priority specific decision functions. These two components, proxy execution and semantic integration, form what we refer to as a complete integration of scheduling semantics with concurrency control. A general complete integration thus requires a general treatment of both components.

The integration of scheduling semantics with concurrency control is illustrated by the priority specific decision functions used in the RT-Mutex implementation. During lock release the highest priority waiter is chosen to become the pending owner, and the priority of a task is again compared to a pending owner to determine if a lock can be stolen. The former operation involves a function evaluated over all waiters on a mutex, and the later is a comparison made between two tasks. Group Scheduling generalizes these decisions using hooks in the RT-Mutex implementation that can be filled in with

Group Scheduling functions that directly implement specific scheduling semantics.

A general treatment of proxy execution requires a general representation of the blocking relations that exist between tasks interacting through a mutex. In PREEMPT_RT these relations are implicitly represented through priority boosting. The nature of the priority inheritance protocol allows for a simple implementation that accumulates the maximum waiter priority at each mutex, and passes the maximum priority value onto the owner. This mechanism is attractive because of its constant memory requirements and simple internal concurrency control, but it masks individual blocking relations making it difficult or impossible to implement semantics with more complex requirements, such as CPU bandwidth limiting, which requires explicit links between a blocked task and its proxy in order to correctly perform resource accounting.

Group Scheduling uses a facility called Proxy Management to create and manage explicit representations of blocking relationships. These relationships are then used by the Group Scheduling framework to implement Proxy Execution.

3.3 Proxy Management

Proxy Management refers to the set of Group Scheduling extensions built into the RT-Mutex framework that track blocking relations between tasks. The difference between Proxy Management and Proxy Execution is that Proxy Management refers to the representation, construction, and maintenance of task-to-task blocking relations, while Proxy Execution refers to the use of this information in Group Scheduling to resolve scheduling policy conflicts.

There are two types of blocking relations tracked by Proxy Management. The first is the relation that describes a task being blocked, either directly or indirectly, on another task holding a resource. The second type of relation maintained by Proxy Management is the *proxy relation*. This relation is also a blocking relation, but represents the first conflict that must be resolved in order to "unblock" a task as soon as possible. While a task may be a part of any number of blocking relations, it is always associated with exactly one proxy relation. Intuitively a proxy relation tracks the head of a given locking chain, however in practice determining a proxy is an iterative process that involves walking the lock chain.

The implementation of Proxy Management uses two distinct data structures, one to represent generic

blocking relations, and a second to represent proxy relations. Additionally, changes to the task_struct and existing RT-Mutex data structures were necessary.

3.3.1 Blocking Graph Representation

Conceptually a blocking graph is a set of blocking relations linked together to reflect the state of a locking chain. Blocking graphs in Proxy Management explicitly represent all blocking relations in a chain, both direct and indirect, by linking together data structures representing single blocking relations. The data structure used to represent a blocking relation is the *Waiter Node*. This structure represents a blocking relation between two tasks, T_1 and T_2 , and a lock L_1 , where T_1 is the owner of L_1 and T_2 is blocked, either directly or indirectly on L_1 . Consider the graph in Figure 2. This figure depicts two indirect, and four direct, blocking relations represented by the nodes labeled $N_{2,3,4,5}$.

A *Proxy Relation* refers to the blocking relation that exists between a blocked task, and the owner of the mutex at the head of the blocked task's locking chain. This type of relation is represented by the *Proxy Waiter* structure. For example in Figure 2 four proxy relations exist between tasks $T_{2,3,4,5}$ and T_1 , the owner of the lock L_1 at the head of the chain. The *proxy relation* data structures in Figure 2 are labeled $W_{2,3,4,5}$. Each of the dashed lines in the figure represent linked lists that are used to organize the Proxy Management data structures, and their association to each other and to the existing components of the RT-Mutex framework.

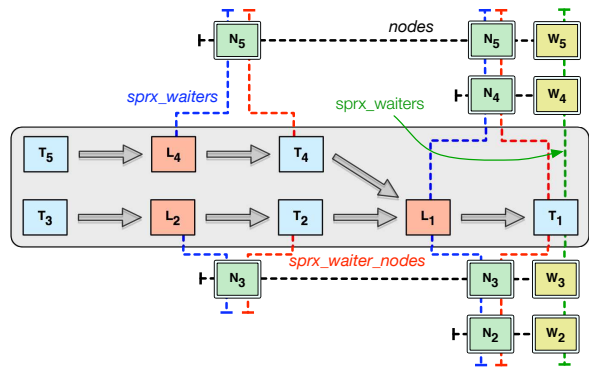


FIGURE 2: Proxy Management data structure representation

3.4 Proxy Execution

Proxy Execution is the Group Scheduling mechanism used to resolve arbitrary scheduling conflicts that arise when a task blocks on a mutex. Consider the state of the system shown in Figure 2 in which tasks $T_{2,3,4,5}$ are blocked on task T_1 . A Group Scheduling policy that selects one of these blocked tasks as the most desirable task will use Proxy Execution to resolve this conflict by instead running T_1 as a *proxy* of the desired blocked task. Notice the correspondence between task T_1 and the Proxy Relation associated with each blocked task. In fact, Group Scheduling implements Proxy Execution using the proxy relations created by the Proxy Management framework.

Proxy relations alone cannot be used to implement Proxy Execution because a proxy relation represents a scheduling policy agnostic view of a blocking relation. That is, it reflects only the task-level view of a relation. This is in conflict with the goals of Proxy Execution which by definition resolve conflicts between specific scheduling policies. Additionally, two tasks on a system can be involved in at most a single blocking relation with each other. However, Group Scheduling allows tasks to hold memberships in multiple groups. Thus, a proxy relation at the task-level corresponds to proxy relations between one or more Group Scheduling members in possibly distinct scheduling domains. Group Scheduling solves these problems through the use of *avatars*.

An avatar is a special-purpose member scheduled in-place of a specific member corresponding to a task blocked on a mutex. An avatar is created for each membership of a task associated with a Proxy Relation, and may use scheduling parameters identical to those of the task the avatar masquerades as. Proxy Execution is thus achieved by scheduling the blocking task associated with the Proxy Relation that prompted the creation of the avatar. Intuitively the use of an avatar is similar to giving a task temporary group membership under a set of different scheduling parameters. In practical terms an avatar is automatically allocated for each member because a task can only ever be associated with at most one proxy relation. In contrast, it is possible that a task may need to be scheduled as a proxy, simultaneously, from all scheduling domains.

3.4.1 Scheduling Hooks

Avatars are brought in and out of existence as proxy relations are created and destroyed. When a new avatar is created the appropriate SDF is notified by calling the *insert-member* hook, and specifying the

avatar as the member being added to the group. Likewise the generic *remove-member* hook is called to remove an avatar when a proxy relation is destroyed by the Proxy Management framework. An SDF may examine a member to determine if it is a proxy allowing it to execute any special setup or tear-down routines that are required.

3.5 RT-Mutex Extensions

This section covers the extensions made to the RT-Mutex framework that implement Proxy Management. In this section we refer to proxy relations that are created and destroyed. These terms convey heavy-weight operations, however in many circumstances the implementation is improved by re-using data structures.

3.5.1 Blocking

Blocking on a mutex is an operation that results in the creation of at least one new proxy relation. When a task blocks on a mutex a direct blocking relation is always created between blocking task and the owner of the mutex. In addition to the direct blocking relation, zero or more indirect relations are created if the blocking task has existing waiters, and when the owner of the lock being blocked on, is also blocked on a mutex. In the later case the locking chain is "walked" by iteratively examining blocking relations until an owner is found that is not blocked on a mutex. In both cases existing proxy relations are destroyed and new relations created as locking chains are extended.

For each proxy relation that is created or destroyed the scheduling framework is notified by inserting or removing an avatar associated with the affected proxy relation.

3.5.2 Releasing

The release of a mutex must be integrated with scheduling in the following ways: (1) the selection of a pending owner from among the set of direct waiters is integrated with scheduling, and (2) proxy relations must be updated to reflect the change in ownership of the mutex. First, the selection of a pending owner is implemented as a hook into the Group Scheduling framework that allows the decision function to be dynamically chosen. Second, the Group Scheduling framework must be notified of changes made to proxy relations.

Releasing a mutex is a two-step process that updates the proxy relations for all threads waiting on the mutex to be released. In the first step the proxy relations that exist between all waiters on the mutex and the owner are destroyed, including the relation involving the newly selected pending owner. Second, all of the relations except the one involving the pending owner are re-created between the original waiter associated with the relation, and the pending owner. It is important to note that proxy relations are created for a pending owner during release, as opposed to when actual ownership takes place. Even though the choice of pending owner is integrated with scheduling, no guarantee is made that this choice will *actually* be scheduled immediately. Thus, proxy relations must be created to prevent the situation in which a pending owner is never scheduled to fully acquire a mutex.

Stealing

The ability to steal a mutex is an optimization created to exploit the window of time between a pending owner being selected, and it running to fully acquire the lock. The stealing operation is integrated with scheduling in a nearly identical way to that of releasing a mutex. First, the decision to steal a lock from a pending owner must be integrated with Group Scheduling. This decision is implemented as hook that can be replaced by Group Scheduling. Its semantics are dependent on the Group Scheduling policy and configuration, and must be able to make a boolean valued comparison between tasks of potentially different scheduling domains.

When a task attempting to steal a lock is denied permission to take possession of the lock it immediately blocks on the mutex. If a task does steal a lock then it must alter any existing the proxy relations involving the pending owner. First, the stealing task destroys all proxy relations between waiters on the lock and the pending owner. Second, the stealing task re-creates all proxy relations between waiters on the lock and itself.

It is possible that a lock be stolen from an existing waiter. This is a special case taken care of by treating the stealing task no differently than another waiter. The result is a configuration of data structures that represent the new owner being a waiter on itself. However, this inconsistency is resolved before the acquiring task re-enables interrupts and releases the spinlock protecting access to the RT-Mutex.

Waiter Interruption

Interruptable mutex operations allow a task to abort acquisition due to timeout, or the delivery of a signal. However, such an interruption can cause a locking chain to be "broken" at an arbitrary location. When an interrupted task is positioned at an edge of blocking graph (i.e. it has no waiters itself) only the interrupted task's proxy relation is removed. If the interruption occurs within a locking chain the proxy relation of each waiter on the interrupted task must be updated. Specifically, the interrupted task will become the new proxy of each of its waiters, and it is no longer blocked on a mutex.

Unlike the extensions to other RT-Mutex operations, this operation will constrain concurrency within the locking chain being broken. A waiter that is interrupted examines all of its waiters, removing any blocking relation between a waiter and a task up-stream the locking chain from the point at which the chain is being broken.

4 Evaluation

In this section the Group Scheduling framework is evaluated in terms of its generality, and the performance of its implementation. It may be impossible to deliver a formal proof of complete generality for a framework such as Group Scheduling, thus we opt for a showcase of schedulers implemented within the framework. The schedulers chosen are illustrative of the wide variety of semantics supported by the framework including classical schedulers, as well as some exotic breeds. Finally we offer a brief overview of the performance characteristics of Group Scheduling.

4.1 Balanced Progress SDF

We have developed a *balanced progress* SDF that schedules its members in such way that the *progress* of each member does not exceed that of other any other members, within a certain threshold. We use the term *progress* in a general sense, and it may take on different meanings in different applications. The specific application we describe here is the balanced progress of multiple processing pipelines, where progress is defined by the number of data units processed by a pipeline. For example this may be the balanced production of video frames.

GS Hierarchy

Figure 1 shows a typical Group Scheduling hierarchy used to implement an application with balanced progress semantics. At the root of the scheduling hierarchy is the *Sequential SDF* that schedules its members in the sequence they appear within the group. This is analogous to static-priority semantics. Its use here is to provide the computations under control of the *balanced progress* SDF preference over all other system computations, denoted generally as the *Linux* group.

Each pipeline in Figure 1 is shown as a group of computations controlled under a *sequential SDF* group. Each group of computations representing a pipeline is in turn a member of the *balanced progress* group. Each of the *balanced progress* members, denoted P1 and P2, represent a pipeline whose progress is to be balanced against other members of the *balanced progress* group. The scheduling parameters associated with each member in the *balanced progress* group is an integer value representing that pipeline's progress. When a pipeline completes a unit of computation it notifies the *balanced progress* scheduler to update the scheduling state. The implementation of the scheduling decision function that forces progress to never be out of sync by more than one unit of computation is shown in Program 4.1.

Program 4.1 Balanced Progress SDF

```

1 balanced_progress_choose_next(group) {
2     if (progress_is_equal(group))
3         return choose_member(group)
4     else
5         return least_progress(group)
6     endif
7 }
```

4.2 Guided Execution

The *guided execution* programming model was created for what we refer to as *deterministic concurrency testing* in which we provide a method for *guiding* a set of processes (or threads) into specific execution states in order to create a desired interleaving. A similar technique is used by the rt-tester to test execution scenarios *within* the RT-Mutex framework that are specified as a schedule of operations. The *guided execution* programming model is a generalization of the same concept, and can be applied to arbitrary codes using schedules that cross the user/kernel space boundary.

SDF Overview

The Guided Execution SDF schedules members according to a user-specified sequence of *execution contexts*, where an *execution context* is a tuple consisting a thread identifier and an opaque context object describing a state of execution for the thread. An example of such a schedule that guides three threads into a specific interleaving is:

```

1 (top):   (Thread-A, Context-A1)
2:         (Thread-B, Context-B1)
3:         (Thread-C, Context-C1)
4:         (Thread-B, Context-B2)
```

The SDF treats the sequence of execution contexts as a stack where the top of the stack is the *target* context. The SDF executes a schedule by choosing the thread associated with the target context. Once a thread has reached its target context the stack is popped and the next thread in the sequence is scheduled. A thread is chosen *only when* it is associated with the target execution context. In this way a set of threads can be guided into arbitrary interleaving.

In the above example Thread-A is first chosen to run until it reaches Context-A1. When the target context has been reached the stack is popped and Thread-B is chosen to run until it reaches Context-B1. The schedule continues to execute until all threads are in the context associated with their last appearance in the schedule. For example, Thread-B is first guided into Context-B1, and finally reaches Context-B2 at which point the schedule has been completed.

Guided Execution Programming Model

To demonstrate the use of the Guided Execution SDF we have built a programming model around the scheduler that expresses an execution context in terms of specific locations within source code. We refer to these locations as *way-points*. A way-point is a wrapper around the Group Scheduling API that is used to inform the Guided Execution SDF of a thread's current context. Our current implementation requires that code be modified by manually inserting way-points, but nothing will prevent way-points from being inserted using automated techniques such as automatic compiler insertion.

4.3 Other Semantics

Many other semantics have been created using the Group Scheduling framework, and both existing scheduling classes in Linux, real-time static priority and CFS are portable to the Group Scheduling framework. Notable schedulers implemented in the Group Scheduling framework include an *Explicit Plan* scheduler that is similar to the *Guided Execution* scheduler, but uses an explicit schedule of execution periods placed on a time-line. For example, it is trivial to implement periodic execution of threads implementing a work-loop based programming pattern.

Another scheduler with complex semantics implemented within the Group Scheduling framework is the PTides programming model developed at UC Berkeley as part of the Ptolemy group. This scheduler implements an actor-based model where actors communicate using timestamped events. A model implements a specification of a discreet event model. The scheduler is responsible for scheduling an actor to receive an even only when an event is safe to process. Portions of the safe to process analysis can be done statically, and at run-time sensors and actuators relate model time to physical time.

5 Conclusions and Future Work

We have presented Proxy Execution, our extension to the Group Scheduling that supports a general integration of scheduling semantics with concurrency control. While Group Scheduling itself facilitates the creation of arbitrary programming models, Proxy Execution full integrates the scheduling semantics of such models with the concurrency primitives in a system, creating a truly complete solution.

The use of Group Scheduling can reduce costs by easing the implementation of complex application semantics, and increases the accuracy and understandability by using direct rather than indirect implementations. Finally, Group Scheduling can be utilized as a framework for comparison of different scheduling

algorithms and policies. Implementing new ideas in a single framework allow a fair comparison as tests can be performed using identical system configurations and hardware profiles.

Further information and links to our software can be found at the KUSP website: <http://www.ittc.ku.edu/kusp/>

References

- [1] Liu, C. L. and Layland, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment *Journal of the ACM*, 1973.
- [2] Lehoczky, J., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior *Real-Time Systems Symposium*, 1989.
- [3] Tejasvi Aswathanarayana. Integrating concurrency control & proxy execution support and provide a framework for deterministic concurrency testing under the kurt-linux group scheduling model. Master's thesis, University of Kansas, September 2001.
- [4] Steven Rostedt and Darren V. Hart. Internals of the RT Patch. *Proceedings of the Linux Symposium*, June 2007.
- [5] Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:32–43, 2005.
- [6] Michael Frisbie, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Group scheduling in systems software. *Parallel and Distributed Processing Symposium, International*, 3:120a, 2004.
- [7] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

An EDF scheduling class for the Linux kernel *

Dario Faggioli, Fabio Checconi

Scuola Superiore Sant'Anna
Pisa, Italy
{d.faggioli, f.checconi}@sssup.it

Michael Trimarchi, Claudio Scordino

Evidence Srl
Pisa, Italy
{michael, claudio}@evidence.eu.com

Abstract

The Linux kernel is mainly used as general-purpose operating system, i.e., in server and/or desktop environments. During the last years, however, academic institutions and companies showed an increasing interest in using it for real-time and control applications as well.

However, since Linux has not been designed to be a real-time operating, the best-effort scheduling policy is not suited to provide high utilization and strong guarantees to time-sensitive tasks.

We present an enhancement of the Linux scheduler through the implementation of the well known Earliest Deadline First algorithm for real-time tasks, leaving the current behavior of existing policies unchanged. It is integrated with the latest Linux scheduler, supports multicore platforms, it is available for embedded architectures (like ARM) and can be used with either periodic or aperiodic workloads.

1 Introduction

Linux is a General Purpose Operating System (GPOS) originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas. An important part of Linux is the process scheduler (or simply the *scheduler*). This component of the kernel selects which process to execute at any instant of time, and is responsible of dividing the finite resource of processor time between all runnable processes in the system.

During the last years, there has been a considerable interest in using Linux also for real-time and control, from both academic institutions and companies [15, 16]. Some reasons for this could be the free availability of its source code, the support for a

great number of architectures, a rich set of already developed device drivers and the existence of billions of applications running on it.

Unfortunately, Linux has not been designed to be a Real-Time Operating System (RTOS), thus not much attention has been given to real-time issues. Therefore, making a classical real-time feasibility study of the system under development is not possible, and developers cannot be sure that timing requirements of tasks will be met under *every* circumstance. POSIX-compliant fixed-priority policies offered by Linux, on the other hand, are not much sophisticated and often do not suit the specific application requirements.

These issues are particularly critical when designing time-sensitive or control applications (e.g., MPEG players) for embedded devices like smart-phones. In

*This work has been partially supported by the European Commission under the ACTORS project (FP7-ICT-216586).

fact, when size, processing power, energy consumption and cost are tightly constrained, you need to efficiently exploit system resources, and at the same time meet the real-time requirements of the application.

It has to be said that companies exist that started selling modified versions of the Linux kernel with improved real-time support [17, 18, 19]. However, these non-standard versions of Linux are often non-free, and can not avail themselves of the support from the huge development community of the standard kernel.

Therefore, we believe that to be really “general”, Linux should also provide enhanced real-time scheduling capabilities. In this paper, thus, we propose an implementation of the Earliest Deadline First (EDF) algorithm [1, 2], the well known real-time dynamic-priority scheduling algorithm.

The paper is organized as follows: at the beginning an overview of most notable real-time extensions for the Linux kernel proposed in the last decade is provided. Then, the current Linux scheduler is explained and the implementation of the proposed scheduling policy (`SCHED_EDF`) is provided. Last but not least, our implementation is evaluated and validated through a tests and experiments on real hardware, and finally, conclusions are driven.

2 Related work

During the last years, research institutions and independent developers have proposed several real-time extensions to the Linux kernel [4]. It must be said that, none of the extensions described in this section eventually became part of the official Linux kernel yet, thus making some of these projects obsolete.

First of all, in [20] we started investigating how an EDF policy could be implemented as a scheduling class of the new Linux modular scheduling framework, but we concentrated in uniprocessor systems, while the implementation presented here is multiprocessor. Moreover, the focus in [20] was on dealing with critical section, more than with scheduling itself.

Deadline based scheduling is also involved in [21], an even more recent work. However, what we do in there is to assign deadlines to fixed-priority task groups in the `sched_rt` class. Thus, the group to be scheduled is chosen by its deadline, while the task to be scheduled from within the group is selected according to its priority. On the other hand, in this work, a new scheduling class is utilized, and tasks

have their own deadlines.

2.1 OCERA

A real-time scheduler for Linux 2.4 has been developed within the OCERA European project, and it is available as Open Source code [3, 7, 8]. To minimize the modifications to the kernel code, the real-time scheduler has been developed as a small patch and an external loadable kernel module. All the patch does is exporting toward the module (by some *hooks*) the relevant scheduling events.

The approach is straightforward and flexible, but the position where the *hooks* have to be placed is real challenge, and it made porting the code to next releases of the kernel very hard.

2.2 AQuoSA

The outcome of the OCERA project gave birth to the AQuoSA [23] software architecture. It basically consists on the porting of OCERA kernel approach to 2.6 kernel, with a user-level library for feedback based scheduling added.

Unfortunately, it lacks features like support for multicore platforms and integration with the latest modular scheduler [5, 6] and the cgroups filesystem.

2.3 Frescor

A real-time framework based on Linux 2.6 has been proposed by the Frescor European project [9]. It is based on AQuoSA and further adds to it a contract-based API and a complex middleware for specifying and managing the system performances, from the perspective of the Quality of Service (*QoS*) it provides.

Obviously, it suffers from all the above mentioned drawbacks as well.

2.4 LITMUS-RT

LITMUS-RT [10] is a plugin based scheduling framework with a wide variety of implemented real-time scheduling algorithms. The aim of the project, at least for now, is the evaluation of multiprocessor scheduling algorithms and synchronization protocols from research point of view.

Moreover, it only runs on Intel (x86-32) and Sparc64 architectures (i.e., no embedded platforms, the one typically used for industrial real-time and control).

2.5 RTLinux, RTAI, Xenomai

The so called *interrupt abstraction* approach [4] actually allows to schedule even *hard* real-time tasks on Linux. It consists of having an abstraction layer of virtual hardware below the Linux kernel, resulting in some kind of multithreaded RTOS with standard Linux and all its processes running as the lowest priority thread.

This approach has been successfully implemented in several existing frameworks, the most notable examples of which are RTLinux [12], RTAI [11] and Xenomai [13].

It is an efficient solution, as it allows to obtain very low latencies, but is also invasive, and, often, not all standard Linux facilities are available to tasks running with real-time privileges.

3 Modular Scheduling Framework

Recently (since release 2.6.23) the previous $O(1)$ Linux scheduler has been replaced by a completely new modular scheduler implemented by Ingo Molnar [5, 6]. In this section we will try to point out some of its more important characteristics.

3.1 Scheduling classes

This “new” scheduler has been designed in such a way to introduce *scheduling classes*, an extensible set of scheduler modules. These modules encapsulate specific scheduling policies details that the core scheduler needs not to know. The binding between each policy and the related scheduler is done through a set of *hooks* (i.e., function pointers) provided by each scheduling class and called by the core scheduler.

Currently, Linux comes with two scheduling classes¹:

- **sched_fair**: the “*Completely Fair Scheduler*” (CFS) algorithm, for `SCHED_NORMAL` and `SCHED_BATCH` policies. The idea here is to run tasks in parallel and at precise weighted speeds, in, so that each task receives a “fair” amount of processor share;
- **sched_rt**: the POSIX fixed-priority real-time scheduling, for `SCHED_FIFO` or `SCHED_RR` policies with 99 priority levels.

¹actually, a special idle scheduling class also exists to implement the idle task

²CFS act according to a global, fixed period

A (partial) list of the hooks a scheduling class may provide, by filling them in its own `struct sched_class` structure, follows:

- **enqueue_task(...)**: enqueues a task in the data structure used to keep all runnable tasks (runqueue); usually called when the task enters a runnable state;
- **dequeue_task(...)**: removes a task from the runqueue; usually called when the task stop being runnable;
- **yield_task(...)**: yields the processor for other tasks to have a chance to be run;
- **check_preempt_curr(...)**: checks if a task shall preempt the currently running task;
- **pick_next_task(...)**: chooses the most appropriate task eligible to be run next;
- **put_prev_task(...)**: makes a running task no longer running;
- **select_task_rq(...)**: chooses on which runqueue (i.e., on which CPU) a waking-up task has to be enqueued;
- **task_tick(...)**: accounts each periodic system tick to the running tasks.

3.2 Limits of current scheduling classes

Real-time tasks are *computational activities characterized, at least, by a worst case execution time and a timing constraint* [2]. These tasks must be executed by the RTOS in a correct order, so that each one completes within its timing constraints. A typical constraint is the *deadline*, i.e., is the instant the task execution is required to be completed, otherwise the results could turn out to be useless.

The default scheduling policies of Linux (i.e., `SCHED_NORMAL` and `SCHED_BATCH`) cannot provide the guarantees a time-sensitive application may require. This is mainly because no concept of timing constraint (e.g. deadline) can be associated to a task in them. In fact, although it is possible to assign a share of the processor time to a task (or a group of tasks)², there is no way to specify that a task must execute for 20msec within 100msec, as it is possible with real-time scheduling algorithm, such as EDF. Moreover, the time elapsed between two consecutive executions of a task is not deterministic and can not

be bound, since it highly depends on the number of tasks running in the system at that time.

Even POSIX-compliant fixed-priority policies (i.e., `SCHED_RR` and `SCHED_FIFO`) diverges from what the real-time research community refer to as “real-time” [2, 14]. For instance, as best-effort policies, they do not allow to assign timing constraints to tasks. Moreover, it is well known [1] that they provide lower performances in term of both schedulability guarantees [?] and flexibility (e.g., graceful degradation and bounded tardiness), both on uniprocessor and multiprocessor systems.

4 SCHED_EDF: features and implementation

In this section we present our new scheduling class, `sched_edf`, implementing the `SCHED_EDF` scheduling policy. The purpose of this paper is to give as much details as possible about this first implementation of the new policy.

As we will point out, thorough experimental evaluation to validate, and maybe correct, some of the design choices we made, is the subject of ongoing and future work by us.

All the information about the code, together with links to installation and usage instructions can be found at http://www.evidence.eu.com/sched_edf/

4.1 Main features

Our scheduling class implements the Earliest Deadline First (EDF [?]) algorithm and uses the Constant Bandwidth Server (CBS [22]) to provide bandwidth isolation among tasks. They both are well known and discussed in the real-time community, thus we are not going into too much details of them here, for space reasons.

This scheduling class has been developed from scratch, without starting from any existing project, taking advantage of the modularity currently offered by the Linux scheduler.

The implementation is aligned with the current mainstream kernel, and it relies on standard Linux mechanisms (e.g., control groups) to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

4.2 Interaction with existing policies

The addition of our scheduling class to the Linux kernel does not change the behavior of the existing scheduling policies, neither best-effort and real-time ones, as shown in Figure 1).

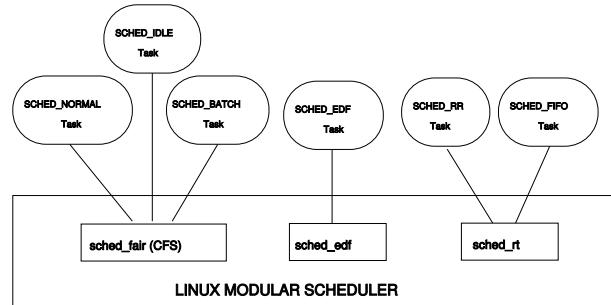


FIGURE 1: *Linux scheduler with SCHED_EDF.*

However, given the current Linux scheduler architecture, there is some interaction between scheduling classes. In fact, since each class is asked to provide a runnable task in the order they are chained in a linked list, “lower” classes actually run in the idle time of “upper” classes.

We decide to put our new scheduling class, `sched_edf`, in between the best-effort scheduling class, `sched_fair`, and the POSIX real-time scheduling class `sched_rt`. This means a ready EDF task always “win” the struggle for the CPU against a best-effort task, but always “loose” it against a fixed-priority POSIX real-time one. This has been done for the following reasons:

- backward application compatibility and (POSIX) standard compliance require `SCHED_FIFO/RR` tasks to run as soon as they can;
- it may be useful, either for the user or for the kernel (e.g. during system power down or CPU offlining), to have an easy way to specify an “always winner” task, without the need of dealing with the deadlines that are present in the system at that particular instant;
- since EDF is more efficient than fixed-priority in term of schedulability, it will be easier for it to achieve high utilization even if it runs in the idle time of fixed-priority, than the vice-versa;
- real-time analysis techniques already exists to deal with exactly such scheduling architecture.

It should be easy to notice that this solution has the drawback that EDF tasks will suffer from the

interference of fixed-priority tasks, but it has to be said that:

- the amount of interference coming from the user's `SCHED_FIFO/RR` tasks can be taken into account given the techniques cited before;
- the amount of interference coming from Linux system `SCHED_FIFO/RR` tasks can be accounted for as system overhead;
- in recent kernels, the bandwidth the whole `sched_rt` scheduling class occupies, can be forced to stay below a certain limit ³.

Experimental evaluation of the overhead and the experienced latency under different load conditions, the actual system schedulability and predictability, etc., is deferred to future works.

4.3 Multiprocessor scheduling support

In the real-time and scheduling community global and partitioned multiprocessor scheduling schemes exists. In global scheduling each task may be picked up and run on each CPU; in partitioned scheduling a task has its own CPU where it is always scheduled. Discussing advantages and drawbacks of both, is not in scope here, let us just say that the global case is usually thought and implemented by means of one global ready queue for all the CPUs, partitioning by one ready queue per CPU.

The way Linux deals with multiprocessor scheduling is often called *distributed runqueue*, which means each CPU as its own ready queue, as in partitioning; however, tasks can, if wanted or needed, migrate between the different queues. This adds the overhead of migrations but, at least, make it easy to pin some task on some processor (*scheduling affinity*) and, more important, addresses the serious locking scalability issues that –it is easy to figure out– concerns the single-runqueue solutions.

We are interested, for our new scheduling class, in a general solution, i.e. a globally scheduled system in which it is easy to ask one or more tasks to stay on a pre-specified CPU (or set of CPU). Therefore, the easiest and simplest way of achieving this is to go for the same approach of Linux itself, which means:

- we have one runqueue for each CPU, implemented with a *red-black tree*, for efficient manipulation;

- we migrate the tasks among the runqueues in a way such that:
 - we always try to have, on an m CPU system, the m earliest deadline ready tasks running on the CPUs;
 - we always respect the affinity the tasks specify.

Therefore, if the affinity of a particular task is equal to only one precise CPU, that task will never be migrated and, when runnable, it always run only on it. Thus, it is sufficient that each EDF task in the system has its affinity correctly set, to achieve what we called partitioned scheduling, with very few overhead.

4.4 Tasks scheduling

The `sched_edf` scheduling class does not make any restrictive assumption on the characteristics of its task, thus, it can handle:

- periodic tasks, typical in real-time and control applications;
- sporadic tasks, typical in soft real-time and multimedia applications;
- aperiodic tasks.

A key feature of task scheduling in this scheduling class is that *temporal isolation* is ensured. This means, the temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of any other task in the system. In other words, even if a task misbehaves, it is not able to exploit larger execution time than it has been allocated to it and monopolize the processor.

In fact, each task is assigned a *budget* (`sched_runtime`) and a *period*, considered equal to its *deadline* (`sched_period`). This means the task is guaranteed to execute for an amount of time equal to `sched_runtime` every `sched_period` (task *utilization* or *bandwidth*). When a task tries to execute more than its *budget* it is slowed down, by stopping it until the time instant of its next deadline. When, at that time, it is made runnable again, its budget is refilled and a new deadline computed for him. This is how the CBS [22] algorithm works, in its hard-reservation configuration.

However, although the CBS algorithm is very effective to encapsulate aperiodic or sporadic –real-time or non real-time– tasks in a real-time EDF

³which is something we are trying to improve as well [21]

scheduled system, it imposes some overhead to “standard” periodic tasks. Therefore, we make it possible for periodic task to specify, before going to sleep waiting for the next activation, the end of the current instance. This avoid them (provided they behave well!) being disturbed by the CBS.

4.5 API for tasks

The existing system call `sched_setscheduler()` has not been extended, because of the binary compatibility issues that modifying its `struct sched_param` parameter would have raised for existing applications. Therefore, another system call, called `sched_setscheduler_ex(...)` is implemented. It allows to assign or modify the scheduling parameters described above (i.e., `sched_runtime` and `sched_period`) for tasks running with `SCHED_EDF` policy. The system call has the following prototype:

```
#define SCHED_EDF 6
struct sched_param_ex {
    int sched_priority; /* Backward compliance */
    struct timespec sched_edf_period;
    struct timespec sched_edf_runtime;
};

int sched_setscheduler_ex(pid_t pid, int policy,
    struct sched_param2 *param);
```

For the sake of consistency, also `sched_setparam_ex(...)` and `sched_getparam_ex(...)` have been implemented.

On the other hand, we did not add any system call to allow a periodic task to specify the end of its current instance, and the semantic of `sched_yield(...)` for EDF tasks has been modified for that purpose.

4.6 API for task groups

The scheduling class has also been integrated with the control groups mechanism in order to allow the creation of groups of tasks with a cap on their total utilization.

Therefore, the cgroups interface now provides two new entries, `cpu.edf_runtime_us` and `cpu.edf_period_us`. These files are created once the cgroup filesystem is mounted, to get and set the group bandwidth. In fact, a group has not actual budget or period/deadline, and these values are used to derive the utilization of the group. Consistency check is run when:

- a task is created or moved inside a group,
- the parameters of a task (if inside a group) are modified,
- a group is created or moved inside another group,
- the parameters of a group (if inside another group) are modified,

to check if the cumulative utilization of tasks and groups is below the one of the group that contains them (i.e., their *parent* group).

On multicore platforms, tasks and task groups can be moved among different processors using existing Linux mechanisms, i.e., `sched_setaffinity(...)` for tasks (either threads or processes), and `cpuset.cpus` for task groups.

5 Evaluation

Given the preliminary status of this work we report here only a very simple example showing our new policy at work.

First of all we modified the `schedtool` package to make it use the new `sched_setscheduler_ex(...)` system call, and thus be able to set `SCHED_EDF` as the scheduling policy of a task, with proper *budget* and *period*. We then ran a simple `yes` Linux process as a `SCHED_EDF` task with 100ms `sched_period` and 20ms `sched_runtime`. The actual schedule is obtained by using the `sched_switch` tracer from the `ftrace` infrastructure, recently introduced into the kernel. Finally, the output has been converted to VCD format by means of the `analyze`⁴ program. Thus, we can use `GtkWave` to visualize what happens in the system. Notice that all the tools and programs mentioned above, are freely available and easy to install and use.

We can see what happens on the CPU where our `SCHED_EDF yes` process is being run in Figure 2).

Green “low” lines correspond to task execution, while yellow “high” ones to task being not runnable. Red vertical lines happens when a task become ready to run, and thus dark red rectangle means the task is ready but not running, usually because some other task of higher priority or scheduling class is keeping it out from the CPU. The `yes` EDF task is at the second line in the Figure; other lines represent other tasks running on the same CPU, as shown in the `Signals` box on the left. All these tasks but

⁴<http://www.osadl.org/Visualize-the-temporal-relationship-of-L.taks-visualizer.0.html>

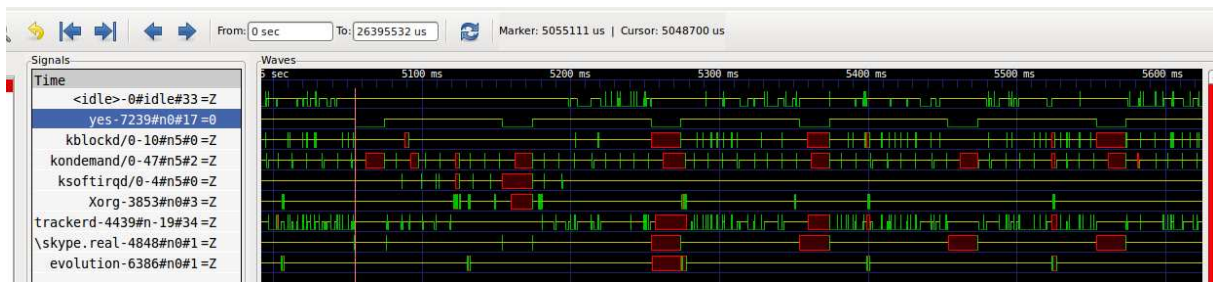


FIGURE 2: Scheduling trace with a *SCHED_EDF* task

yes, either being user or system tasks, run with *SCHED_NORMAL* as their scheduling policy.

Looking at the figure we notice at least the following:

- our *sched_edf* task runs as soon as it become runnable, preempting or preventing to be run all the best-effort tasks;
- thanks to the bandwidth isolation enforcing mechanism, even if it would like to run continuously, it is allowed to only exploit its full *sched_runtime* every *sched_period*, i.e., 20ms every 100ms.

6 Conclusions

In this paper, we presented an implementation of the Earliest Deadline First (EDF) algorithm for the Linux kernel.

With respect to similar work proposed in the past, our implementation takes advantage of the modularity offered by the new Linux scheduler, leaving the current behavior of existing policies unchanged. The new scheduling class is integrated with the latest Linux scheduler, and relies on standard Linux mechanisms (e.g., control groups) to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

Our implementation does not make any restrictive assumption on the characteristics of the task. Thus, it can handle periodic, sporadic and aperiodic tasks. Another interesting feature of this scheduling class is the *temporal isolation* among the tasks handled. The temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of the other tasks: if a task misbehaves and requires a large execution time, it cannot jeopardize the processor. Thanks to the temporal isolation

property, each task executes as it were on a slower dedicated processor, so that it is possible to provide real-time guarantees on a per-task basis. Such property is particularly useful when mixing hard and soft real-time tasks on the same system.

References

- [1] C.L.Liu, J.W.Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association for Computing Machinery, 1973.
- [2] John A. Stankovic, K.Ramamritham, M.Spuri, G.Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, Kluwer Academic Publishers, 1998.
- [3] L.Abeni, G.Lipari, *Implementing Resource Reservations in Linux*, Real Time Linux Workshop, 2002.
- [4] G.Lipari, C.Scordino, *Linux and Real-Time: Current Approaches and Future Opportunities*, IEEE International Congress ANIPLA, 2006.
- [5] *Design of the CFS scheduler*, Linux documentation, linux/Documentation/scheduler/sched-design-CFS.txt.
- [6] I.Molnar, *Modular Scheduler Core and Completely Fair Scheduler*, <http://lkml.org/lkml/2007/4/13/180>
- [7] I.Ripoll, P.Pisa, L.Abeni, P.Gai, A.Lanusse, S.Sergio, B.Privat, *WP1 — RTOS State of the Art Analysis: Deliverable D1.1 — RTOS Analysis*, OCERA, 2002.
- [8] C.Scordino, G.Lipari, *A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks*, IEEE Transactions on Computers, 2006.

- [9] *FRESCOR: Framework for Real-time Embedded Systems based on COnTRACTs*, <http://www.frescor.org>
- [10] J.Anderson and students, *LITMUS RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*, <http://www.cs.unc.edu/~anderson/litmus-rt/>
- [11] L.Dozio, P.Mantegazza, *Real-Time Distributed Control Systems Using RTAI*, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.
- [12] V.Yodaiken, *The RTLinux Manifesto*, Fifth Linux Expo, 1999.
- [13] P.Gerum, *The XENOMAI Project, Implementing a RTOS emulation framework on GNU/Linux*, 2002.
- [14] G.Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Springer, 2005.
- [15] G.Francis, L.Spacek, *Linux Robot with Omnidirectional Vision*, TAROS06, 2006.
- [16] F.Kanehiro, K. Fujiwara, S.Kajita, K.Yokoi, K.Kaneko, H.Hirukawa, Y.nakamura, K.Yamane, *Open Architecture Humanoid Robotics Platform*, IEEE International Conference on Robotics and Automation, Washington DC, 2002.
- [17] Windriver Linux, <http://www.windriver.com/products/linux/>
- [18] Timesys Embedded Linux kernel, <https://linuxlink.timesys.com/3/Linux>
- [19] Montavista Real-Time Linux, <http://www.mvista.com/real.time.linux.php>
- [20] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, A. Mancina, *An Implementation of the Earliest Deadline First Algorithm in Linux*, 24th ACM SAC, March 2009.
- [21] Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari, *Hierarchical Multiprocessor CPU Reservations for the Linux Kernel*, 5th OSPERT Workshop, July 2009.
- [22] L. Abeni, G. Buttazzo, *Resource Reservation in Dynamic Real-Time Systems*, Real-Time Systems, Vol. 27, No. 2, pp. 123-167, July 2004.
- [23] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, *AQuoSA - Adaptive Quality of Service Architecture*, Software: Practice and Experience, April 2008, doi 10.1002/spe.883 Expand Abstract.

Towards Linux as a Real-Time Hypervisor

Jan Kiszka

Siemens AG, Corporate Technology, CT SE 2
Otto-Hahn-Ring 6, D-81739 Munich
jan.kiszka@siemens.com

Abstract

Combining virtualization and real-time is important for an increasing amount of use cases, from embedded system to enterprise computing. In this paper, we will analyze the real-time capabilities of Linux as a hypervisor when using KVM and QEMU. We will furthermore introduce and evaluate a paravirtual scheduling interface that helps resolving priority inversion problems in embedded virtualization scenarios.

1 Introduction

Combining virtualization and real-time requirements is so far primarily a topic in embedded systems. In this domain, a general purpose operating system (GPOS) often needs to be executed aside a real-time operating system (RTOS). A hypervisor may then be used to manage shared resources and isolate the OSes from each other. Such architectures can be found in industrial control systems where the RTOS takes over the time-critical control of a machine while the GPOS runs, for example, the visualization software. Further examples are single-processor smartphones where an RTOS is used to manage critical tasks of the radio communication while a GPOS hosts the typical set of mobile phone applications.

With the increasing importance of real-time in enterprise computing scenarios, the overlap with virtualization becomes larger as well. In the absence of highly demanding real-time requirements, virtualization is already used for server consolidation and load balancing. Thus, improving real-time capabilities of virtualization solutions will extend their usability also into the real-time enterprise domain.

But the gap between embedded real-time and enterprise virtualization is still large. In many embedded scenarios, specialized hypervisors are used that have a reduced feature set, are mostly statically configured, but can provide better performance and require less resources than full-featured versions. Hypervisors from the enterprise domain focus on good average performance as well as resource control at a comparatively high level. They manage significantly larger resource sizes and include mechanisms to dy-

namically shift load between physical nodes.

Linux as an operating system has already demonstrated that it can adopt to a very broad range of use cases: from low-end embedded up to large-scale machines with thousands of CPU cores [1]. In the real-time domain, Linux is currently approaching more demanding use cases with the PREEMPT-RT patch [2, 3]. And Linux includes built-in virtualization support with the KVM subsystem [4, 5]. Consequently, one question is what quality of service could already be achieved by combining these technologies—which furthermore include the advantages of open source, thus are also easily adaptable to specific requirements whenever needed. A realistic question is not if Linux can solve everything, but how large the share in use cases is that it can cover and how it can be gradually extended so that all Linux users benefit.

2 Linux as Hypervisor

Using the Linux OS itself as a hypervisor requires two components: A kernel driver and a user space application. Both will be briefly introduced in the following.

2.1 Kernel-Based Virtual Machine

The Kernel-Based Virtual Machine (KVM) is a Linux device driver whose main task is to manage unprivileged access to virtualization features that can only be used directly by the privileged kernel. It was originally developed to grant this access to Intel's and AMD's x86 hardware-based virtualization extensions, but meanwhile provides essentially the identi-

cal interface on ia64, s390, and PowerPC hosts—even though not all of them provide comparable hardware virtualization support.

Another important task of KVM is to provide fast virtualization for frequently accessed guest devices. So far, interrupt and timer controllers are available as kernel-hosted models. The advantage is that no consultation of the controlling user space process is required if a guest accesses any of these devices, which reduces the virtualization overhead. In-kernel support for paravirtualized network adapters is currently under development.

KVM is a so-called type-2 hypervisor [6], that is it makes use of the host OS for tasks like setup/cleanup, scheduling, native interrupt and memory management etc. For this reason, the execution model of KVM is very simple: A Linux user space process acting as a hypervisor can request to create and configure a virtual machine (VM) by opening the KVM character device and issuing a series of IOCTLs. This VM shares the host memory map with its controlling process, while the hypervisor can freely configure the mapping between the VM's physical and its own virtual memory.

The hypervisor process can furthermore request to create virtual CPUs within the VM. Given a guest CPU state, they can then execute arbitrary guest code in a confined environment. Any guest activity that cannot be handled by the host CPU's virtualization extension will trigger an exit from the virtual CPU (VCPU) execution environment. If the KVM kernel code is not able to handle the exit internally, for example by swapping in a host memory page that the guest has requested by accessing a virtual page, the exit event is propagated to the user space hypervisor.

The execution context of the VCPU is implicitly defined by the user space hypervisor: It invokes the VCPU execution request from within one of its threads, and KVM preserves this scheduling context while running the guest code. Thus, if the Linux host scheduler decides to switch out a hypervisor thread, it is the corresponding guest CPU which is effectively scheduled out. The hypervisor process and/or the system administrator can therefore apply standard means at the level of threads, processes, users or control groups to influence the scheduling of virtual machines as well as further host services.

Guest exits also take place on asynchronous host events, foremostly interrupts. As soon as KVM has restored the host OS context, it re-allows host interrupt processing, thus the corresponding Linux interrupt handler is able to run. If the interrupt handler creates the need to reschedule, KVM detects this and invoke the corresponding service that possibly

switches to a different host task.

2.2 QEMU

QEMU [7] is a fast CPU and peripheral hardware emulator. Its emulation does not depend on specific host features, that is it also works across architectures. The KVM project chose QEMU as the foundation for its user space hypervisor as it already came with most of the additionally required infrastructure like I/O device emulation.

KVM's QEMU version basically replaces the CPU emulation by the hardware-assisted execution provided by the kernel driver. It furthermore splits the single-threaded execution model of QEMU into VCPU threads and a main I/O processing loop, enabling more scalable SMP virtualization. As the QEMU core code is not yet prepared for multi-threaded execution, a global lock protects any access to the device emulation layer. Thus, only one VCPU or the I/O main loop may query or modify emulated devices of a virtual machine at the same time.

The KVM project also adds asynchronous I/O (AIO) support to QEMU. As Linux's native AIO implementation performs worse than a pthread-based approach, QEMU gained AIO via a thread-pool: for each AIO request a user space thread is spawned or obtained from the pool that executes the request and then signals the completion to the main I/O thread.

Although the upstream QEMU version meanwhile includes KVM support, we developed our modifications only for KVM's QEMU branch [8, commit f2593a0] as it includes some required features that have not yet been merged into upstream.

3 Prioritizing QEMU/KVM

A straightforward approach to improve guest responsiveness over QEMU/KVM is to raise the priorities of QEMU's threads and lift them into a real-time scheduling class. This comes with the risk of starving host services that the guest may indirectly busy-wait upon. For example, the guest could issue an I/O request that is handled asynchronously by the host, but requires completion within the context of non-real-time service like the kernel's event or AIO thread.

This risk can be mitigated by avoiding CPU over-commitment, that is running less VCPUs than real processor cores exist. Furthermore, Linux comes with a real-time throttling mechanism [9] that restricts the CPU slice of all real-time tasks in the host system. The latter approach, however, can result in very low performance as the guest system consumes a lot of CPU time without making relevant progress.

3.1 Realization

While QEMU's main I/O thread and its VCPU threads could also be adjusted after start-up via `chrt`, its AIO thread pool grows and shrinks dynamically. Therefore a QEMU extension was required to establish a stable prioritization of all involved threads. We defined a new command line switch for QEMU:

```
-rt maxprio=prio[,policy=ff|rr|ts]
    [,aioprio=prio][,aiopolicy=ff|rr|ts]
```

If `aioprio` and `aiopolicy` are omitted, the scheduling policy is applied to all existing and future threads of the QEMU process. QEMU's main thread and its AIO threads then gain the specified maximum priority, while VCPU threads will use the next lower priority level. This ensures that a spinning guest cannot starve itself from receiving I/O events that are routed through those threads. However, if asynchronous I/O is not considered most important for a guest, the priority and policy of corresponding completion threads can be separately specified. The effect is discussed in Section 3.3.

We furthermore added

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

in case a real-time scheduling class is selected. This prevents any swapping of QEMU's process memory, including guest pages, and avoids lazy mapping. While `mlockall` and memory overcommitment are generally not compatible, real-time requirements deny the latter in most cases anyway.

A further real-time enhancement for QEMU concerned switching mutexes used for synchronizing I/O handling to the priority inheritance protocol. The impact of this change, however, can be considered minor at this point, as it is pointed out in Section 6.

3.2 Test Setups

Our test setup consisted of a host system with an Intel dual-core processor (Core2 T5500 at 1.6 GHz) with 2 GB of RAM. Host as well as guest systems were based on OpenSUSE 11.1. We started our evaluation with a host kernel built out of KVM's git tree [10, commit 0b972aa], which basically generated a 2.6.31-rc4 kernel. We enabled `CONFIG_PREEMPT` and `CONFIG_PREEMPT_RCU` and disabled `CONFIG_ACPI_PROCESSOR` for best latencies. Tracing facilities were built in, but remained disabled during latency tests (`echo 0 > /proc/sys/kernel/ftrace_enabled, echo 0 > /sys/kernel/debug/tracing/tracing_enabled`).

To limit the size of the test matrix, we decided to focus on a single real-time aspect in the following evaluations: the latency of timed high-priority user space threads in the guest system. For measuring this latency, we used `cyclictst` from the `rt-test` project [11, v0.50] with the following parameters: `-m -n -p 99 -l 1000000 -h 100000 -q`, that is a 1 ms periodic nanosleep test over 1 million iterations. As our kernel configuration allowed both host and guest system to use the native TSC as clock source, we were able to rely on the measurement results obtained within the guest. Moreover, our interest in the latency distribution was stronger than in a reliable upper bound for the worst case. We therefore ran all tests only for 15 minutes, which is typically not enough to obtain maximum latencies but sufficient to compare distributions.

The guest kernel was kept identical for all tests: 2.6.29.6-rt23 plus paravirtual scheduling patches for Linux guests that are introduced in Section 5. The latter extensions were always enabled as Linux disables them automatically during boot-up if the hypervisor does not provide this feature.

QEMU was started with the parameters

```
qemu-system-x86_64 -m 512 OpenSuse_64.raw \
    -nographic -serial /dev/null -net nic \
    -net user,hostfwd=::2222-:22
```

that is using 512 MB RAM, a raw disk image and no local console. Instead we controlled the guests via a forwarded SSH connection. The reasons for choosing a raw image over QEMU's QCOW2 disk format and avoiding local graphic or even serial console emulations is discussed in Section 6.

The tests on PREEMPT-RT host kernels required us to bind prioritized QEMU to the second core of our host CPU as we otherwise faced live-locks during boot-up. We did not analyze the reason for this in further detail, but instead decided to establish this CPU binding consistently for all tests. The binding to the second core was not only applied to the QEMU process but also to the load applications as well as the host's disk interrupt. A positive side-effect of moving the tests was that we were able to avoid occasional latency peaks of a few hundred microseconds due to system management interrupts (SMI) which apparently only hit the first core on our platform.

We defined two load applications that were running in parallel in endless loops during the tests: `bonnie 1.4` (as provided by OpenSUSE 11.1) was executed with `-y -s 2000` to stress the I/O subsystems. Furthermore, the cache calibrator [12] was run with `1660 4M /tmp/calibrator.log`, that is a 4 MB RAM test area to stress memory caches. For

all tests, the calibrator load stayed on the host as the intended cache pollution can be triggered from both host and guest domain equally well.

3.3 Evaluation

To evaluate the effect of raising QEMU's thread priorities, we first measured the latencies natively achievable with `cyclicttest` on the host. For this test, the `bonnie` load ran on the host, and no QEMU instance was started. Figure 1 shows the result for the `CONFIG_PREEMPT` host kernel we used in this first round. Note that the axes in all graphs are logarithmically scaled.

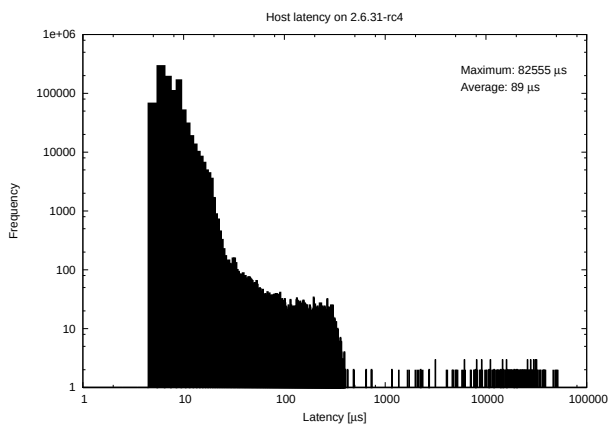


FIGURE 1: *Host Latency*

Next we measured the guest latency without any QEMU prioritization. `Bonnie` now ran inside the guest. In Figure 2, the difference to the host latency is visible in form of a shifted frequency maximum and a much higher probability of multi-millisecond latencies. In fact, the selected upper bound for the histogram of 100 ms did not catch the maximum latencies. Clearly, this setup is unsuited for latency sensitive guest load.

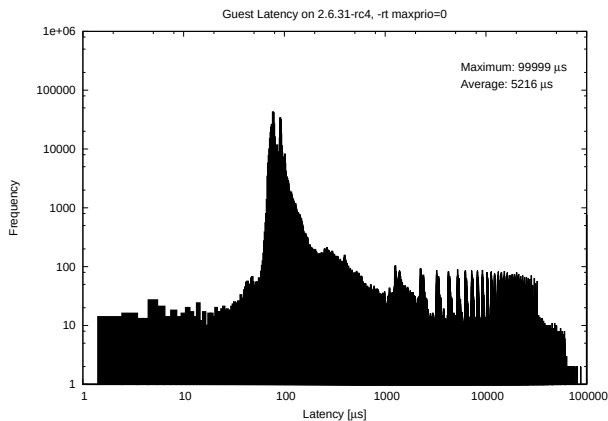


FIGURE 2: *Unprioritized Guest Latency*

We then applied `SCHED_FIFO` with a priority of 98 for I/O and 97 for VCPU threads on QEMU. The result is depicted in Figure 3: the average latency dropped significantly.

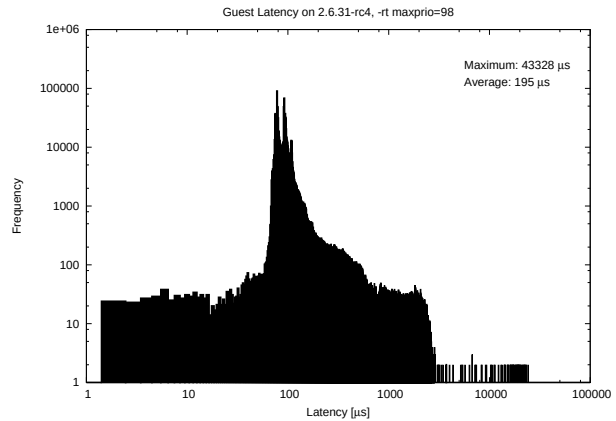


FIGURE 3: *Prioritized Guest Latency*

In the aim to reduce the I/O load impact on our test scenario, we moved QEMU's AIO threads out of the real-time scheduling class again. Figure 4 visualizes that this kept the overall distribution shape but reduced the probability of high or low latencies.

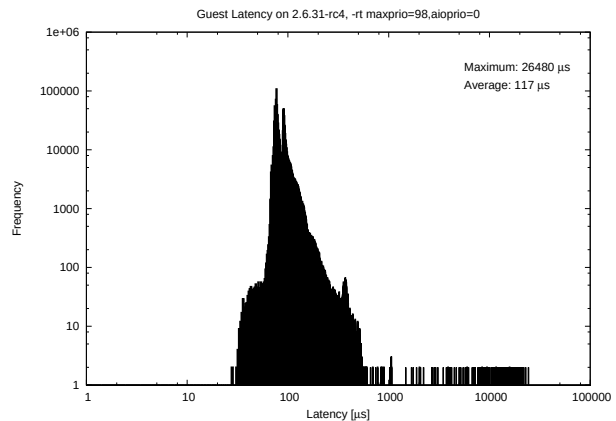
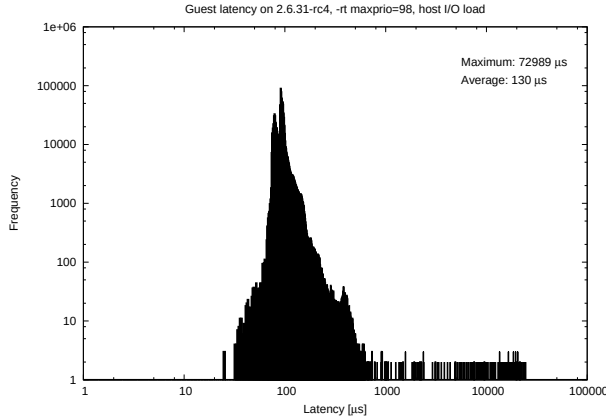


FIGURE 4: *Lowered AIO Priority*

Processing guest originated I/O load, even if handled asynchronously, apparently contributes to excessive latencies in `cyclicttest`. To confirm this assumption, we moved the `bonnie` test back on the host. This roughly resulted in the same latency distribution as in the previous experiment, see Figure 5.


 FIGURE 5: *I/O Load Moved to Host*

4 PREEMPT-RT hosting

In the next step, we switched the host to the same PREEMPT-RT kernel that was already in use for the guest system. The aim was to evaluate if the I/O-related priority inversions we found on a standard kernel are caused by QEMU/KVM itself or by the host kernel not respecting priorities properly in all cases.

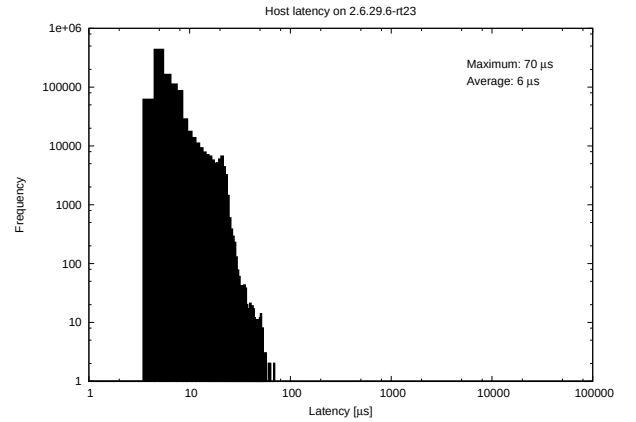
4.1 Priority Assignment

Selecting the maximum priority of QEMU under PREEMPT-RT has to be done carefully in order to avoid livelocks. A Linux guest, for example, performs a timer check during boot-up, busy-waiting for timer interrupts to arrive. On the host side, these interrupts flow from the timer IRQ handler via the hrtimer kernel thread to QEMU's I/O thread and finally to the VCPU. If the hrtimer thread is not given a higher priority than QEMU, the guest will spin endlessly, preventing any timer IRQ delivery. As our test setups focus on guest timer IRQ latency, we lifted hrtimer processing above any others interrupt threads.

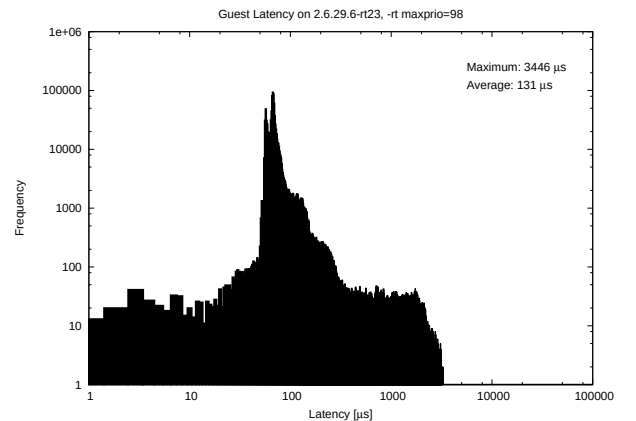
```
# chrt -p -f 99 'pgrep hrtimer/1'
```

4.2 Evaluation

Again we measured the timer latency via cyclictst on the host first, running bonnie and calibrator as load. Figure 6 shows promising numbers. They indicate that PREEMPT-RT is avoiding priority inversions because of low priority I/O load as far as this short test can reveal.


 FIGURE 6: *PREEMPT-RT Host Latency*

Running QEMU with raised priority on PREEMPT-RT results in clear worst case latency improvement over a standard PREEMPT kernel. Figure 7 depicts this measurement; it corresponds to Figure 3. On PREEMPT-RT, the latency maximum moved below 100 μs , and the average latency dropped by one third.


 FIGURE 7: *Guest Latency on PREEMPT-RT Host*

Corresponding to the test shown in Figure 4, we next dropped AIO priorities to 0. Figure 8 visualizes the best guest latency distribution we were able to achieve with our test setup. Here the latency average drops to only 75 μs , and a maximum latency of less than 500 μs was observable (again noting that the test only ran for 15 minutes).

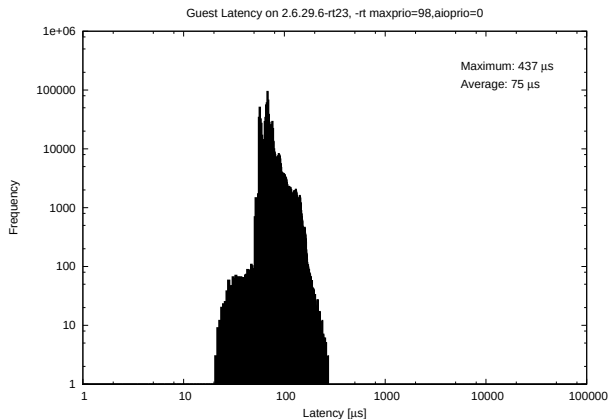


FIGURE 8: *PREEMPT-RT with Lowered AIO*

5 Paravirtualized Scheduling

To fulfill the requirements of real-time tasks running in a VM, the hypervisor may assign guaranteed resource time shares to that VM. All real-time task requirements must then be fulfillable by granting an adequate time slice in a predefined scheduling cycle. Such a static time slice allocation can be impractical in a highly reactive environment.

Alternatively, the hypervisor may prioritize the real-time VM over any other VM. This approach is sufficient as long as the real-time VM only executes tasks that have a higher importance than *any* other task in the system. Otherwise, background jobs in the prioritized VM may use up all CPU resources of the host, starving the other VMs. A straightforward approach to avoid this priority inversion is to give the hypervisor a hint about the internal states of its guests. The hypervisor can then adjust the schedule of the VCPUs accordingly.

Such an approach is already used for resolving efficiency issues related to guest CPUs waiting on contended spinlocks [13]. Consider one VCPU holding a spinlock, then being preempted by another VCPU which starts to spin on the very same lock. Without any knowledge of the guest state, the hypervisor can only keep the second VCPU running until it consumed its time slice. But given some information that the second VCPU is currently spinning on a lock, the hypervisor can instead grant the host CPU to the first VCPU that is able to make progress on its jobs. Moreover, there is a probability that this VCPU will have released the spinlock the next time it is preempted by the second VCPUs. Thus, the overall efficiency increases as less CPU time is burned unproductively.

Today this additional information about spinlock contentions can only be provided by the guest OS itself. Linux introduced paravirtual spinlock operations for this purpose. These replace the native spinlock code when Linux detects that it running over a hypervisor which supports an alternative mechanism. Currently only Xen makes use of them [14]. Upcoming Intel and AMD CPUs will include hardware-assisted detection of spinlock loops, which triggers a guest exit so that the hypervisor can schedule a different VCPU.

However, hardware assistance cannot be expected for solving the scheduling problem of prioritized guest CPUs because task scheduling decisions are done in software on commodity platforms like x86. Therefore, we need a software interface between guest and hypervisor to propagate the required information, and we have to define an execution model which both hypervisor and guest OSes can adopt.

5.1 Execution Model

The least common denominator in task scheduling today is fixed-priority scheduling. We chose POSIX [15] to define the detailed semantics of the supported scheduling policy. The following policies are so far supported by our model:

- **SCHED_OTHER:**
Default time-sharing policy of the host.
- **SCHED_FIFO:**
FIFO policy as defined by POSIX.
- **SCHED_RR:**
Round-robin policy as defined by POSIX.

The hypervisor schedules a VCPU according to the last scheduling policy and priority that have been reported by the guest. The available priority range is $[0, p_{max}]$. p_{max} is a per-VCPU priority limit that the hypervisor can be configured to. The guest-provided priority p_{guest} , confined to the range of $[0, 99]$, is mapped to the host priority p according to

$$p = \left\lfloor p_{guest} \frac{p_{max}}{99} \right\rfloor$$

The guest-provided scheduling parameters apply if no virtual interrupt is waiting to be injected or is currently processed by the guest. As soon as the hypervisor marks an interrupt for injection into a VCPU, it raises the VCPU to p_{max} . If the guest's current scheduling policy is **SCHED_OTHER**, it is changed to **SCHED_FIFO**, otherwise it is left unmodified. The boost is kept until the guest reports that it completed interrupt handling on the corresponding VCPU.

The model also supports one level of interrupt nesting to allow for accurate non-maskable interrupt (NMI) virtualization. If a virtual NMI is marked pending while interrupt boosting is already applied, the NMI boosting is added. In the following, completion for both interrupt types has to be reported by the guest CPU for deboosting it to its base policy and priority.

5.2 Interface

Our current version of a paravirtual scheduling interface requires two new hypercalls, that is paravirtual service calls from the guest to the hypervisor:

- **Set Scheduling Parameters**

This hypercall informs the hypervisor about a potentially changed scheduling policy and/or priority for the specified VCPU. The hypercall's argument list consists of the target VCPU, the new scheduling policy and the new priority value p_{guest} .

In order to support SMP guest use case where one VCPU may change the scheduling parameters of another VCPU, the hypercall interface can also be invoked on remote VCPUs. In our implementation for x86, we use the physical ID of the emulated local APICs to specify the target VCPU of this hypercall.

If the target VCPU is the same as the caller of the hypercall and the VPU currently undergoes interrupt priority boosting, the boost is cleared so that no additional *Interrupt Done* hypercall has to be issued.

- **Interrupt Done**

This hypercall has to be invoked by a VCPU that finished its interrupt handling but will not invoke *Set Scheduling Parameters* as no reschedule is required. No arguments are passed.

For the integration in QEMU/KVM, we chose KVM's own hypercall interface that is based on vmcall/vmmcall, the vendor-specific x86 instructions to call from a guest into its hypervisor. Like other KVM hypercall interfaces, this extension is advertised to the guest via a flag in KVM's CPUID leaf.

5.3 Host-Side Realization

The modifications to implement our paravirtual scheduling interface in QEMU/KVM focused on

KVM's kernel module. Here we had to provide the *Set Scheduling Parameters* hypercall interface which, after checking and mapping the passed parameters as well as potentially deboosting the VCPU, finally invokes `sched_setscheduler()` for the Linux task of the corresponding VCPU.

Standard interrupt handling was also straightforward to implement. We hooked into `kvm_vcpu_kick()` and the IOCTLs to trigger an interrupt or an NMI from user space and applied the required boosting. Furthermore, we added the *Interrupt Done* hypercall interface.

More challenging was the implementation of VCPU boosting on APIC and PIC timer events. To avoid a priority boost gap between the host timer IRQ and the guest IRQ injection, we have to start the boosting as soon as the host timer fires. KVM uses Linux hrtimers for setting up guest timer events. When these timers fire, a KVM-provided handler is invoked, looks up the target VCPU and sets a flag that pending timers have to be considered on next guest entry.¹ It is not yet clear at this point if an IRQ or NMI will finally be injected. Moreover, on non-*PREEMPT-RT* kernels, the hrtimer handler runs in interrupt context, preventing a direct invocation of `sched_setscheduler()` due to the IRQ-unsafe implementation of that service.

To work around this, we register per-CPU kernel threads, running at highest host priority, forwarding our boost requests from the timer handlers to the Linux scheduler. We additionally introduce an intermediate boosting state, "timer pending", that is kept until KVM decides if the pending timer will raise an IRQ or an NMI. This boost is not affected by any guest-originated clearing.

As we rely on the physical APIC ID for looking up the internal VCPU state during the *Set Scheduling Parameters* hypercall, our implementation is so far limited to the standard case of in-kernel guest IRQ controller emulation. All information we need is at hand here, while it would otherwise reside only in user space.

To configure p_{max} for every VCPU, the new IOCTL `KVM_SET_MAX_PRIO` is now available for KVM's VCPU file descriptors. User space is expected to call it during setup of a new virtual CPU.

The QEMU user space part only had to be extended by the `pvsched=on|off` option for the `-rt` command line switch. It controls whether paravirtual scheduling is advertised to the guest and VCPU maximum priorities are configured to the value defined by `maxprio` minus one.

¹A plain flag is enough as timers always fire on the same host CPU the associated VCPU may run on. Thus, the guest is either already preempted by the host timer and the flag will simply be evaluated on guest re-entry, or this will happen as soon as the VCPU resumes (i.e. re-enters guest context).

5.4 Guest-Side Realization

To demonstrate the guest-side implementation we chose Linux as well. Linux has the advantage of providing a well-established OS paravirtualization layer called paravirt-ops. It comes with detection and runtime deactivation services that allow to reuse a image in the absence of paravirtual scheduling support without noticeable overhead.

We first introduced a new group of paravirt-ops, `pv_sched_ops`. This group contains two callbacks that directly map on two previously defined hypervisor calls. The wrappers for these calls are;

```
void cpu_set_sched(int cpu, int policy,
                  int priority);
void cpu_interrupt_done(void);
```

`cpu_set_sched()` is called from `schedule()`, equipped with the parameters of the next task. Furthermore, we hook into `__sched_setscheduler` where we call `cpu_set_sched()` if the manipulated thread is the current one on its assigned CPU.

`cpu_interrupt_done()` is inserted into `nmi_exit()`, where it is called unconditionally, and into `irq_exit()`. In the latter case, we only invoke the hook if we left interrupt handling and no reschedule is pending that will deboost the current CPU anyway. The following code extract shows the modified function:

```
void irq_exit(void)
{
    ...
    sub_preempt_count(IRQ_EXIT_OFFSET);
    if (!in_interrupt()) {
        if (local_softirq_pending())
            invoke_softirq();
        if (!need_resched())
            cpu_interrupt_done();
    }
    ...
}
```

While the `pv_sched_ops` declaration and setup is currently only available for x86, our hooks were exclusively added to generic code and, thus, are portable to other architectures. Moreover, no adaption of this paravirtual interface was required to use it in a PREEMPT-RT guest. But we were forced to disable KVM's paravirtual MMU operations as their current implementation is incompatible with PREEMPT-RT's MMU subsystem changes.

5.5 Evaluation

Ideally, enabling paravirtual scheduling should not have any negative impact on our test scenarios. It

should not raise the worst case latency nor shift the average. Realistically, paravirtual scheduling introduces overhead, both related to leaving the guest to inform the host about priority changes as well as applying priority changes because of guest activity or interrupt injections.

To evaluate the impact, we repeated the test of running `cyclictest` with raised maximum but low AIO priority both over the standard as well as the PREEMPT-RT kernel, this time with `pvsched=on`. The results can be found in Figures 9 and 10 that correspond to Figures 4 and 8, respectively. We see a slight increase in the average latency on both host kernels, which is assumed to reflect the paravirtual scheduling overhead. Moreover, we see a higher worst case latency on the standard PREEMPT kernel. But as this increase does not repeat over PREEMPT-RT and the tests may have not revealed the actual worst case, we consider this a random effect.

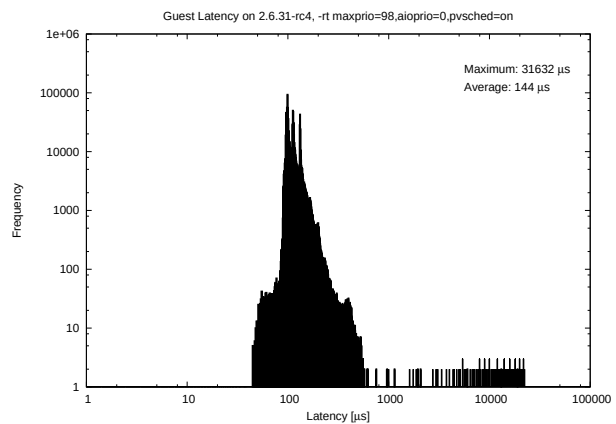


FIGURE 9: *Paravirtual Scheduling on CONFIG_PREEMPT*

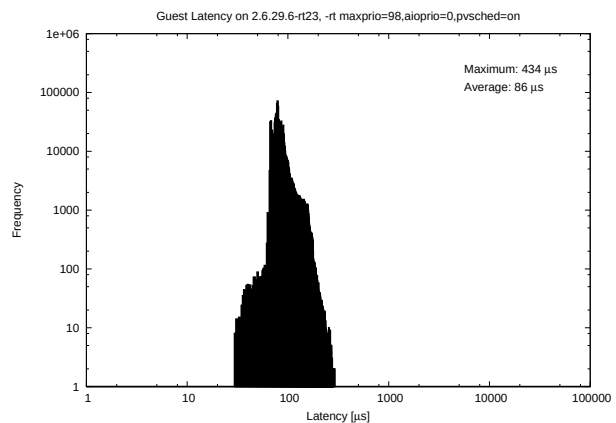


FIGURE 10: *Paravirtual Scheduling on CONFIG_PREEMPT_RT*

6 Analysis

The measurements of the real-time properties QEMU/KVM can provide for its guest have shown the following:

- With careful tuning, worst case timer-driven guest scheduling latencies below 1 ms and average latencies below 100 μ s can be achieved on a PREEMPT-RT host kernel.
- Parallel I/O activity of the guest has significant influence on its overall latency. This can be mitigated by lowering the priority of AIO completion handling.
- Our paravirtual scheduling interface can be implemented in a way that no obvious priority inversions are introduced, while only causing a slight average latency increase.

We furthermore observed the following effects during the tests:

- Using QCOW2 images (backing or temporary) imposes high latencies on the guest, irrespective of PREEMPT-RT use and careful priority settings.
- Using local SDL graphic output or applying high load on the emulated serial port also causes priority inversions for the guest.

The latter effects are apparently related to the locking scheme in QEMU user space part. As a global "I/O lock" is held while accessing potentially blocking Linux services, every concurrent access to QEMU's I/O device model, for example by a VCPU, will suffer from the same delay. These bottlenecks are expected to cause problems also on large SMP guest systems as the number of parallel I/O access naturally increases here. On the long-term, QEMU's device model will therefore have to be made thread-safe by applying a more fine-grained and scalable locking scheme.

Yet hidden behind the above effects and/or not sufficiently stressed by our tests are delays caused by significant CPU load inside QEMU's I/O layer. For example, the user space networking stack may face large packet queues or decide to fork-off helper programs. Such paths require further analysis, even when a more scalable locking scheme has already been established. They could still impact their direct caller, namely VCPUs issuing synchronous I/O operations or the main I/O thread dispatching incoming data from the host.

As the priority inversion effects in user space dominated our tests, we had no closer look at the

locking situation in KVM's kernel model. However, at the time of writing, there were ongoing activities to establish a more fine-grained locking scheme also in kernel space [16].

7 Conclusion

In this paper, we analyzed QEMU/KVM regarding its currently achievable soft real-time capabilities. We demonstrated that, when carefully tuning the setup and using PREEMPT-RT on the host, sub-millisecond scheduling latencies inside guests can be achieved.

We furthermore introduced a paravirtual scheduling interface. It overcomes the priority inversion problems on embedded systems that are related to black-box scheduling of guests. Measurements demonstrated that our first implementation does not introduce obvious priority inversions on host or guest side. It currently comes with an overhead that results in 15% to 25% higher average latencies and is expected to have a measurable impact on the overall system performance as well. This impact should be reducible by optimizing our approach, specifically by avoiding unneeded guest exits due to paravirtual scheduling updates. We expect improvements by skipping updates on context switches that do not change any parameters and by using lazy update algorithms.

Along with the publication of this paper, we will also release the source code that was developed for this research work. Our goal is to evolve the code base towards mainline acceptability.

References

- [1] *Top 500, Operating System share for 06/2009*, TOP500.Org, 2009. <http://www.top500.org/stats/list/33/os>
- [2] *Internals of the RT Patch*, Steven Rostedt, Darren Hart, Proceedings of the Linux Symposium, 2007.
- [3] *Real-Time Linux Wiki*, 2009. <http://rt.wiki.kernel.org>
- [4] *KVM: The Linux Virtual Machine Monitor*, Avi Kivity et al., Proceedings of the Linux Symposium, 2007.
- [5] *Kernel Based Virtual Machine*, 2009. <http://www.linux-kvm.org>
- [6] *IBM Systems Virtualization, Version 2 Release 1*, IBM Corporation, 2005. <http://publib.boulder.ibm.com/infocenter/eserver/v1r2/top>

- [7] *QEMU - open source processor emulator*, 2009. <http://www.qemu.org>
- [8] *qemu-kvm git repository*, 2009. <http://git.kernel.org/?p=virt/kvm/qemu-kvm.git>
- [9] *Real-Time group scheduling*, Linux kernel 2.6.31, Documentation/scheduler/sched-rt-group.txt, 2009.
- [10] *kvm kernel git repository*, 2009. <http://git.kernel.org/?p=linux/kernel/git/avi/kvm.git>
- [11] *RT test utils*, 2009. <http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git>
- [12] *The Calibrator (v0.9e)*, Stefan Manegold, 2004. <http://homepages.cwi.nl/~manegold/Calibrator>
- [13] *Preventing Guests from Spinning Around*, Thomas Friebe, Xen Summit 2008. <http://www.xen.org/files/xensummitboston08/LHP.pdf>
- [14] *[PATCH RFC 0/4] Paravirtual spinlocks*, Jeremy Fitzhardinge, 2008. <http://permalink.gmane.org/gmane.comp.emulators.xen.devel/53239>
- [15] *The Open Group Base Specifications Issue 6, IEEE Std 1003.1*, IEEE and Open Group, 2004.
- [16] *KVM Mailing List: [RFC] more fine grained locking for IRQ injection*, Gleb Natapov, 2009. <http://permalink.gmane.org/gmane.comp.emulators.kvm.devel/37114>

Requeue-PI: Making Glibc Condvars PI-Aware

Darren Hart

International Business Machines
15300 SW Koll Pkwy, DES2 4F 1, Beaverton, OR 97006, United States
dvhltc@us.ibm.com

Dinakar Guniguntala

International Business Machines
Embassy Golf Links, Koramangala Ring Road, Bangalore, KA 560071, India
dino@in.ibm.com

Abstract

Glibc 2.5 and later provide some support for priority inheritance (PI), but some gaps in the `pthread_cond*` APIs severely cripple its utility and lead to unexpected priority inversions and unpredictable thread wake-up patterns. By adding kernel support for proxy locking of rt-mutexes and requeueing of tasks to PI-futexes, glibc can provide PI support across the entire spectrum of mutex and condvar APIs. For some complex applications, completely avoiding priority inversion can be impractical if not impossible. For such applications, this provides a robust POSIX threading mechanism. Broadcast wake-ups can now be more efficient, waking threads up in priority order as the contended resource becomes available.

1 Introduction

Condition variables (condvars) provide a convenient userspace API to block a thread (or group of threads) on an event. Under the covers, condvars are implemented using futexes. A condvar uses multiple futexes, one for use as a wait-queue and one for locking its internal state. In addition, each condvar has an associated mutex specified by the waiters, which is also implemented using a futex. Threads blocking on an event call `pthread_cond_wait()`¹ with the associated mutex held (glibc will release the mutex prior to calling into the kernel), see Figure 1. When the waking thread calls `pthread_cond_broadcast()`², the caller expects the blocked threads to stop waiting on the event and begin contending for the associated mutex. The mutex is acquired before control returns to the application code (glibc performs the mutex acquisition). `pthread_cond_broadcast()` notifies all

threads of the event (while `pthread_cond_signal()` only notifies one)³ [11].

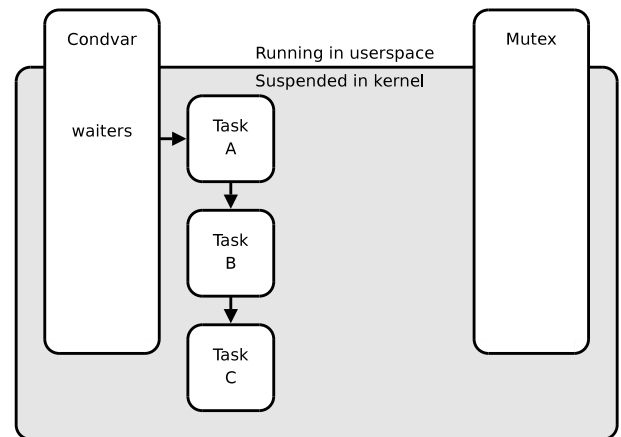


FIGURE 1: *Threads blocked on condvar after multiple `pthread_cond_wait()` calls.*

¹Throughout the paper, when `pthread_cond_wait()` is mentioned, the same applies to `pthread_cond_timedwait()` unless stated otherwise.

²Throughout the paper, when `pthread_cond_broadcast()` is mentioned, the same applies to `pthread_cond_signal()` unless stated otherwise.

³See the man pages for more gory details on the POSIX standard.

In the event of a broadcast, only one thread is woken to return to userspace, while the rest are requeued from the condvar to the mutex ⁴. The thread that is woken then acquires the mutex (in glibc) before returning to the application, see Figure 2. The unlocking of the mutex by the application then triggers the next eligible thread to be woken, and so on.

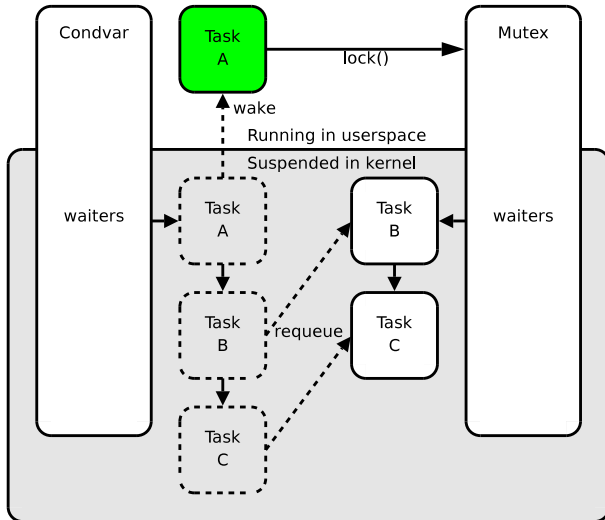


FIGURE 2: Thread state after `pthread_cond_broadcast()` (non PI case).

Futexes make use of the kernel-side `rt_mutex` for priority inheritance support. These locks have complex semantics regarding their owner and waiters which caused requeue support for PI futexes (requeue PI) to be deliberately left out in the initial implementations. Without kernel support for requeue PI, glibc could only wake all the waiters blocked on the condvar’s wait-queue and leave them to race to return to userspace and try to acquire the associated mutex, see Figure 3. The first one to return got the mutex and the rest were put back to sleep on the mutex ⁵. This effectively requeues all the threads but one, but does so in a very inefficient way. This causes additional context switches and compromises deterministic scheduling since no care is taken to ensure the highest priority waiting task acquires the mutex. This “wake all” approach creates what is commonly referred to as a “thundering herd”.

Additionally, the internal lock protecting the condvar’s state is not PI-aware. This means that even while using a PI-aware mutex, users of condvars are susceptible to unbounded priority inversion on the condvar’s internal lock. An unbounded priority inversion can occur when a high priority task is blocked on a lock held by a lower priority task that

has been preempted by a medium priority task. In this scenario, the high priority task is blocked for as long as there are runnable tasks of higher priority than the lock holder.

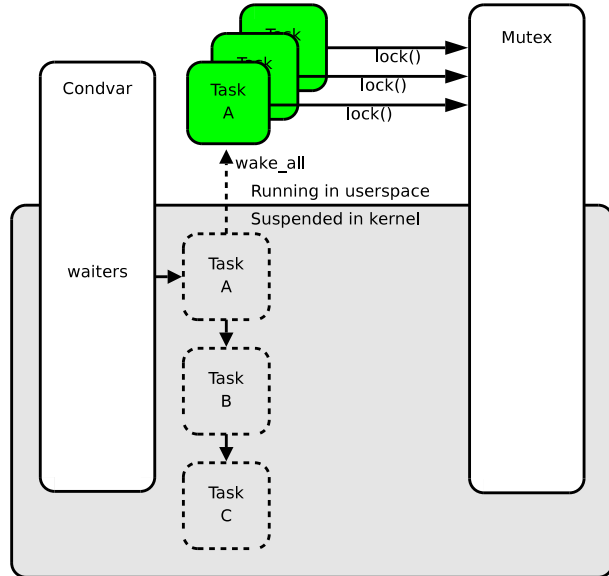


FIGURE 3: Thread state after `pthread_cond_broadcast()` (PI case without requeue PI support).

PI is often used in conjunction with other real-time features of the Linux kernel. These gaps in the PI stack hurt performance for applications using PI mutexes with condvars at best, and can lead to non-intuitive priority inversions at worst. The additional wake-ups, latency, and the unbounded priority inversion can be a show stopper for real-time applications with hard deadlines. Earlier attempts [14] to solve this problem were rejected for various reasons, including complexity and inefficiency. Nonetheless, those early attempts were valuable in guiding the development of this solution.

In the following sections, we will describe our solution to the problem, which consists of both kernel and glibc changes. Section 2 presents examples of test cases that fail or are adversely affected by this problem, as well as detailing the specific technical challenges involved in devising a solution. Section 3 provides an overview of the general Linux futex implementation as background for Section 4 which describes the Linux kernel and glibc aspects of the solution. Finally, Section 5 presents the successful results of the same test cases shown earlier.

⁴ Assuming the mutex does not have the priority inheritance attribute set.

⁵ This scenario is complicated slightly if the signaling thread holds the mutex across the call to `pthread_cond_broadcast()`.

2 Problem Definition

2.1 Failure Scenarios

The disorderly wake-up is really a functional issue, especially when viewed in the context of real-time applications. The prio-wake test case from the LTP [6] real-time test suite illustrates the problem readily. Prio-wake creates `NR_CPUS SCHED_FIFO` threads of increasing priority and puts them to sleep on a condvar. It then calls `pthread_cond_broadcast()` and logs the threads' wake-up order. If they do not wake in strict priority order the test fails, as illustrated in Listing 2, lines 12-13. The test can be run with or without the associated mutex held across the `pthread_cond_broadcast()` call. It is next to impossible for a current system to pass the test without the lock held, but even with it held, failures are very common (75% in our testing).

It is difficult to provide a non-contrived test case for unbounded priority inversion on the condvar internal lock. However, the problem has been observed on Real Time Specification for Java (RTSJ) [8] compliance test suites and other large thread-intensive applications. This problem usually manifests by the program not making progress with high priority threads blocked on a `pthread_cond_*` call.

2.2 No Ownerless RT-Mutexes

The mechanism used to implement PI within the Linux kernel is the `rt_mutex`. The `rt_mutex` maintains a list of tasks blocked on it in its `wait_list` field. Each task maintains a list of the highest priority task blocked on every `rt_mutex` it owns in the `pi_waiters` list of the `task_struct`. These lists combine to form the priority inheritance chain (PI chain) that is used to boost the priority of the `rt_mutex` owner to that of the highest priority task blocked on the `rt_mutex`. The details of the PI chain are beyond the scope of this paper, but are well documented in [2] and [15]. It is sufficient to state that to ensure the PI boosting logic succeeds whenever a new task attempts to acquire a contended lock, an `rt_mutex` must never be in a state where it has waiters and no owner. Without an owner, the PI boosting logic would not find a task to boost, and the PI mechanism would break down.

The original glibc implementation of `pthread_cond_wait()` for non-PI mutexes waits for the waiter to return from the kernel after a `pthread_cond_`

`broadcast()` before trying to acquire the associated mutex, as illustrated by lines 7-9 of Listing 1.

The problem with this implementation is the window of time between the wake-up after line 7 and the lock on line 9. Using the futex requeue mechanism, the waking thread would wake one waiter and requeue the remaining to the associated mutex. It then returns to userspace and releases the associated mutex⁶. Meanwhile, the newly woken thread attempts to acquire the mutex. Should the waking thread release the mutex before the waiter can block on it, the mutex will be left with no owner and multiple waiters, violating the usage semantics of the `rt_mutex`.

```

1 cond_wait(cond, mutex)
2 {
3     lock(cond->__data.__lock);
4     unlock(mutex);
5     do {
6         unlock(cond->__data.__lock);
7         futex(cond->__data.__futex,
8             FUTEX_WAIT);
9         lock(cond->__data.__lock);
10    } while (...)
11    unlock(cond->__data.__lock);
12    lock(mutex);
13 }
```

Listing 1: Original `pthread_cond_wait()` pseudo-code.

As mentioned previously, the original solution to this problem was to wake all the tasks blocked on the condvar, rather than requeue them as waiters, and allow them to race for acquisition of the associated mutex. Once the mutex is acquired by one of the waiters, the remaining late arrivals will then be put to sleep on the mutex. Future unlocks will work the same as any `pthread_mutex_unlock()` call would, giving the lock to the next highest priority waiter and waking it up. The initial race leads to unpredictable wake-up order and a lot of unnecessary context switches and thrashing about within the scheduler.

3 Meet the Futex

Prior to diving into the problems priority inheritance presents futexes, a basic understanding of what futexes are and how they work is needed. The best reference for the basic operations of futexes and how they can be used to create higher-level locking mechanisms is probably still Ulrich Drepper's "Futexes

⁶It is possible that the waking thread did not hold the mutex, which is a valid usage of the API, in which case it will not need to unlock it after signaling the waiters.

```

1 # ./prio-wake
2
3 -----
4 Priority Ordered Wakeup
5 -----
6 Worker Threads: 8
7 Calling pthread_cond_broadcast() with mutex: LOCKED
8
9 00000527 us: Master thread about to wake the workers
10
11 Criteria: Threads should be woken up in priority order
12 FAIL: Thread 7 woken before 8
13 Result: FAIL
14 000000: 00000102 us: RealtimeThread -8388768 pri 001 started
15 000001: 00000166 us: RealtimeThread -8389264 pri 002 started
16 000002: 00000217 us: RealtimeThread -8389760 pri 003 started
17 000003: 00000284 us: RealtimeThread -8390256 pri 004 started
18 000004: 00000332 us: RealtimeThread -8390752 pri 005 started
19 000005: 00000386 us: RealtimeThread -8391248 pri 006 started
20 000006: 00000432 us: RealtimeThread -8391744 pri 007 started
21 000007: 00000481 us: RealtimeThread -8392240 pri 008 started
22 000008: 00000691 us: RealtimeThread -8391744 pri 007 awake
23 000009: 00000753 us: RealtimeThread -8392240 pri 008 awake
24 000010: 00000813 us: RealtimeThread -8391248 pri 006 awake
25 000011: 00000855 us: RealtimeThread -8390752 pri 005 awake
26 000012: 00000885 us: RealtimeThread -8390256 pri 004 awake
27 000013: 00000929 us: RealtimeThread -8389760 pri 003 awake
28 000014: 00000953 us: RealtimeThread -8389264 pri 002 awake
29 000015: 00000968 us: RealtimeThread -8388768 pri 001 awake
    
```

Listing 2: Failing prio-wake example.

are Tricky” [12]. “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux” [13] is also a good historical reference, although fairly out of date at this point.

A futex is a fast userspace lock [3], although there is truly no such structure that can be said to encapsulate a futex. A futex is a very basic locking mechanism, and is typically used to construct higher level primitives. The most basic component of a futex is its userspace variable, a four byte unsigned integer on all platforms. The uncontended path usually involves the futex variable being marked as locked in userspace without having to call into the kernel. While the futex implementation initially made no requirements on how this value is used, the priority inheritance implementation uses this value to determine the owner of the futex and whether or not it has waiters.

From the kernel side, each futex address is converted into a `futex_key` using the physical memory address so that processes sharing a futex can use their own address space to reference the same futex. Each task blocked on a futex is represented by a `futex_q` structure, see Listing 3, which uses

the `futex_key` to add itself to a hash-table of priority lists of `futex_q` structures. As there are typically only 128 entries in the hash-table, collisions are expected; each hash-list will contain `futex_q` structures representing tasks waiting on multiple futexes. The `futex_key` is used to determine which elements are of interest to a given operation. Figure 4 illustrates this structure. A simple single-linked list is depicted for clarity, however, a `plist` is used in the actual implementation to ensure tasks are queued in fifo order by priority [15].

```

struct futex_q {
    struct plist_node list;
    struct task_struct *task;
    spinlock_t *lock_ptr;
    union futex_key key;
    struct futex_pi_state *pi_state;
    struct rt_mutex_waiter *rt_waiter;
    union futex_key *requeue_pi_key;
    u32 bitset;
};
    
```

Listing 3: struct futex_q.

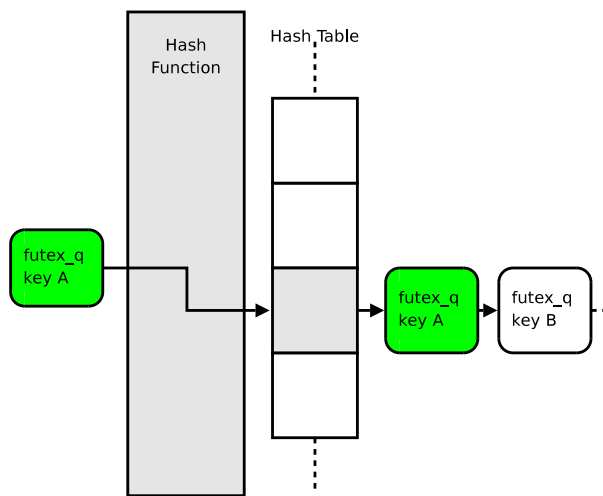


FIGURE 4: *Futex hash table queueing structure.*

Futexes are fast in the uncontended case, because the acquisition can be done from userspace using atomic instructions, such as `cmpxchg` on x86. In the contended case, the kernel is invoked to handle blocking and wake-up. There is a single system call for all futex related operations, `sys_futex`, shown in Listing 4. It accepts an operation argument, `op`, which is used to determine the appropriate kernel function.

```
long sys_futex(void *addr1, int op,
               int val1, struct timespec *timeout,
               void *addr2, int val3);
```

Listing 4: `sys_futex` definition

The supported futex operations include: `FUTEX_WAIT`, `FUTEX_WAKE`, `FUTEX_CMP_REQUEUE`, `FUTEX_LOCK_PI`, `FUTEX_UNLOCK_PI` (and now `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI`). This listing omits a couple more obscure operations that are not particularly relevant to the problem at hand.

`FUTEX_WAIT` implements a wait-queue using futexes. The calling thread is suspended until it is either notified, times out, or receives a signal. A key implementation detail worth noting is that before the thread is suspended, the kernel compares the expected value of the futex (`val1`) with what it reads from the userspace address (`addr1`). If they do not match, `-EWOULDBLOCK` is returned to userspace. This check ensures that the kernel and userspace are in agreement with the state of the futex. This ensures, for instance, that a wake-up is not issued immediately before the waiting thread is enqueued on the futex and suspended, only to sit there indefinitely having missed the wake-up.

`FUTEX_WAKE` is used to wake one or more threads blocked on a futex. As an update to the "Futexes are Tricky" article [12], futex queues are now implemented with a `plist` and waiters are woken in priority order.

`FUTEX_CMP_REQUEUE` is used to wake one or more threads waiting on `addr1` and move the rest to wait on `addr2` without waking them first. As with `FUTEX_WAIT`, the expected value of the first futex is compared prior to performing the requeue; `-EAGAIN` is returned in the event of a mismatch. This protects against another wait, wake, or requeue event completing before the hash-list locks can be acquired. Once acquired, the futex value is guaranteed not to change by convention, as userspace obviously could write to the value at any time. The obvious user of this operation is `pthread_cond_broadcast()`.

The PI futex operations diverge from the others in that they impose a policy describing how the futex value is to be used. If the lock is unowned, the futex value shall be 0. If owned, it shall be the thread id (`tid`) of the owning thread. If there are threads contending for the lock, then the `FUTEX_WAITERS` flag is set. With this policy in place, userspace can atomically acquire an unowned lock or release an uncontended lock using an atomic instruction and their own `tid`. A non-zero futex value will force waiters into the kernel to lock. The `FUTEX_WAITERS` flag forces the owner into the kernel to unlock. If the callers are forced into the kernel, they then deal directly with an underlying `rt_mutex` which implements the priority inheritance semantics. After the `rt_mutex` is acquired, the futex value is updated accordingly, before the calling thread returns to userspace.

For an exhaustive discussion of the various operations and the subtleties of the interface, see [12], [3], [1], and of course, `kernel/futex.c`.

Glibc uses futexes to implement both `pthread` condition variables and `pthread` mutexes, which are the two primitives relevant to this discussion. `Pthread` barriers are also implemented using futexes. Other locking mechanisms, such as System V semaphores [9], are unrelated to futexes.

4 The Solution

The core of the problem is ensuring that an `rt_mutex`, and by extension the PI futex, is never left in a state with waiters and no owner. This is a unique situation to `futex_requeue()` because the waiting task blocked on a non-pi futex (typically the wait-

queue of a condvar), and so did not prepare the necessary accounting prior to going to sleep. Therefore, before it can wake up, this accounting must be done on its behalf by another thread - proxy locking as I refer to it in the source. Figure 5 illustrates the desired result.

With the kernel handling the locking of the associated mutex, glibc must be updated to not try to acquire the lock after wake-up in the PI case. Since there is no POSIX interface to explicitly set the type of a pthread condvar, the condvar will need to be able to dynamically determine if it should use a PI-aware mutex for its internal lock based on the mutex the user associates with it at wait time.

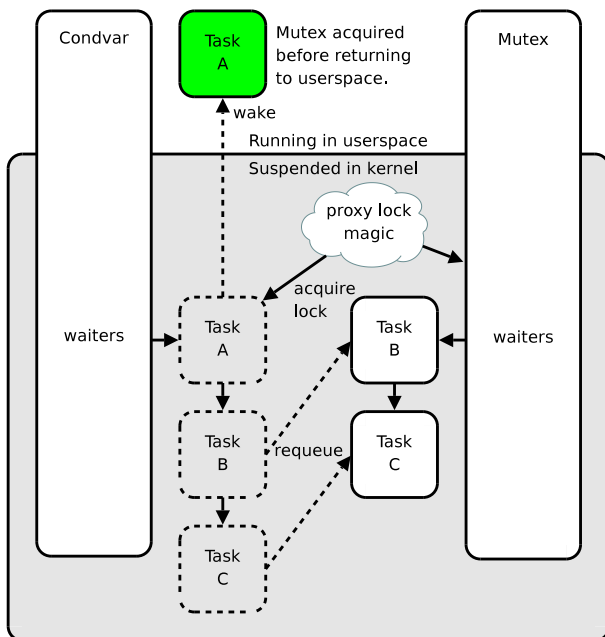


FIGURE 5: Thread state after `pthread_cond_broadcast()` (PI case with requeue PI support).

4.1 Kernel Details

4.1.1 RT-Mutex Proxy Locking

As the solution revolves around the `rt_mutex`, a basic familiarity with its usage is necessary. Acquiring an `rt_mutex` is always done in two basic steps: try to acquire it atomically, failing that, block waiting for the owner to release the lock. This is actually performed in a loop as small race windows exist between the task waking up and taking the necessary locks and actually acquiring the `rt_mutex`, during which a lock steal may occur. Before blocking, the

contending thread must set up an `rt_mutex_waiter` and add it to the `rt_mutex.wait_list`. It is from this list that the next owner is selected when the `rt_mutex` is released. Some other accounting is also performed related to maintaining the PI chain to ensure the owner is always running at the priority of the highest priority waiter. In order to implement userspace-accessible-PI-aware locking primitives, futexes are bound to an `rt_mutex` via the `futex_pi_state` structure. The details of the priority inheritance implementation and the priority boosting mechanism are beyond the scope of this paper, refer to [1] and [15] for more details.

Requeuing a futex involves two separate execution contexts: the waiter and the signaler, corresponding to `pthread_cond_wait()` and `pthread_cond_broadcast()` respectively. The waiter will suspend on a futex and the signaler will wake or requeue it to another futex. As the intended result is serialized priority ordered wake-up, to avoid the thundering herd, the requeue will wake at most one thread and requeue the rest.

If the requeue target (corresponding to the PI mutex associated with the condvar) is held for the duration of the requeue, then no wake-up will occur. However, a simple requeue is not sufficient as the requeued threads are now officially waiters for the `rt_mutex` backing the requeue target futex. The `rt_mutex_waiter` structures must be set up and the various PI chain structures must be updated so that, if necessary, the priority of the `rt_mutex` owner may be boosted to that of the highest priority requeued task (the top-waiter). When the owner releases the lock, the top-waiter will be woken and will need to complete the lock acquisition the signaler started on its behalf.

Should the requeue target be uncontended, or if a lock steal is possible⁷, the top-waiter should acquire the `rt_mutex` and return to userspace. Unfortunately, the lock acquisition cannot be left up to the waiter for reasons discussed earlier (a race back to userspace that may leave the `rt_mutex` with waiters and no owner). The signaler must acquire the `rt_mutex` on behalf of the top-waiter prior to either of them returning to userspace.

Two new `rt_mutex` routines were added to enable proxy locking: `rt_mutex_start_proxy_lock()` and `rt_mutex_finish_proxy_lock()`. First, `rt_mutex_slowlock()` was carefully refactored into its atomic and blocking sections and assumptions about the task to acquire the lock being `current` were removed. This refactoring maximizes the amount of

⁷It is also possible that the owning task died and we are using robust futex.

code reused in the new routines.

The proxy lock process begins with a call to `rt_mutex_start_proxy_lock()`, as defined in Listing 5.

```
int rt_mutex_start_proxy_lock(
    struct rt_mutex *lock,
    struct rt_mutex_waiter *waiter,
    struct task_struct *task,
    int detect_deadlock)
```

Listing 5: `rt_mutex_start_proxy_lock()` definition.

Here, `task` is the task for which `lock` is to be acquired. After taking the `lock.wait_lock`, an atomic acquisition is attempted which will succeed if the lock is uncontended (no owner and no waiters) or if a lock steal succeeds. If successful, `task` is set as the new owner, and the routine returns. Failing an atomic lock, the `waiter` structure is prepared and the PI chain is propagated via `task.blocks_on_rt_mutex()`, but the current execution context is not suspended since it is not acquiring the lock for itself. `task` is now blocked on `lock` as though it had called `rt_mutex_lock()` itself.

If the lock was not acquired atomically, the waiter will have to complete the lock acquisition when it is woken with a call to `rt_mutex_finish_proxy_lock()`, as defined in Listing 6.

```
int rt_mutex_finish_proxy_lock(
    struct rt_mutex *lock,
    struct hrtimer_sleeper *to,
    struct rt_mutex_waiter *waiter,
    int detect_deadlock)
```

Listing 6: `rt_mutex_finish_proxy_lock()` definition.

With the `waiter` already prepared and the PI chain accounting taken care of, the newly woken waiter reuses the refactored `_rt_mutex_slowlock()` to complete the lock acquisition. This will try to take the lock atomically, which will often succeed in the common case where the task is woken by the previous owner. Should the lock be stolen by a higher priority waiter, the task will suspend itself and wait for another wake-up. The loop ends when the lock is acquired or the timeout, `to`, expires.

4.1.2 New Futex Operations

The restrictions imposed by the use of the `rt_mutex` made it necessary to create two new futex operations: `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI`. The former demuxes through the system call to a new function, `futex_wait_requeue_pi()`, while the existing `futex_requeue()` was able to handle

the latter with some modifications. The new `futex_wait_requeue_pi()` function is the result of a high-speed collision between the existing `futex_wait()` and `futex_lock_pi()`, both of which were refactored slightly in order to maximize code reuse.

A task wishing to block on a futex (`futex1`) until another thread indicates it should try and acquire another futex (`futex2`) (such as `pthread_cond_wait()`) starts by calling:

```
futex_wait_requeue_pi(uaddr, fshared,
    val, timeout, bitset, clockrt, uaddr2);
```

Here, `uaddr` is the userspace address representing `futex1`, `fshared` is 0 (implying a private mapping), `val` is the expected value of `uaddr`, `timeout` is the timeout in absolute time, and `bitset` is currently unused and may be removed in a future revision. `clockrt` is 1 or 0, indicating which clock id (`CLOCK_REALTIME` or `CLOCK_MONOTONIC`) `abs_time` is represented in. Finally, `uaddr2` is the userspace address representing `futex2`.

`futex_wait_requeue_pi()` will first go through the same steps as `futex_wait()` to prepare to block on `futex1`, but prior to doing so will also prepare an `rt_mutex_waiter` and a `futex_key` (encoding `futex2`) on the stack and reference then in the `futex_q` for the requeue code to reference later. Once suspended on `futex1`, the waiter will wake due to a futex wake-up event, a signal, or timeout, in one of two states: while queued on `futex1` or after requeue to `futex2`. By way of introduction, consider the expected path: being deliberately woken due to a futex action (rather than a signal or a timeout) after being requeued to `futex2`. Once woken, the waiter must determine if the `rt_mutex` has been fully acquired on its behalf, by `futex_requeue()`, or if it must complete the lock acquisition. This is a simple matter of testing the `futex_q.rt_waiter` for NULL, as shown in the overly simplified pseudo-code in Listing 7.

```
if (!q.rt_waiter)
    return 0;
else
    rt_mutex_finish_proxy_lock(pi_mutex,
        to, &rt_waiter, 1);
```

Listing 7: Simplified pseudo-code of the lock acquisition test in `futex_wait_requeue_pi()`.

If the lock was acquired atomically by the requeue code, the `rt_waiter` in the `futex_q` will be NULL. If it is not, the waiter must now complete the lock acquisition via the proxy locking code described earlier. Either way, the waiter will return to userspace holding `futex2` and the `rt_mutex` associated with it.

Due to the nature of futexes, which necessitates access to userspace values from within the kernel, the actual implementation is much less cut and dry. Futex code is susceptible to faults, signals, timeouts, and must protect the kernel from any sort of evil userspace can conceive to throw through the system call. These routines were carefully constructed to group code that may fault together and place it early in the routine to minimize cleanup should a fault occur.

The new `futex_wait_requeue_pi()` function presents a particular challenge due to its numerous potential states. Figure 6 illustrates a slightly simplified state transition diagram for the life-cycle of the function. In the PRE WAIT state, the function checks its arguments and read-write permission on the futex address (where it may fault), prepares the `futex_q`, sets up the timeout (if there is one), and acquires the hash-list lock. If the bitset is empty or if an unrecoverable fault occurs, the routine returns an appropriate error code. Once the hash-list lock is acquired, the routine enters the QUEUEING state where it will compare the expected futex value with the actual value (where it may have to drop the hash-list lock and return to PRE WAIT to handle a fault, or return `-EFAULT` in the event of an unrecoverable fault), if the values differ, it returns `-EWOULDBLOCK`. If they match, the task is suspended on the first futex and enters the QUEUED state. This is the first “sleep state”, indicated in gray.

From any sleep state, a futex routine will resume execution for one of three reasons: a signal was delivered, a timeout occurred, or a futex wake event occurred. Each of these must be processed on exit from the QUEUED state. Before exiting with `-ETIMEDOUT` or `-ERESTARTNOINTR` (due to a signal), the QUEUED state must first determine which futex the task was woken on. By comparing the `futex_q.key` with that of the second futex, it determines if the task was requeued by `futex_requeue()` already. If the task was woken prior to requeue, then the cause of the wakeup is either a timeout or a signal. The appropriate error code is returned. If the task was requeued prior to wakeup, then those are still valid causes, but `futex_requeue()` may have also acquired the lock on its behalf, in which case the signal and timeout are ignored, and the routine returns 0, indicating a successful wakeup and lock acquisition. This can be determined by testing the `futex_q.rt_waiter` which `futex_requeue()` will set to NULL in the event of an atomic lock acquisition. If the `rt_waiter` is non-null, the REQUEUED state is entered, meaning the task has been requeued, but the lock has not yet been acquired.

At this point, before checking for a timeout or signal, the routine attempts to atomically acquire the lock in case it became available. Failing that, it will check for timeout and signal. In the event of a timeout, `-ETIMEDOUT` is returned. If a signal is pending, the system call could be restarted, but since the requeue has already occurred the futex value will have changed, causing the restarted syscall [10] to immediately return `-EWOULDBLOCK`. As such, in the event of a signal `-EWOULDBLOCK` is returned as a slight optimization. Finally, if no timeout or signal was received, the task will continue to finalize the `rt_mutex` acquisition. If contended, or in the event of a lock steal, the task is suspended. It will wake due to a signal, timeout, or `rt_mutex` unlock event, at which point the test for lock acquisition, signal, and timeout continues until one of the three is true.

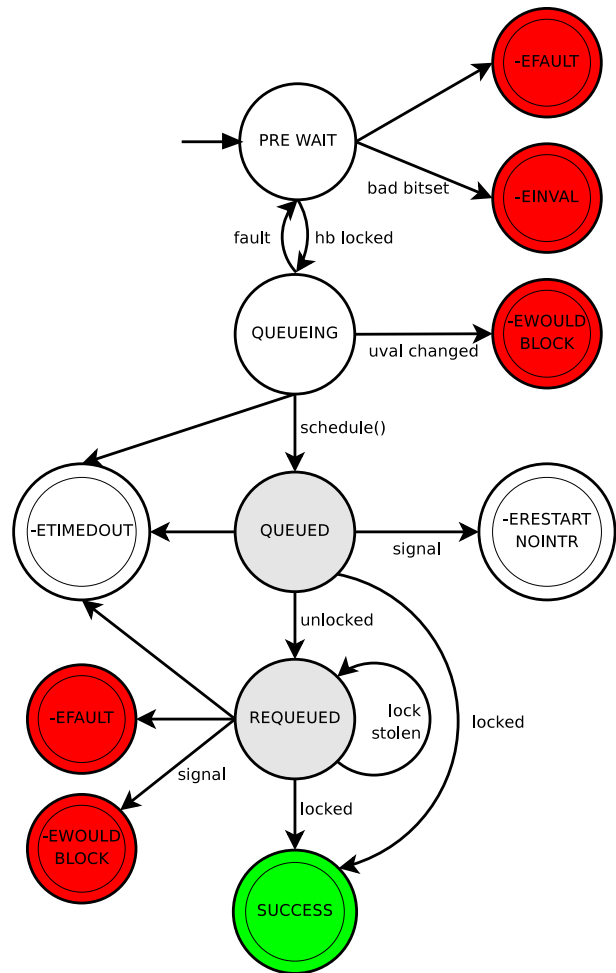


FIGURE 6: State transition diagram for `futex_wait_requeue_pi()`.

Fortunately, `futex_requeue()` was able to be modified to support requeue PI and a new function was not needed. A task wishing to wake the

tasks blocked on `futex1` and have them wake and acquire `futex2` in priority order (such as `pthread_cond_broadcast()`) begins by calling:

```
int futex_requeue(uaddr1, fshared,
                 uaddr2, nr_wake, nr_requeue, cmpval,
                 requeue_pi)
```

Here, `uaddr1` is the user address representing `futex1`, `fshared` is 0 (indicating a private mapping), `uaddr2` is the user address representing `futex2`, `nr_wake` is 1 (any other value is invalid as only one task can own the `rt_mutex` at any given time), `nr_requeue` indicates the number of tasks to requeue (usually 0 or `INT_MAX` for all), `cmpval` is the expected value of `futex1`, and `requeue_pi` is 1 (indicating `futex2` is PI futex).

Figure 7 shows the state diagram for `futex_requeue()` with the `requeue_pi` variable set to 1. In the `SETUP` state, the function checks for valid arguments, ensures a `pi_state` has been allocated for use, and ensures appropriate read-write permissions on the two futexes. If memory cannot be allocated for the `pi_state`, `-ENOMEM` is returned. Once complete, both hash-list locks are acquired, and the function enters the `ATOMIC LOCK` state.

Here, the function compares the expected value with the actual value of `futex1` (which may fault and force an unlock of the hash-list locks and a return to the `SETUP` state). If the values do not match, `-EAGAIN` is returned. Next, the routine attempts to acquire the lock on behalf of the top-waiter. If the lock cannot be acquired, or if `nr_requeue` is non-zero, the `FUTEX_WAITERS` bit is set for `futex2`. This may result in a fault, which is easier to deal with before entering the `REQUEUE LOOP` state. If the lock succeeds, as it will if the lock is uncontended or if a lock steal occurs, the top-waiter's `futex_q` is removed from its hash-list, its `futex_q.key` is updated to that of `futex2` (to indicate a requeue), its `futex_q.lock_ptr` is updated to that of the hash-list for `futex2`, and its `futex_q.rt_waiter` is set to `NULL` to indicate the `rt_mutex` has been acquired on its behalf. The top-waiter is then woken, at which point it will resume from the `QUEUED` state in `futex_wait_requeue_pi()` and promptly return to userspace with the lock held. If this atomic lock acquisition fails, the top-waiter will simply be requeued with the remaining `nr_requeue` waiters in the next state. For the first waiter on the `rt_mutex`, the previously allocated `pi_state` is setup with the current owner of `futex2` as the new `rt_mutex` owner (this may occur here or in the `REQUEUE LOOP` state). If, however, the `futex_q.requeue_pi_key` is found not to match `key2`, then `-EINVAL` will be returned to userspace, indicating that userspace tried to requeue

to a target futex other than that which the waiter was expecting.

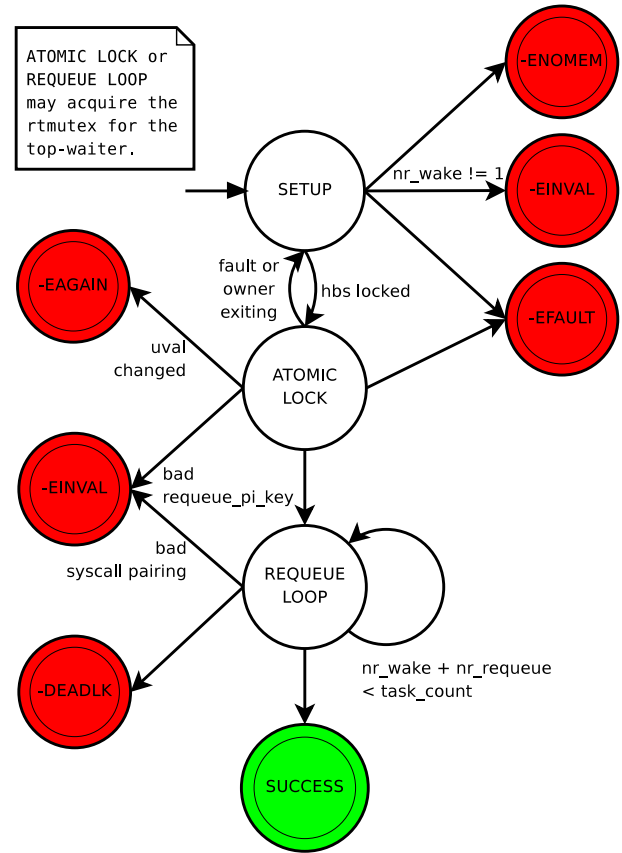


FIGURE 7: State transition diagram for `futex_requeue()`.

The `REQUEUE LOOP` state follows and is the core of the `futex_requeue()` routine. For each matching `futex_q` on the first hash-list, the loop will assign the `futex_q.pi_state` to the that representing the `rt_mutex` backing `futex2` (described above) and call `rt_mutex_start_proxy_lock()`. This call will return 1 if the lock was acquired atomically, 0 on a successful enqueueing on the `rt_mutex`, and `-EDEADLK` for various deadlock scenarios. On an atomic lock acquisition, the task is woken in same manner as in `ATOMIC LOCK`. On a successful enqueue of the task on the `rt_mutex`, the `futex_q` is removed from the first hash-list and placed on the second while its `lock_ptr` is updated accordingly. Once the loop has completed, the total number of tasks woken or requeued is returned to indicate success. The requeue loop may be interrupted in a few special cases. If `requeue_pi` is 1 and the current `futex_q` (this) has a `NULL` `rt_waiter`, `-EINVAL` is returned, indicating that the user mismatched a requeue-PI call with a non-requeue-PI call. The two

new futex operations, `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_CMP_REQUEUE_PI` must be paired only with each other. As with the earlier atomic lock attempt, if the `futex_q.requeue_pi_key` does not match `key2`, `-EINVAL` is returned. Finally, in the event of a deadlock, `-EDEADLK` is propagated to userspace.

4.2 Glibc Details

4.2.1 Making Use of Requeue PI

The changes needed to the glibc implementation of the `pthread_cond*` functions to make use of requeue PI are straight forward. Prior to issuing the system call, they now check the mutex associated with the condvar to see if the PI attribute has been set (`PTHREAD_PRIO_INHERIT`). If so, `pthread_cond_wait()` will use `FUTEX_WAIT_REQUEUE_PI`, rather than `FUTEX_WAKE` for all the waiters as was done previously, as shown in line 9 of Listing 8. The reader will recall that the mutex has been acquired in the kernel upon a successful return from `FUTEX_WAIT_REQUEUE_PI`. Therefore, the updated implementation skips the call to lock the mutex in the PI case, as shown in lines 16-17 of Listing 8.

```

1 cond_wait(cond, mutex)
2 {
3     lock(cond->__data.__lock);
4     unlock(mutex);
5     do {
6         unlock(cond->__data.__lock);
7         if (mutex->kind == PI)
8             futex(cond->__data.__futex,
9                 FUTEX_WAIT_REQUEUE_PI);
10        else
11            futex(cond->__data.__futex,
12                FUTEX_WAIT);
13        lock(cond->__data.__lock);
14    } while (...)
15    unlock(cond->__data.__lock);
16    if (mutex->kind != PI)
17        lock(mutex);
18 }
```

Listing 8: PI-aware `pthread_cond_wait` pseudo-code.

Similarly, `pthread_cond_broadcast()` will use `FUTEX_CMP_REQUEUE_PI`, as opposed to `FUTEX_CMP_REQUEUE`, as shown in lines 5-10 of Listing 9.

With these changes in place, user applications using PI mutexes with condvars can now benefit from the kernel implementation of requeue PI. The mutex will be acquired in priority order upon signaling the waiters of a condvar and scheduling overhead will be reduced.

```

1 cond_broadcast(cond, mutex)
2 {
3     lock(cond->__data.__lock);
4     unlock(cond->__data.__lock);
5     if (mutex->kind == PI)
6         futex(cond->__data.__futex,
7             FUTEX_CMP_REQUEUE_PI);
8     else
9         futex(cond->__data.__futex,
10             FUTEX_CMP_REQUEUE);
11 }
```

Listing 9: PI-aware `pthread_cond_broadcast` pseudo-code.

4.2.2 PI-Aware Condvars

Making use of the new futex operations is just the first part of making condition variables fully PI-aware. The internal lock protecting the condvar's state is a simple futex, and therefore not PI-aware. As stated in Section 1, this can lead to an unbounded priority inversion. Consider a scenario where a low priority task acquires the internal condvar lock as part of a `pthread_cond*` operation but is preempted before it can release the lock. Any high priority task that attempts an operation on the same condvar will now be subjected to an unbounded priority inversion [5]. The fix is to use the PI futex APIs for handling the internal lock if the associated mutex has the PI attribute set, as shown in Listings 10 and 11.

```

1 lock(cond)
2 {
3     mutex = cond->mutex;
4     if (mutex->kind == PI)
5         futex(cond->__data.__lock,
6             FUTEX_LOCK_PI);
7     else
8         futex(cond->__data.__lock,
9             FUTEX_WAIT);
10 }
```

Listing 10: PI-aware condvar `lock()` pseudo-code

```

1 unlock(cond)
2 {
3     mutex = cond->mutex;
4     if (mutex->kind == PI)
5         futex(cond->__data.__lock,
6             FUTEX_UNLOCK_PI);
7     else
8         futex(cond->__data.__lock,
9             FUTEX_WAKE);
10 }
```

Listing 11: PI-aware condvar `unlock()` pseudo-code

Conceptually, these changes are relatively trivial. Indeed, their implementation in the C language

versions of the respective files was trivial. However, C libraries being what they are, that was only part of the solution. Glibc maintains optimized hand-written assembly versions of all of these functions. These implementations were also updated accordingly. The interested reader is referred to the glibc libc-alpha mailing list and source repositories for details [4].

5 Results

With the solution in place, the prio-wake test passes consistently, with or without the associated mutex held across the `pthread_cond_broadcast()` call, as seen in line 12 of Listing 12. This ensures deterministic priority ordered wake-up and acquisition of the associated mutex by the waiting threads.

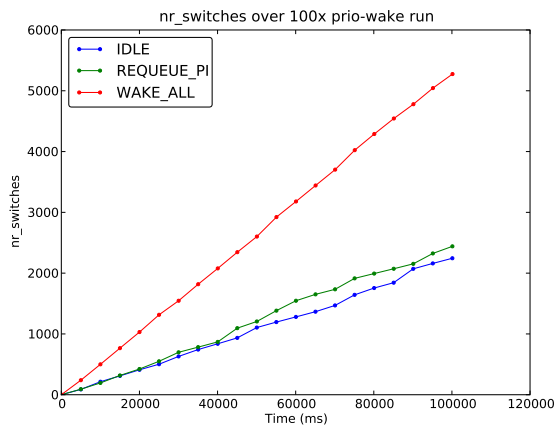


FIGURE 8: *Number of scheduler switches over time during a consecutive run of 100 prio-wake tests in which the associated mutex is not held over the broadcast. The IDLE dataset is a sampling of the system while idle for comparison.*

In addition to functional correctness, the ordered wake-up and lock acquisition also have performance benefits by eliminating the scheduling overhead the thundering herd imposes in the “wake all” approach. Figure 8 plots the number of scheduler switches⁸ that occur over the course of a run of 100 consecutive prio-wake tests, with the associated mutex unlocked across the call to `pthread_cond_broadcast()`. The

WAKE_ALL run experiences many more switches than the REQUEUE_PI run, which sees only a few more than the IDLE run. Over the course of the 102 second test, the WAKE_ALL run sees approximately 3,000 more switches than the REQUEUE_PI run, about 30 switches per second. As each iteration takes approximately one second to complete, it follows that in order to signal eight threads (one per CPU) of an event and have them acquire a shared mutex, the “wake all” approach requires 30 additional switches. The unlocked case is the more difficult one to pass using the “wake all” method as woken threads will not be hindered by the signaling thread holding the mutex, and will race only amongst themselves to return to userspace and acquire the lock, as depicted in Figure 3. Over the 100 runs, the WAKE_ALL run experienced 100% failure as compared to 0% for the REQUEUE_PI run⁹.

The kernel patches have been available in the PREEMPT_RT [7] tree since as early as 2.6.29.1-rt5 and is currently destined for the upcoming mainline 2.6.31 release. At the time of this writing, the x86_64 version of the new futex operations patch is present in glibc CVS. The remaining glibc patches are still under development, and should be available in the near future.

The system used for all test results reported here was an 8-way x86_64 system. The results were collected on a Linux 2.6.31-rc6-rt4 kernel.

6 Acknowledgements

We would like to extend our thanks to Thomas Gleixner for his initial conceptual designs and facilitating the inclusion of the the kernel and glibc patches into their respective upstream repositories. Thanks to Peter Zijlstra for his efforts to reduce overhead related to accessing user pages in the futex code as well as his sound advice at various points throughout the development process. Will Schmidt, Paul McKenney and Venkateswararao Jujjuri (JV) provided early reviews which were critical to making this paper more legible than it would have been otherwise. Thanks to IBM for the resources, both hardware and person-hours, required to complete this project. Finally, we are indebted to our families for their sacrifices to enable us to pursue this effort.

⁸As reported by the `/proc/sched_debug nr_switches` field.

⁹At the time of this writing, the performance benefits are not observed with the associated mutex held over the `pthread_cond_broadcast()` call. This is considered to be a flaw and is expected to be resolved as soon as possible. The functional results remain intact however.

```
1 # LD_LIBRARY_PATH=/test/dvhart/lib/x86_64 ./prio-wake
2
3 -----
4 Priority Ordered Wakeup
5 -----
6 Worker Threads: 8
7 Calling pthread_cond_broadcast() with mutex: LOCKED
8
9 00000620 us: Master thread about to wake the workers
10
11 Criteria: Threads should be woken up in priority order
12 Result: PASS
13 000000: 00000193 us: RealtimeThread -17617056 pri 001 started
14 000001: 00000275 us: RealtimeThread -17617552 pri 002 started
15 000002: 00000323 us: RealtimeThread -17618048 pri 003 started
16 000003: 00000375 us: RealtimeThread -17618544 pri 004 started
17 000004: 00000427 us: RealtimeThread -17619040 pri 005 started
18 000005: 00000479 us: RealtimeThread -17619536 pri 006 started
19 000006: 00000526 us: RealtimeThread -17620032 pri 007 started
20 000007: 00000575 us: RealtimeThread -17620528 pri 008 started
21 000008: 00000666 us: RealtimeThread -17620528 pri 008 awake
22 000009: 00000679 us: RealtimeThread -17620032 pri 007 awake
23 000010: 00000689 us: RealtimeThread -17619536 pri 006 awake
24 000011: 00000706 us: RealtimeThread -17619040 pri 005 awake
25 000012: 00000716 us: RealtimeThread -17618544 pri 004 awake
26 000013: 00000725 us: RealtimeThread -17618048 pri 003 awake
27 000014: 00000735 us: RealtimeThread -17617552 pri 002 awake
28 000015: 00000745 us: RealtimeThread -17617056 pri 001 awake
```

Listing 12: Passing prio-wake example.

References

- [1] *Documentation/pi-futex.txt*. Linux kernel documentation, 2.6.31-rc6-rt4.
- [2] *Documentation/rt-mutex.txt*. Linux kernel documentation, 2.6.31-rc6-rt4.
- [3] *futex - Fast Userspace Locking system call*. Linux futex(2) man page, Ubuntu 9.04.
- [4] Glibc resources. <http://www.gnu.org/software/libc/resources.html>.
- [5] Internal lock in pthread struct is vulnerable to priority inversion. http://sourceware.org/bugzilla/show_bug.cgi?id=5192.
- [6] Linux test project. <http://ltp.sourceforge.net/>.
- [7] Real-time linux wiki. <http://rt.wiki.kernel.org>.
- [8] Real time specification for java. <http://www.rtsj.org>.
- [9] *semop, semtimedop - semaphore operations*. Linux semop(2) man page, Ubuntu 9.04.
- [10] BOVET, D., AND CESATI, M. *Understanding the Linux Kernel 3rd Edition*, 3 ed. O'Reilly Media, Inc., November 2005.
- [11] BUTENHOF, D. *Programming with POSIX Threads*. Addison-Wesley, Boston, MA, USA, 1997.
- [12] DREPPER, U. Futexes are tricky. <http://people.redhat.com/drepper/futex.pdf>, August 2009.
- [13] FRANKE, H., RUSSELL, R., AND KIRKWOOD, M. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux symposium* (2002), pp. 479–789.
- [14] PEIFFER, P. [patch 2.6.19.1-rt15][rfc] - futex_requeue_pi implementation (requeue from futex1 to pi-futex2). <http://lkml.org/lkml/2007/1/3/62>, 2007.
- [15] ROSTEDT, S. *Documentation/rt-mutex-design.txt*, 2006. Linux kernel documentation, 2.6.31-rc6-rt4.

Using the Realtime Preemption Patch on ARM CPUs

Jan Altenberg

Linutronix GmbH

Auf dem Berg 3, 88690 Uhldingen-Muehlhofen, Germany

jan.altenberg@linutronix.de

Abstract

During the last few years, Linux has established itself as the fastest growing platform in the embedded sector. This arose from the fact that there is no other Operating System which supports such a big variety of different hardware. Also the ongoing rapid development of the Realtime Preemption Patch made Linux a perfect choice for automation and control applications which require hard-realtime capabilities.

ARM CPUs are widely used in embedded designs. There is a huge variety of cheap ARM9 and powerful ARM11 and Cortex CPUs which are predestined for use in embedded realtime controllers (because of the low power consumption and the growing number of features).

This paper demonstrates the behavior of the Realtime Preemption Patch on different ARM based embedded designs. The specific measurement environments will be explained in detail and the basics on how to get the Realtime Preemption Patch running on a specific ARM platform (and the major pitfalls) will be shown. Also the differences among ARM9 / ARM11 and Cortex CPU designs and how those differences influence the realtime behavior will be examined. As a result, the paper provides a decision guide for choosing the best fitting ARM design for a Realtime Linux based embedded platform.

1 What is Realtime?

1.1 Terms and definitions

Before looking into different ARM CPU designs and their behaviour in a Realtime Linux environment, I'd like to introduce some terms and definitions. First of all we should clarify the meaning of the word "Realtime". Misleadingly realtime systems are often considered to be as fast as possible. But realtime systems should be "as fast as specified rather than as fast as possible" (Doug Niehaus). Realtime systems have to meet specific timing constraints, which means they have to behave in a completely deterministic way. Calculations have to be done at a specific point in time and the system has to respond to external events in a given timeframe. The time duration between the occurrence of an external event and the correct reaction to that event is termed as latency. The variation of the latency is called jitter. Realtime systems are evaluated on the basis of their latency and their jitter (are those values deterministic and do they meet the needed timing constraints?).

2 ARM designs

2.1 ARM9

ARM9 is a 32 bit RISC CPU design. With this generation of CPUs ARM has moved from a "von Neumann" architecture to a Harvard architecture (which means the CPU can read an instruction and perform a data memory access at the same time). There are two ARM9 subfamilies:

- the ARM9T family (based on the ARMv4T architecture)
- the ARM9E family (based on the ARMv5TE architecture, which implements longer pipelines. It also has a slightly enhanced instruction set)

The major disadvantage of the ARMv4 / ARMv5 designs is their cache implementation. The cache is organized as virtual indexed and virtual tagged, which means that both the index and the tag are

based on virtual addresses. On Linux each process has its own virtual address space. This implies that a physical address might be mapped to different virtual addresses at a time. So this cache implementation will lead to a more frequent cache flushing. The cache has to be invalidated after each context switch. The cost of the cache flushing is about 1k - 18k CPU cycles. The "indirect cost" (influenced by the side operations for filling up the cache lines and the TLB) can take up to 50k CPU cycles. This means 200us for a 250MHz CPU! There are two different approaches for addressing this problem:

- Using a flat address space (uClinux)
- Using the Fast Context Switch Extension (FCSE)

On ARM9 FCSE offers a 7 bit PID register, which can be combined with 25 bit virtual addresses. This will lead to 128 independent address spaces. Richard Cochran recently posted a patch on the Xenomai-Devel list, which implements FCSE for the Intel IXP425 architecture. Gilles Chantepredrix posted an enhanced version on the ARM kernel list, which has been tested on a AT91RM9200 CPU. The advantage of this approach is the combination of memory protection and a good cache performance. The disadvantage is the limitation of 128 processes and 32MB address space (although this might be sufficient for most of the embedded systems).

2.2 Xscale

Xscale is Marvell's implementation of the ARMv5 architecture. The Xscale family consists of five different families:

- IXP (network processors)
- IXC (control plane processors)
- IOP (i/o processors)
- PXA (application processors)
- CE (consumer electronics processors)

There are several powerful Xscale CPUs, which are designed to run at high CPU frequencies. But since the design is also based on the ARMv5 architecture, Xscale also suffers the problem with virtually tagged caches.

2.3 ARM11

ARM11 is a 32 bit RISC design which implements the ARMv6 instruction set. There are 4 subfamilies of the ARM11 architecture:

- ARM1176 (targeted for consumer electronics)
- ARM1156 (targeted for automotive, embedded control, imaging, ...)
- ARM1136 (targeted for network infrastructure, consumer electronics and infotainment)
- ARM11 MPCore, which is a multiprocessor design

The complete family implements the Jazelle technology for efficient Java execution (on ARM9 this is only available for the ARM926-EJS cores). ARM1136JF-S, ARM1176JZF-S and ARM1156T2F-S cores also have a powerful Vector Floating Point Unit. The main difference to the ARM9 family is the cache design. Unlike the ARM9 implementation, the ARM11 cache implementation is virtually indexed and physically tagged. In combination with its high performance memory system this speeds up the context switching time by several orders of magnitude (in comparison to the ARM9 family). Furthermore, the ARM11 family is designed to run at higher CPU frequencies.

2.4 Cortex

Cortex constitutes the new generation of ARM CPUs. The Cortex family is splitted into three families for different applications:

- Cortex A: Application (variable caches, MMU / MPU)
- Cortex M: Microcontroller (no cache, MPU optional)
- Cortex R: Realtime (variable cache, MPU optional)

Those families are evolved from the applications which are addressed by the most popular ARM designs on the market: ARM7, ARM9 and ARM11. This doesn't mean that Cortex implements those designs! The Cortex families are just designed to meet the requirements of the target applications. The Cortex M series addresses the Microcontroller market, so it can be considered as a modern successor of the ARM7 design. The M series combines a simple

programming concept with the the performance of modern CPUs. The Cortex A series are designed for complex applications and for modern operating systems. The Cortex R series are designed for Realtime applications (Low Level RTOSses, ...). In view of Realtime Linux the Cortex A series is quite interesting, because it can be considered as the successor of the ARM11 design. The Cortex A series also offers a multicore design (Cortex A9 MPcore).

ARM platform.

Feature	Kernel
Deterministic scheduler	vanilla
Preemption Support	vanilla
PI Mutexes	vanilla
High-Resolution Timer	vanilla (since 2.6.24)
Preemptive Read-Copy Update	vanilla (2.6.25)
IRQ Threads	vanilla (since 2.6.30)
Full Realtime Preemption Support	Preempt RT

TABLE 1: *Preempt RT features on ARM*

3 Realtime Preemption Patch

The Realtime Preemption Patch was started by Ingo Molnar and Thomas Gleixner. Unlike other approaches Preempt RT doesn't introduce a Microkernel. Preempt RT brings hard realtime capabilities directly into the Linux kernel. One of the basic features which have been introduced by the Realtime Preemption Patch, are Threaded Interrupt Handlers. IRQ Handlers are moved into their own kernel thread, which means they can be scheduled like any other process in the system. The basic feature of Preempt RT is the replacement of the locking primitives. Spinlocks are relaxed by sleeping mutexes (that's why Preempt RT is sometimes called the "sleeping spinlocks patch"). Lots of the Preempt RT features already made it into Mainline: PI Mutexes, High Resolution Timer, Preemptive RCU (Read Copy Update), IRQ Threads.

3.1 The Realtime Preemption Patch on ARM

The great thing about the Realtime Preemption Patch is, that everything is implemented in a portable and generic way. Which means that if a platform is supported there's nothing special to care about. So the usage on an ARM CPU isn't that different to the usage on a generic x86 based system. Recent versions of Preempt RT have full ARM support. Table 2 shows the available features and the kernel version since when they are available for the

In addition to the kernel features, there are also some prerequisites for the userspace environment. The C Library needs a proper threading implementation, PI futex support and clock handling. For GLIBC at least version 2.5 is needed. Be careful: Currently uClibc can't be used for Realtime applications. The most recent release has still no support for priority inheritance pthread mutexes.

4 Measurements

4.1 Suitable test scenarios

4.1.1 Load scenarios

As already mentioned in 1.1 the behaviour of a Realtime system must be deterministic. A Realtime system is characterized by its latencies and by its jitter. When choosing a Realtime platform, one has to prove, that this platform meets the timing requirements of the application under all circumstances. Benchmarking a Realtime system means to examine the latencies and the jitter in a worst-case scenario. An Operating System has to manage the system resources (CPU, memory, ...) and to coordinate the activities like external events and user applications. So, a worst-case scenario for an OS means a high utilization of the system resources and a large number of applications and external events (like interrupts). For a Linux system such a scenario could be easily achieved with:

- **hackbench:** Hackbench has been designed as a scheduler benchmark. It creates n groups of 20 servers and 20 clients. The servers and the clients talk to each other via sockets. The number of groups is given by a commandline

argument (Example: hackbench 10 would create 400 processes: $10 * 20 * 20$). This scenario generates a high CPU load and high memory utilization.

- Flooding from another system: Adding the -f options to the ping command causes ping not to wait after sending an ICMP message. It'll just send the packets as fast as possible. "Ping flooding" the target system from a different machine will cause a huge number of network interrupts. Be careful: The Flooding will cause the IRQ Handler of the network interface to run very often. It'll also increase the runtime of the Soft IRQ Handlers for the RX and TX handling. All of them are Kernel Threads with Realtime priority. So, on PREEMPT_RT please check /proc/sys/kernel/sched_rt_runtime_ns. This defines a threshold for the runtime of Realtime Tasks (to prevent a starvation of the low priority tasks). The default value is 950ms, which means if the rt runtime exceeds 950ms, the rt tasks won't be scheduled up to the full second. You can disable this behaviour by writing -1 to sched_rt_runtime_ns.

4.1.2 Test tools

In addition to an appropriate load scenario, suitable test applications are needed. Those test applications should prove the determinism of our Realtime System and they should give us an idea about the worst-case latencies and the jitter. Therefore the following parameters are of interest:

- The responsiveness to external events
- The accuracy of tasks which are scheduled for a specific time

The response times for external events can easily be determined by a simple UIO driver. The UIO driver just implements an interrupt handler, which:

- responds to an interrupt, which might be generated by a GPIO input and
- toggles a pin

The response time can be determined with a scope.

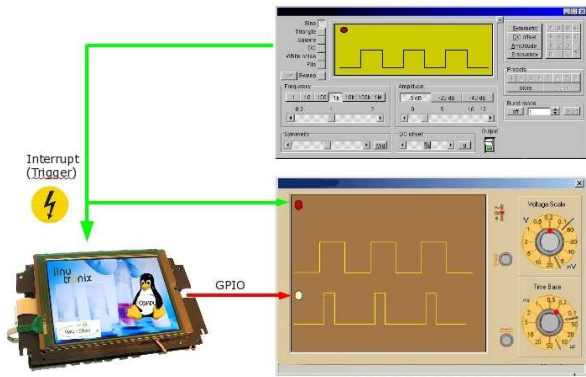


FIGURE 1: *Measuring the IRQ response time*

Figure 1 shows this test scenario: A function generator is used as an interrupt source, which is connected to a general purpose pin of the target. The UIO driver responds to this interrupt and toggles a pin. Both, the interrupt source and the "response pin" are connected to different channels of a scope. So, the time difference between the slopes on both channels shows the response time of the interrupt. Since we implement the interrupt handling with an UIO driver, we can wait for the interrupt accuracy with a userspace task. The userspace task can toggle the "response pin" one more time. So, the time difference between two peaks of the "response pin" can give us an idea about the context switching time.

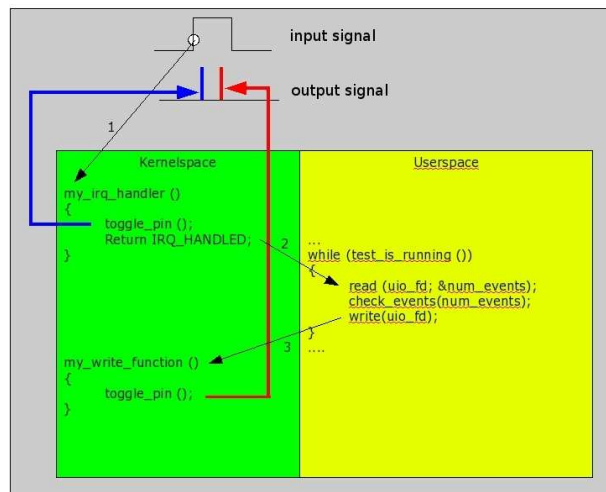


FIGURE 2: *Measuring the context switching time*

As shown in figure 2 the userspace task does a blocking read on the UIO driver. After an interrupt event has occurred, the IRQ handler will be executed and the UIO core will wake up all waiters on the UIO

device. The userspace task itself "abuses" the UIO write() function to toggle the response pin. This will cause two context switches between two peaks of the "response pin": Interrupt occurs - Userspace task will be woken up - Userspace Task calls back into kernel to toggle the pin. Be careful: We can't force the UIO interrupt handler to run in hard interrupt context. The UIO core uses wakeup_interruptible (which is not allowed in hard interrupt context). Since the read() call to the UIO driver will return the number of interrupt events, we use this to check, if we have missed an interrupt event. The UIO driver can be implemented as a UIO platform driver (uio_pdrv), which can be registered in the board support. The following listings show, how this mechanism can be implemented:

```
static irqreturn_t uio_latency_handler
    (int irq,
     struct uio_info *dev_info)
{
    at91_set_gpio_value(AT91_PIN_PB21, 1);
    udelay(2);
    at91_set_gpio_value(AT91_PIN_PB21, 0);

    /* edge triggered interrupt
     * so just returning IRQ_HANDLED is ok
     */
    return IRQ_HANDLED;
}

int uio_write_gpio
    (struct uio_info *info,
     s32 irq_on)
{
    at91_set_gpio_value(AT91_PIN_PB21, 1);
    udelay(2);
    at91_set_gpio_value(AT91_PIN_PB21, 0);
    return 0;
}

struct uio_info uiointfo_latency = {
    .name = "at91_latency_measurement",
    .version = "0.0.1",
    .handler = uio_latency_handler,
    .irqcontrol = uio_write_gpio,
};

static struct platform_device
    uio_latency = {
    .name = "uio_pdrv",
    .id = -1,
    .dev.platform_data = &uiointfo_latency,
};

void __init
    at91_add_device_uio_latency(void)
{
    uiointfo_latency.irq =
```

```
    gpio_to_irq(AT91_PIN_PB19);
    /* configuring the gpio pins */
    at91_set_gpio_input(AT91_PIN_PB19, 0);
    at91_set_deglitch(AT91_PIN_PB19, 1);
    at91_set_gpio_output(AT91_PIN_PB21, 0);

    platform_device_register(&uio_latency);
}
```

The userspace task for the test environment looks like:

```
int32_t num_events = 0, old_events = 0;
int32_t irq_on = 1;
...
fd = open("/dev/uio0", O_RDWR);
...
while(1) {
    old_events = num_events;
    read(fd, &num_events,
         sizeof(num_events));
    if((num_events - old_events) > 1
        && !first_run)
        printf("Missed events\n");
    write(fd, &irq_on, sizeof(irq_on));
    if(first_run)
        first_run = 0;
}
```

The accuracy of a timer task can be benchmarked with cyclictst. Cyclictst is a high resolution timer test software, which has been initially written by Thomas Gleixner. It's part of the rt test tools (maintained by Clark Williams). Cyclictst is highly configurable and also includes some tracing options for ftrace and other kernel tracers. Since cyclictst covers several codepaths (we set up a timer - the systems programs the timer chip - the timer interrupt occurs - the system has to wake up the task) it gives a real good overview of the realtime performance of a system.

4.1.3 Test scenarios used for the measurements

Based on the scenarios shown in Chapter 4.1.1 and Chapter 4.1.2 we define the following test environment:

1. External events: A function generator will be set up to generate events with a frequency of 1000 Hz. A UIO driver and a userspace task will respond to those events. The interrupt response time and the time which is needed for two context switches will be measured. The Interrupt handler will be prioritized with SCHED_FIFO 99 and the userspace task with SCHED_FIFO 98.

2. Timer accuracy: Cyclicttest will be set up to create a timer task based on clock_nanosleep. The timer interval is 1ms, the priority is SCHED_FIFO 80.

For both test cases load will be generated with hackbench and with a "Ping Flooding" from a different machine.

4.2 ARM9

4.2.1 Target hardware

The measurements were taken on an Atmel AT91SAM9263 evaluation kit. Technical data:

- CPU: Atmel AT91SAM9263 (ARM926-EJS) @ 180MHz
- RAM: 64MB SD-RAM

4.2.2 Results

Figure 3 shows the result for the interrupt latency. The falling edge of the upper channel shows the interrupt event. The rising edge of the lower channel is triggered by the IRQ handler. The worst-case interrupt latency in a long-time run was at 248us.

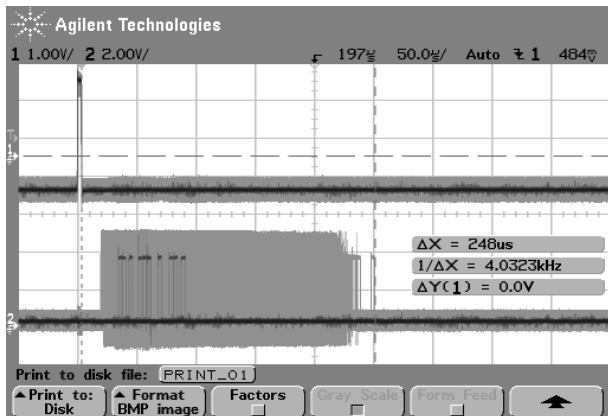


FIGURE 3: *IRQ latency ARM9*

Figure 4 combines the interrupt latency and the context switching time. The worst-case latency for waking up a userspace task and calling back into the kernel was 374us. The userspace task didn't miss

any interrupt events.

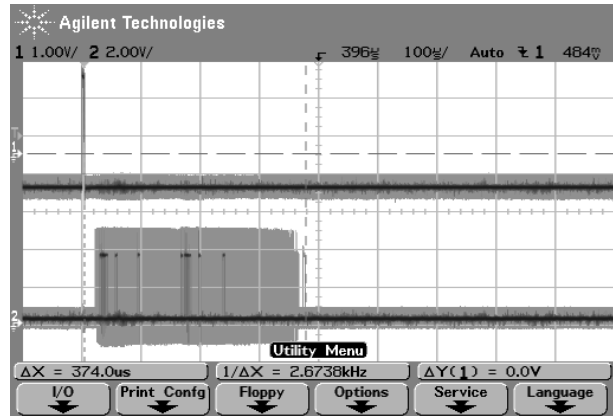


FIGURE 4: *context switching time on ARM9*

Figure 5 shows the histogram of the cyclicttest run. The average latency is at about 85us, the worst-case latency is 312us. In chapter 4.1.2 I mentioned, that cyclicttest gives a good overview of the realtime behaviour of a system. This cyclicttest run proves that theory once again, since the test run shows the same worst-case latencies as the interrupt latency measurements.

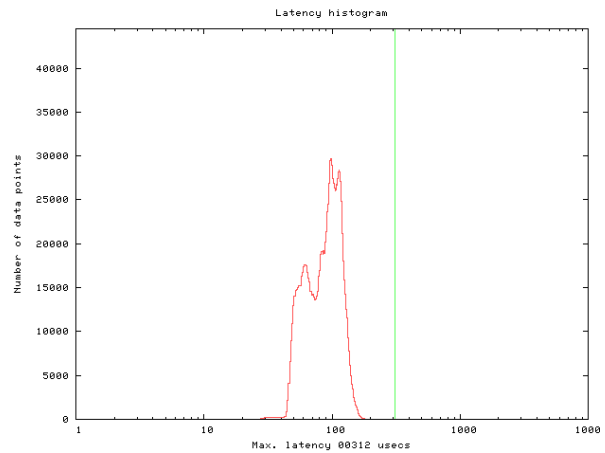


FIGURE 5: *High resolution timer on ARM9*

4.3 ARM11

4.3.1 Target hardware

These measurements were done on a Garz & Fricke Adelaide module, which is based on a Freescale i.mx31 CPU. Technical data:

- CPU: Freescale i.MX31 @ 532 Mhz
- RAM: 64 MB DDR RAM

4.3.2 Results

Figure 6 shows the results for the interrupt latency and the context switching time. The rising edge of the lower channel shows the occurrence of the irq event. The first peak on the upper channel shows the response of the UIO handler. The second peak on the upper channel shows the response which is generated by the write() call of the userspace task. A long-time test run showed a worst-case latency for the UIO handler of about 65us. The worst-case latency for the UIO handler + 2 context switches was about 110us. The application didn't miss an interrupt event.

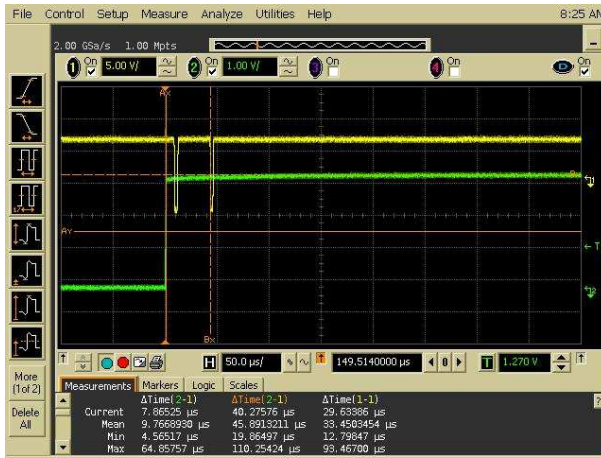


FIGURE 6: *IRQ latency and context switching time on ARM11*

Figure 7 shows the result for the cyclicttest run. It shows an average latency at 35us and a worst-case latency at 72us.

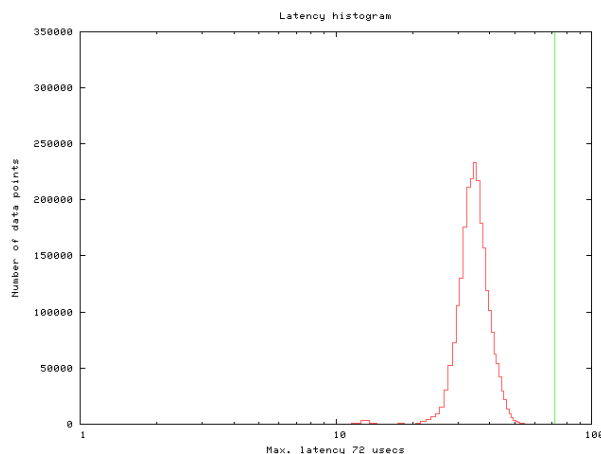


FIGURE 7: *High resolution timer on ARM11*

Since the results of those test were quite impressing, one more test scenario has been added for the

ARM11 case. The Interrupt frequency has been increased until the system became unstable. In this scenario the system kept being responsive up to an interrupt frequency of 150kHz! After reducing the frequency the system started to be responsive again! So the system didn't get into an unstable state. It was just busy servicing interrupts.

5 Conclusion

5.1 Analyzing the results

The benchmarking results on both platforms have proven a 100% deterministic behaviour of Preempt RT on ARM. So, Preempt RT on ARM CPUs can meet hard realtime requirements. Table 2 contrasts the benchmarking results of ARM9 with the results on the ARM11 platform.

	ARM9 (AT91SAM9263)	ARM11 (i.mx31)
IRQ latency	248us	68us
UIO wakeup	374us	110us
cyclicttest	312us	72us

TABLE 2: *Comparison of the worst-case latencies on ARM9 and ARM11*

It's quite obvious, that ARM11 can achieve much better latencies than ARM9. This is mainly related to the caching issue mentioned in chapter 2.1. The worst-case latencies on ARM11 are about 200us lower.

5.2 Decision guide

ARM9 based systems can achieve worst-case latencies of about 250us. If we are running a cyclic task, the worst-case latencies shouldn't exceed 25% of our cycletime. So, the lowest cycletime which can be achieved on ARM9 platforms is about 1ms. The worst-case latencies on ARM11 are much better. The worst-case latency is at about 100us. This should be sufficient for cycle times down to 400/500us. So, when worst-case latencies lower than 250us and cycletimes lower than 1ms are needed the system should be based on an ARM11 or a Cortex A8 design (which is designed to meet the requirements of the ARM11 platform). Figure 8 gives an overview of the boundary conditions for choosing an appropriate ARM design.

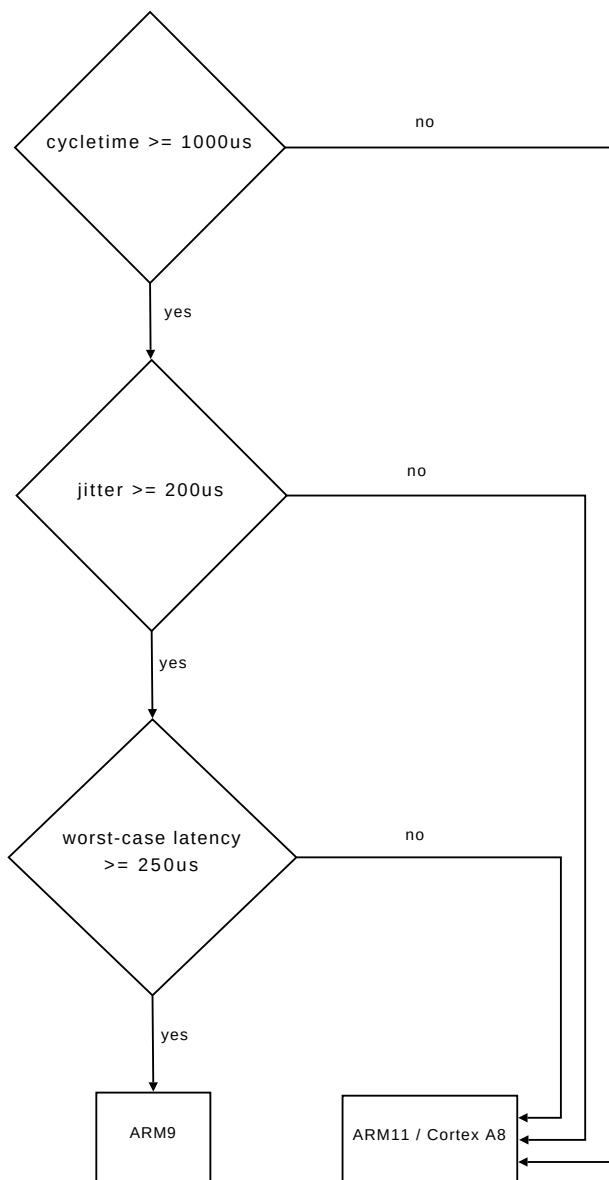


FIGURE 8: *Choosing an appropriate ARM CPU for Preempt RT*

References

- [1] The RT Wiki: http://rt.wiki.kernel.org/index.php/Main_Page
- [2] The OSADL Web Page: <http://www.osadl.org>
- [3] Richard Cochran's FCSE Patches: <http://www.opensubscriber.com/message/xenomai-core@gnat.org/7615242.html>
- [4] Gilles Chantepredrix' enhanced FCSE Patch: <http://article.gmane.org/gmane.linux.ports.arm.kernel/47943>
- [5] Building Embedded Linux Systems (2nd Edition) Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Phillippe Gerum, Steven Rostedt. Published by O'Reilly
- [6] The rt test tools: [git://git.kernel.org/pub/scm/linux/kernel/git/clrkwlms/rt-tests.git](http://git.kernel.org/pub/scm/linux/kernel/git/clrkwlms/rt-tests.git)

Faithful Virtualization on a Real-Time Operating System

Henning Schild

Adam Lackorzynski

Alexander Warg

Technische Universität Dresden
Department of Computer Science
Operating Systems Group

01062 Dresden

{hschild, adam, warg}@os.inf.tu-dresden.de

August 14, 2009

Abstract

The combination of a real-time executive and off-the-shelf time-sharing operating systems has the potential of providing both predictability and the comfort of a large application base. Isolation between the components is required to protect the real-time subsystem from a significant class of faults in the (ever-growing) time-sharing operating systems but also to protect real-time applications from each other. Recent commodity computer hardware significantly improved the ability of these machines to support faithful virtualization. Virtual machines provide the strong isolation required for security reasons. But questions regarding the temporal isolation remain open. In this paper we analyze how and to which degree recent x86 virtualization extensions influence the interrupt-response times of a real-time operating system hosting virtual machines.

1 Introduction

Reusing legacy time-sharing operating systems together with a real-time kernel is a common approach to get a large application base and predictability. Isolation in these systems ranges from shared-space systems, where real-time applications and the time-sharing operating system kernel reside in the most privileged mode, to separate-space systems, where only the real-time kernel runs in this mode, whereas all applications and the time-sharing operating system are deprivileged [10].

Separate-space systems have in common that paravirtualization is used to host the time-sharing operating system. Paravirtualization means that the time-sharing operating system is modified in a way that it can be executed next to the real-time kernel. The applicability of approaches based on this technique is limited by the need of source code access and by the considerable development and maintenance effort that the modifications require.

Recent commodity computer hardware has a significantly improved ability to support efficient virtu-

alization because virtualization support was added to the architecture[5][9]. With faithful virtualization unmodified guest operating systems can be executed in virtual machines. Using such virtual machines to host time-sharing operating systems next to the real-time executive does not only decrease the development effort, it also allows for the use of closed source operating systems, opening access to an even wider range of applications. The isolation provided by this virtual machine technology is comparable to the strong isolation provided by separate-space systems.

The usage of hardware extensions for virtualization has implications on the responsiveness of the host operating system. In this paper we analyze the degree to which the interrupt-response times of a real-time operating system can be influenced by using these extensions.

We will proceed as follows: Section 2 will revisit the fundamentals, before we go into details of hardware-assisted virtualization in section 3. Our experiments are described in section 4.

2 Background

In this section we briefly introduce virtualization techniques, the x86 hardware extensions for virtualization, and L4/Fiasco, the microkernel we used for our experiments.

2.1 Virtualization Techniques

Virtualization is a term that is used in many different aspects of computer science and especially in the area of operating systems. In this paper we concentrate on virtual machines that enable running complete operating systems with all their applications on top of the host operating system [12]. We focus on CPU and memory virtualization for the x86 architecture. There are three fundamental techniques to implement virtual machines: emulation, paravirtualization, and faithful virtualization.

2.1.1 Emulation

Emulation is a technique that translates guest code into host code. This can be accomplished by translating single guest instructions one after another as the guest execution proceeds. To improve the performance of emulators, techniques like binary translation and caching can be applied.

One advantage of emulation is that it can be applied when the guest and the host instruction set architectures (ISAs) are not the same. However, for identical ISAs it is desirable to execute the guest code directly instead of using emulation. Compared to virtualization techniques where guest code is executed directly, the performance of emulation is poor. Optimized emulators like QEMU [7] run a guest system multiple times slower than executing the same code on bare hardware.

Still, emulation is a common technique in cases where the guest can or must not be given direct access to the hardware. Emulation is often used to provide peripheral devices for virtual machines.

2.1.2 Paravirtualization

Paravirtualization is a technique where the guest operating system is modified in a way that it may run on top of another operating system. Privileged instructions and code sequences are exchanged with calls to the underlying operating system environment.

After this modification, the guest code is executed directly on the host CPU, the performance achieved by paravirtualization is close to executing the guest on bare hardware [6] [8].

Paravirtualization requires access to the source code of the guest operating system. Therefore it is applicable only to guest operating systems where the source code is accessible. Another disadvantage of paravirtualization is a considerable development and maintenance effort.

2.1.3 Faithful Virtualization

Faithful virtualization is to construct a virtual machine that allows most of the guest code to execute directly on the CPU. When guest code cannot be executed, emulation is used for these specific parts. The basic technique for faithful virtualization is called *trap and emulate*. The guest code is executed in a processor mode where privileged instructions must not be executed. Instead of executing these instructions, the processor signals a fault to the host operating system. These faults are called traps. Instructions that cause a trap are then executed using emulation. After a short emulation phase direct execution resumes in the deprivileged processor mode.

On x86 processors *trap and emulate* cannot be applied easily [13]. Some instructions do not cause a trap when they are executed in a deprivileged mode, although they would need to in order to preserve the illusion of a real CPU. However, recent x86 CPUs have virtualization extensions to overcome this problem. Using faithful virtualization, a single implementation of a virtual machine monitor (VMM) is suitable for a series of unmodified guest operating systems. Faithful virtualization yields good performance because most of the guest's code is executed directly.

2.2 Hardware-Assisted Faithful Virtualization

Hardware extensions found in recent processors make it easier to implement faithful virtualization for the x86 architecture. The second generation of these hardware extensions did not only help to reduce the software complexity required for virtual machines, it also helped to increase virtual machine performance significantly by providing support for memory virtualization through nested page tables. Hardware-assisted virtualization has a series of advantages making it a promising technology. It is widely available in commodity hardware and yields good perfor-

mance.

This section describes how hardware-assisted virtualization works. The provided details are required for the comprehension of the rest of this paper.

Two virtualization implementations are available, Intel VT[9] and AMD's SVM [5]. Because we conducted our experiments using SVM, we use AMD's notions throughout this document. However, the principles discussed in this section apply for Intel VT as well.

2.2.1 Processor Modes

The hardware extensions for virtualization allow executing privileged guest code directly on the CPU preserving the virtualization requirements. Two different modes of operation are supported by the processor.

The *host mode* is used for the host operating system. The *guest mode* is used to execute guest operating systems and their applications. Each mode supports all x86 privilege levels. Privileged instructions that access the CPU state in guest mode read or write the guest CPU state. Therefore guest operating system's code that expects to run on the highest privilege level may be executed at the level it expects. The isolation from the host is enforced by the hardware.

2.2.2 Control Flow

Before entering the guest mode, the host operating system needs to configure it. Therefore a control data structure for a virtual machine needs to be set up. This structure, the *VMCB*, contains the guest mode's CPU state and control information.

When guest code should be executed, the host operating system issues the *VMRUN* instruction to enter the guest mode. This instruction saves the host CPU state, loads the guest state provided by the *VMCB*, and begins execution in the guest mode. Interrupts triggered by host devices, faults caused by the guest code, or an attempt to execute certain instructions may lead to an automatic switch back to the host mode (*#VMEXIT*). Flags in the *VMCB* control which events or instructions exit the guest mode.

When switching back to the host mode, the guest CPU state is stored in the *VMCB* and the host state is automatically restored. After *#VMEXIT* has finished, the execution proceeds in the host mode until another switch to the guest mode is issued by the host.

2.3 The L4/Fiasco Microkernel

The L4/Fiasco microkernel is a small operating system kernel developed by our research group. In contrast to monolithic designs, microkernels aim at a minimal kernel. Functionality is only admitted into the kernel if it cannot be implemented at user-level without compromising security, or if a user-level implementation severely impacts performance. Components like device drivers, network protocol stacks, and filesystems are not part of the kernel, they are implemented on top of it. The reduced complexity at the kernel-level makes microkernel-based systems suitable for setups where strong security properties [14] and real-time capabilities [10] are required.

The L4/Fiasco microkernel provides a few basic mechanisms that can be used to construct complex systems. Virtualization allows us to reuse legacy software in our operating system. With L⁴Linux [8] we are able to execute Linux and unmodified Linux applications. The recently added support for hardware-assisted virtualization [11] enables us to reuse arbitrary unmodified x86 operating systems in a secure way, while reaching close to native performance and keeping the engineering effort low.

3 Hardware-Assisted Virtualization vs. Responsiveness

In this paper we are interested in the effects of hardware-assisted virtualization on the interrupt latency and therefore the real-time capabilities of the host operating system.

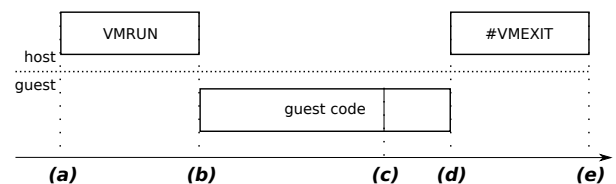


FIGURE 1: *Switching between processor modes*

The control flow described earlier is shown in figure 1. It highlights five points in time that are of particular interest.

- (a) Execution of the *VMRUN* instruction begins
- (b) CPU starts executing guest code
- (c) Execution of final instruction before *#VMEXIT* begins

- (d) Switching back to the host mode begins
- (e) Host operating system resumes

Switching between the guest and the host mode needs certain amounts of time during which the processor cannot accept interrupts. In order for the host to regain control, `VMRUN` always needs to be followed by `#VMEXIT`. We identified two possible sequences that are candidates for having the most significant influence on the host's interrupt-response times.

An interrupt triggers right after point (a). In this case no guest code is executed and `VMRUN` is directly followed by `#VMEXIT`. The duration of the context switch is at most $(b - a) + (e - d)$.

The interrupt triggers in the guest mode right after (c). The processor finishes the execution of the currently running instruction until (d) and issues a `#VMEXIT`. The maximum duration of the context switch is $(e - c)$ for this sequence.

Which of the two sequences takes more time than the other depends on the duration of its components. The duration of `VMRUN` and `#VMEXIT` is one of these components. We need to answer the question how long they occupy the CPU and whether these times are constant or depend on the guest state. Another important component is the maximum duration of instructions that may be executed in the guest mode, leaving the processor in an uninterruptible state.

4 Experimental Results

In this section we qualify the observations of the previous section with experimental results. For the experiments we used an AMD PhenomTM 9550 Quad-Core CPU on an ASUS M3A78-EM motherboard equipped with 2GB DDR2-800 RAM and a Samsung HD080HJ hard disk.

To analyze how using hardware-assisted virtualization increases the host's interrupt-response times we begin with a series of synthetic benchmarks, followed by application benchmarks that substantiate our results in more realistic scenarios. The application benchmarks were conducted using Linux with the `PREEMPT_RT` patch and L⁴Linux running on the L4/Fiasco microkernel.

4.1 Measuring Interrupt Latency in L4/Fiasco

Most of the tests were executed on the L4/Fiasco microkernel. We developed a tool to measure the

interrupt latency for an application on top of it.

This application sets up a periodic timer via the *High Precision Event Timer* (HPET) and attaches itself to the corresponding interrupt. The application then waits for the kernel to forward interrupts to it. When an interrupt triggers the microkernel schedules the application and delivers the interrupt. L4/Fiasco uses a scheduler based on fixed priorities. To make sure the test application is scheduled right after the interrupt arrives, the test program is configured to have the highest priority in the system.

This tool serves as our real-time application that handles external events, which are signaled via interrupts. The routine that handles the interrupts inside the application is used to measure the time it took to switch to this function. That is done by calculating the difference between the time the HPET was programmed to trigger an interrupt and the time this interrupt arrived in the user-level handler.

4.2 Analyzing Specific Mode Switching Scenarios

To answer the question of how long `VMRUN` and `#VMEXIT` take and whether the times depend on the guest state, we developed a simple virtual machine monitor. We used this application to set up various scenarios to measure their effect on the host's interrupt-response times.

With the help of this tool we found the duration of `VMRUN` and `#VMEXIT` to be dependant on various factors. Table 1 shows the maximum measured interrupt latencies for different test scenarios running on top of the L4/Fiasco microkernel. Some of them use virtualization with specific mode switching, others do not use hardware-assisted virtualization.

	setup	virtualization	time [μ s]
1	busy loop	–	7.05
2	vm simple	X	8.38
3	vm expensive	X	9.57
4	vm triplefault	X	12.78
5	cache	–	10.41
6	cache vm	X	12.5

TABLE 1: *Interrupt latencies for experimental setups on the L4/Fiasco microkernel.*

The baseline for the latencies is defined by the first setup ("busy loop"). In this setup an endless loop is executed next to the latency measurement. It does not make use of L4/Fiasco's virtualization capabilities. Running the simple VMM in an endless

loop where it keeps entering and leaving the guest mode ("vm simple"), the latency increases by 1.33 μs .

The setup "vm expensive" shows the results of the VMM that keeps entering and leaving the guest mode with a couple of conditions that increase the mode switch duration. In this scenario the VMM enters the guest mode injecting an exception and flushing the tagged TLBs. The guest's stack pointer is prepared in a way that the CPU needs to access two separate memory pages to store the exception return information. The *interrupt description table* (IDT) is set up, so that the injected exception causes another exception. Additionally it is aligned in a way that the CPU needs to access two distinct memory pages to read the entries required for the execution of the test. All these conditions increase the mode switch duration. Overall, this scenario has an interrupt latency that is 2.5 μs above "busy loop".

We stopped trying to increase the interrupt latency for the "vm expensive" scenario further after we came up with the "vm triplefault" setup. In the "vm triplefault" scenario the VMM keeps entering the guest mode with a condition that causes a triplefault in the guest. The VMM injects an exception that causes another exception, which causes a *shutdown* event [5] in the guest mode. In this setup no guest code is ever executed. After the `#VMEXIT` we never observed another exit reason than the *shutdown* event. Therefore we think this `VMRUN/#VMEXIT` combination is not interruptible. The interrupt latency for this setup is 12.78 μs , which is 5.73 μs above the baseline.

Like ordinary applications virtual machines influence the timing behaviour of the overall system through using the processor's caches. To show that, we developed another synthetic benchmark. Setup number 5 is an application that stresses the CPU's caches by writing to 32 MB contiguous memory. In the sixth setup ("cache vm") we executed the same code in the guest mode using our simple VMM. It can be seen that both setups increase the interrupt latency by using the caches. The increase that is caused by executing the code in a virtual machine is 2.1 μs , which is between the differences measured for the cheapest and the most expensive scenario.

From the results we conclude that the duration of the mode switches clearly depends on the guest state that is switched to. We measured increases from 1.33 up to 5.73 μs in host interrupt latency when SVM was used. Our results also confirm that virtual machines influence the timing through caches.

4.3 Long running Instructions

As described in section 3, we also need to analyze how long instructions in the guest mode may take to execute.

To find the durations of single instructions we used a manual provided by AMD [4]. It contains detailed information on the execution times of the individual instructions. Notably, there are instructions where the manual does not provide figures because their execution times are not fixed. All these instructions may be executed in the CPU's highest privilege level only. Therefore they do not pose a problem for real-time operating systems that execute applications in user-level. Developers can avoid using these long running instruction in the operating system kernel.

When hardware-assisted virtualization is used, such instructions can be executed in the highest privilege level of the guest mode. Because we want to be able to run legacy operating systems in the guest mode, we either need a mechanism to enforce that these instructions are not executed, or we will have to account for their duration in our worst case latency.

`WBINVD` is an example for one of these long running instructions. It writes data that was modified in the CPU's caches back to main memory. Its execution time depends on the amount of modified cache entries. For another experiment we used the virtual machine that writes to the contiguous memory region, which was used for our earlier cache experiments. We modified it to execute the `WBINVD` instruction after having written the 32 MB of memory.

The highest host interrupt latency we measured when executing this code in the guest mode was 643.66 μs . However, the `VMCB` can be used to control, which instructions must not be executed using instruction intercept flags. Instead of executing intercepted instructions, the processor issues a `#VMEXIT`. Intercepting the real-time critical instruction resulted in a worst case interrupt latency of 12.15 μs . This latency is close to the one measured for the "cache vm" setup that was modified.

The results of this experiment show that all real-time critical instructions should be intercepted if possible. SVM allows to intercept all long running instructions we found using the AMD manual. This means, the host can prevent the guest from executing them.

4.4 Interrupt Latency for L⁴Linux and KVM-L4

After running the synthetic benchmarks we analyzed the effects of using faithful virtualization in a more complex setup. In these experiments we did not specifically try to trigger special cases for virtualization. Instead we used application benchmarks to put the system under load. We compared two setups:

The first one was L4/Fiasco with L⁴Linux running on it. L⁴Linux was executing a Linux kernel build and the `lmbench3` benchmark suite.

In the second setup we added virtual machines using *Kernel-based Virtual Machine* [3] for L⁴Linux (KVM-L4). On the host, we ran the same benchmarks as in the first setup. As guests we used two virtual machines running Linux. One of them was running a Linux kernel build, the second virtual machine was executing a busy loop.

For the first scenario, where hardware-assisted virtualization was not involved, we measured a maximum latency of $29.05\ \mu\text{s}$. When we used hardware-assisted virtualization with KVM-L4 the maximum latency was $30.38\ \mu\text{s}$. Figure 2 shows the distribution of interrupt-response times for the two setups.

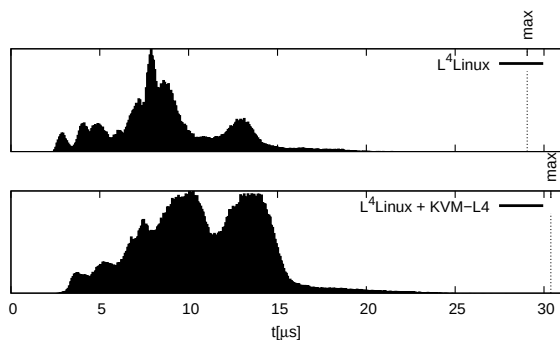


FIGURE 2: *Interrupt latency histograms for L⁴Linux without and with KVM-L4*

The results of these benchmarks show that even in a complex scenario, the influence of using virtualization is within the bounds we found with the synthetic benchmarks. The difference of the individual maxima is $1.33\ \mu\text{s}$, which does not exceed the $5.73\ \mu\text{s}$ increase we measured for the "vm triplefault" setup. The two maxima only show virtualization effects on the critical paths. They cannot be used to conclude that switching modes never took more than $1.33\ \mu\text{s}$ or $5.73\ \mu\text{s}$ in the setup using KVM-L4. But the histograms give confidence in the figures from the synthetic benchmarks. The shapes of the two graphs are similar. The one that displays the KVM-L4 setup is

shifted to the right by $1.5 - 2\ \mu\text{s}$ and contains more samples in the rightmost spike.

4.5 Linux with *PREEMPT_RT* patch

So far we only presented results from experiments using L4/Fiasco. But the figures should be applicable to other implementations as well. The mode switch durations are mainly implied by hardware, instead of software. To show that the measured latency penalty applies for other implementations as well, we conducted some experiments with Linux. For our measurements we used Linux 2.6.29.6 and applied the *PREEMPT_RT* patch set version rt23 [2]. To measure the latency we used the `cyclictst` application [1] developed by Thomas Gleixner.

Again, we set up two scenarios with the same load as in the L⁴Linux setups. For the first scenario, which did not involve KVM, we measured a maximum latency of $52\ \mu\text{s}$. When we used hardware-assisted virtualization with KVM the maximum latency was $55\ \mu\text{s}$. The results of these experiments are given in figure 3. Again the graphs have a similar shape. Using hardware-assisted virtualization through KVM increases the latency by $3\ \mu\text{s}$. This result meets the expectations derived from the earlier experiments. The cost of using the hardware extension for virtualization applies for Linux/KVM as well. It is within the bounds we expected it to be.

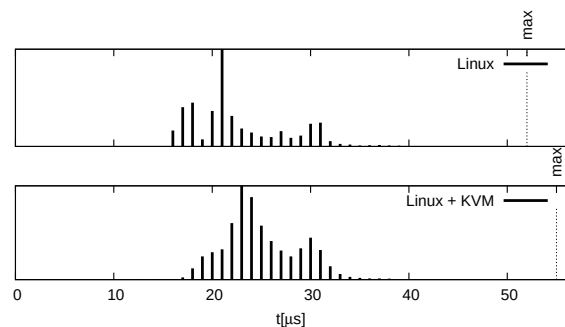


FIGURE 3: *Interrupt latency histograms for Linux with *PREEMPT_RT* patch, without and with KVM*

5 Conclusion

Hardware assisted faithful virtualization is an easy and efficient way of running legacy third party applications on top of a real-time operating system. We explained how it works and how virtual machines might influence the responsiveness of the host operating system. In our experiments, using the hardware extension SVM increased the interrupt latency

of the real-time operating system by at most $5.73 \mu\text{s}$ on a 2.2 GHz machine. We also showed that virtual machines may cause a significantly bigger increase when they are not configured with attention to real-time. The host operating system needs to make sure that long running instructions cannot be executed in guest mode.

6 Acknowledgement

We would like to thank Björn Döbel, Marcus Völp, and Jean Wolter for their constructive criticism on the way to this version of the paper. Authors of this paper have been supported by the European Research Programme FP7.

References

- [1] Cyclictest . <http://rt.wiki.kernel.org/index.php/Cyclictest>, 2009-06-17.
- [2] CONFIG PREEMPT RT Patch . http://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch, 2009-07-29.
- [3] Kernel Based Virtual Machine . <http://www.linux-kvm.org/>, 2009-08-11.
- [4] Advanced Micro Devices. *AMD AthlonTM Processor, x86 Code Optimization Guide*, publication no. 22007, rev. k edition, 2002.
- [5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, rev 3.14 edition, 2007.
- [6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.
- [7] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [8] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ -kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM.
- [9] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2*, 253669-028us edition, 2008.
- [10] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, Dec. 2002.
- [11] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, Nuremberg, Germany, 2009. ACM.
- [12] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [13] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.
- [14] M. Roitzsch and H. Härtig. Ten Years of Research on L4-Based Real-Time Systems. In *Proceedings of the Eighth Real-Time Linux Workshop*, Lanzhou, China, 2006.

Deterministic Synchronization in Multicore Systems: the Role of RCU

Paul E. McKenney

IBM Linux Technology Center

15350 SW Koll Parkway, Beaverton, OR 97006 USA

paulmck@linux.vnet.ibm.com

Abstract

Although real-time operating systems and applications have been available for multicore systems for some years, shared-memory parallel systems still pose some severe challenges for real-time algorithms, particularly as the number of CPUs increases. These challenges can take the form of lock contention, memory contention, conflicts/restarts for lockless algorithms, as well as many others. One technology that has been recently added to the real-time arsenal is a preemptible implementation of read-copy update (RCU), which permits deterministic read-side access to read-mostly data structures, even in the face of concurrent updates. In some cases, updates may also be carried out in a deterministic manner.

1 Introduction

RCU is a synchronization mechanism that was accepted into the Linux kernel in late 2002, though precursors to RCU date back to at least 1980 [4], with a more complete history available elsewhere [6, Section 2.2.20 and Table 3.2]. RCU improves scalability by permitting reads to run concurrently with updates. This is a departure from most prior synchronization primitives, for example, conventional locking primitives allow only one access at a time, and even reader-writer locking primitives, while allowing multiple concurrent readers, do not permit readers to run concurrently with updaters. More recent primitives, such as seqlock, appear to allow concurrent readers and updaters, but such readers are always retried. Research work on transactional memory permits concurrent readers and updaters, but conflicting accesses will result in rollback and retry. Although it is possible to construct real-time systems using blocking and rollback-retry primitives, primitives with better determinism would be desirable.

In contrast, RCU enables readers and updaters to run concurrently, with no blocking and no rollback-retry. Updaters handle conflicts with con-

current readers by maintaining multiple versions of data structures, and ensuring that old versions are not freed up (or, more generally, *reclaimed*) until all pre-existing readers (which might have references to the old versions) have completed. RCU also provides efficient and scalable mechanisms for publishing and subscribing to new data structures, and also for retracting previously published data, by deferring reclamation of old data structures that have since been rendered inaccessible to readers.

The RCU mechanisms distribute work among readers and updaters so as to make read paths extremely fast. In fact, in some cases, but unfortunately for neither real-time kernels nor user applications, RCU's read-side primitives can have zero overhead. In addition, as we will see, RCU is somewhat specialized, being intended primarily to protect read-mostly data structures.

Section 2 gives a brief overview of RCU, Section 3 recapitulates the progress made over the past five years towards a high-performance real-time implementation of RCU, Section 4 discusses where one might use RCU to meet real-time latency requirements, and finally, Section 5 presents concluding remarks.

2 What is RCU?

At its most basic level, RCU is a way of waiting for other things to get done. An RCU implementation must implement three fundamental operations [17]:

1. Updaters publish new data in presence of concurrent readers.
2. Readers subscribe to existing data in presence of concurrent updaters.
3. Updaters retract data previously published in presence of concurrent readers.

The retraction operation is particularly interesting, as it is not possible to retract data that a reader is currently subscribed to, given that readers cannot be blocked, rolled back, or aborted. Instead, updaters first make the data inaccessible to new readers, and then wait for a “grace period” to elapse, with the grace period being sufficiently long to permit all pre-existing readers to finish with their subscriptions. Once this grace period has completed, there will be no subscribers to the data, so that destructive operations (such as freeing) may safely be carried out.

Publication and subscription can be implemented deterministically, and are thus prime candidates for real-time use. Retraction’s long grace periods often rules it out for use on real-time critical code paths, but it is possible to use asynchronous primitives in cases where critical code paths need to start a grace period, but need not wait for it to finish.

These fundamental operations can be used to implement a reader-writer locking replacement, a bulk reference counter, a “poor man’s” garbage collector, and a form of existence guarantee, in many cases with an extremely low and deterministic overhead [10]. Each of these three operations is described in the following sections.

2.1 RCU Publication

RCU publication uses the `rcu_assign_pointer()` primitive. This primitive publishes a pointer to a structure, and guarantees that any initialization carried out prior to publication is seen by any RCU reader subscribing to the newly published structure.

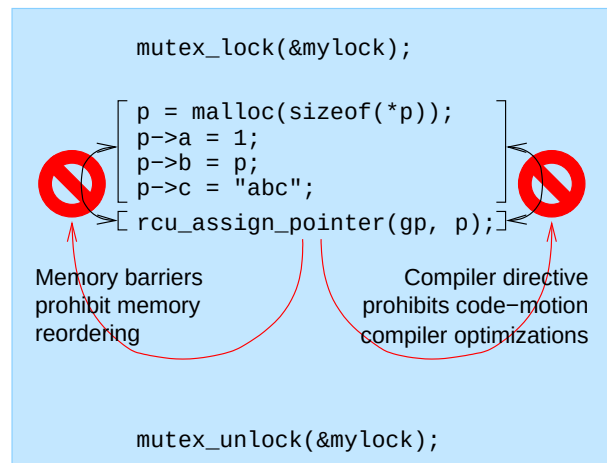


FIGURE 1: RCU Publication Constraints

The `rcu_assign_pointer()` primitive is a normal pointer assignment (hence the name), but possibly augmented with memory barriers and compiler directives. Such directives are often required in order to prohibit CPU or compiler optimizations that would otherwise cause previous initializations to be seen by RCU readers as occurring after the pointer assignment. An example of these prohibitions is shown in Figure 1. Please note that the need for such constraints is not specific to RCU. For example, locking primitives typically use memory barriers and compiler directives in order to prevent the CPU and the compiler from moving code out of the lock’s critical section.

RCU may be used to publish into more complex data structures, including linked lists, trees, and hash tables. The insertion primitives for such RCU-protected data structures, for example, `list_add_rcu()`, contain carefully placed calls to `rcu_dereference()`.

Note that RCU coordinates among a single updater and concurrent readers. Some other synchronization primitive (typically locking) must be used to coordinate among concurrent updaters, as exemplified by the `mutex_lock()` and `mutex_unlock()` in Figure 1.

2.2 RCU Subscription

RCU subscription uses the `rcu_dereference()` primitive to subscribe to structures published by `rcu_assign_pointer()`, returning a pointer to the specified structure. However, two additional primitives, `rcu_read_lock()` and `rcu_read_unlock()`,

are also required to bound the section of code throughout which the subscribed pointer may safely be dereferenced. Such sections of code are called “RCU read-side critical sections”.

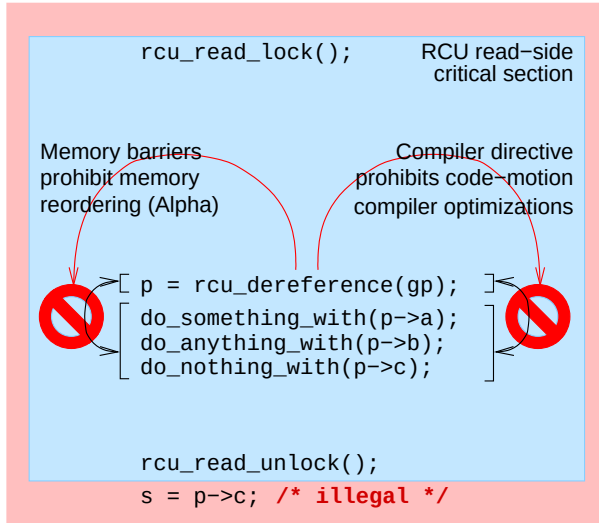


FIGURE 2: *RCU Subscription Constraints*

The `rcu_dereference()` primitive also acts as a normal pointer assignment, but again augmented with compiler directives and memory barriers. However, in contrast with `rcu_assign_pointer()`, `rcu_dereference()` requires memory barriers only when running on DEC Alpha. As with `rcu_assign_pointer()`, the directives and barriers associated with `rcu_dereference()` prohibit CPU and compiler optimizations that could otherwise permit the reader to see pre-initialed data, as shown in Figure 2.

Optimizations capable of disrupting users of `rcu_dereference()` can be quite counter-intuitive. An example optimization involves pointer-value speculation, which can result in the following sequence of events:

1. CPU 0 executes code that speculates the value that will be returned by `rcu_dereference()`, but incorrectly guesses a pointer that currently points into the freelist.
2. CPU 0 continues blithely executing `do_something_with()`, `do_anything_with()`, and even `do_nothing_with()`, making wild references into unallocated memory.
3. CPU 1 decides to publish a new data structure, and has the bad luck to have `malloc()` return exactly the value that CPU 0 guessed.

4. CPU 1 initializes the fields of the newly allocated structure, then uses `rcu_assign_pointer()` to publish it.
5. CPU 0 now validates the speculated value, finds that it matches, and therefore incorrectly concludes that its speculation was OK.

The `rcu_dereference()` primitive must prevent such speculation, as shown in Figure 2. In some (but not all!) RCU implementations, the `rcu_read_lock()` and `rcu_read_unlock()` primitives must also constrain the compiler and CPU.

As with `rcu_assign_pointer()`, `rcu_dereference()` is used in access primitives of more complex data structures, for example, the `list_for_each_entry_rcu()` list-traversal primitive.

2.3 RCU Retraction

RCU retraction is a three-step process that typically makes use of a data-element-removal primitive such as `list_del_rcu()`, the `synchronize_rcu()` primitive, and some reclamation primitive such as `free()`. This retraction process is shown in Figure 3, along with the relationship between retraction and subscription.

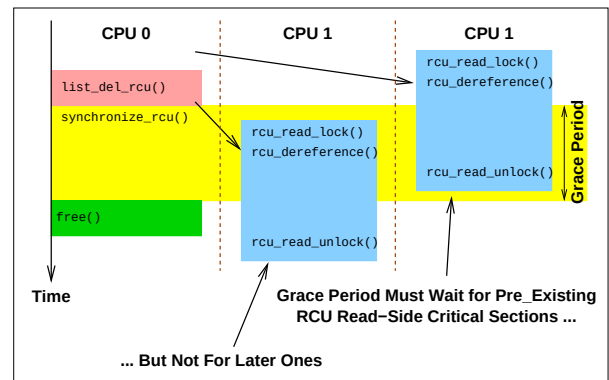


FIGURE 3: *RCU Retraction*

A key point about the retraction process is that only pre-existing readers can possibly hold a reference to the newly removed element. In the figure, the right-most reader might reference the data element, due to the fact that the reader started execution before the element was removed. In contrast, the left-most reader cannot possibly hold a reference, as it did not start execution until the removal step completed. RCU’s `synchronize_rcu()` primitive takes

advantage of this distinction, waiting only for pre-existing RCU readers, not for any that start execution afterwards.

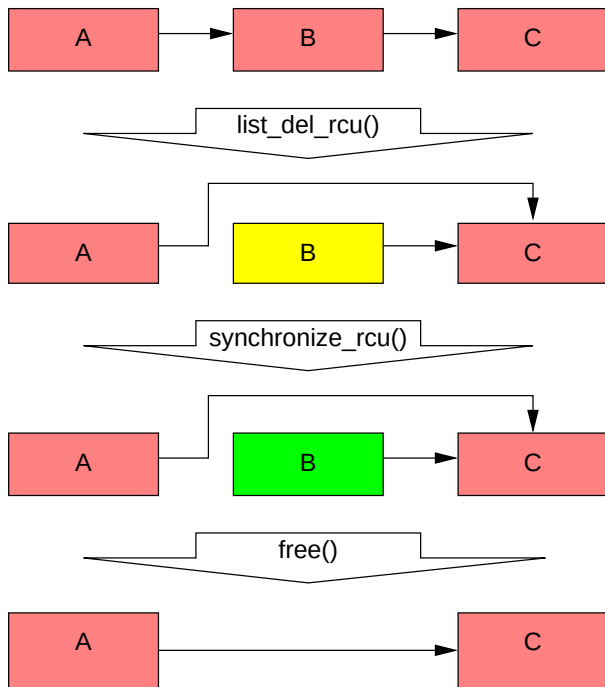


FIGURE 4: *RCU Retraction For Deletion From List*

Figure 4 shows how RCU retraction may be used to safely remove element B from a linked list containing elements A, B, and C. Initially, readers might have subscribed to (and thus might hold references to) each of these elements, signified by their red color. The `list_del_rcu()` primitive unlinks element B from the list, but leaves the element itself intact, including the pointer to element C, as shown by the second row of Figure 4. This permits readers holding concurrent references to element B to proceed normally, but prevents any new readers from acquiring such a reference, as signified by element B's yellow color. Therefore, pre-existing readers might still see the list as {A, B, C}, but because new readers have no way to gain a reference to element B, they will see only {A, C}.

The `synchronize_rcu()` primitive then waits for all pre-existing concurrent readers to exit their RCU read-side critical sections, in other words, `synchronize_rcu()` waits for a grace period to complete. Because RCU readers' subscriptions are not valid outside of an RCU read-side critical section, once all such pre-existing RCU read-side critical sections have completed, there can be no readers holding references to element B, as signified by its green color in the third row.

Because no readers can possibly hold a reference to element B, it is now safe for the updater to carry out destructive operations, in this case, `free()`. This leaves the list containing only elements A and C, as shown in the last row of Figure 4.

Retraction as described above is normally not used in critical real-time code paths due to the fact that grace periods can extend for many milliseconds. However, there is an asynchronous counterpart to the `synchronize_rcu()` primitive named `call_rcu()`. The `call_rcu()` primitive registers a function and argument to be invoked at the end of a subsequent grace period, but incurs only queuing overhead on the critical code path. In addition, the advent of expedited RCU grace periods might enable some real-time applications to place expedited grace periods on critical code paths using the new `synchronize_rcu_expedited()` primitive [11].

2.4 Discussion

A large number of concurrent algorithms may be constructed using RCU's publication, subscription, and retraction primitives. However, such algorithms are useful only if they provide adequate performance, scalability, and real-time response. Because RCU is designed primarily for read-mostly (or subscription-mostly) situations, the next section will give a rough feel for the overhead of RCU's read-side primitives.

3 Towards a High-Performance Real-Time RCU Implementation

This section gives a quick overview of some Classic RCU and Preemptable RCU read-side primitives, followed by some references to work on user-level RCU implementations. This section is only intended to give a rough feel for the overhead of these primitives, but citations are given to direct the reader to additional information.

3.1 Classic RCU

Server-class (!CONFIG_PREEMPT) builds of the Linux kernel use the extremely efficient implementations for `rcu_read_lock()` and `rcu_read_unlock()` shown in Table 1. It is hard to imagine an implementation with less overhead.

Unfortunately, !CONFIG_PREEMPT kernel builds

```

1 static inline void rcu_read_lock(void)
2 {
3 }
4
5 static inline void rcu_read_unlock(void)
6 {
7 }

```

Table 1: Server-Class RCU Read-Side Primitives

```

1 static inline void rcu_read_lock(void)
2 {
3     preempt_disable();
4 }
5
6 static inline void rcu_read_unlock(void)
7 {
8     preempt_enable();
9 }

```

Table 2: Desktop-Class RCU Read-Side Primitives

have unimpressive real-time capabilities, so `CONFIG_PREEMPT` desktop-class real-time support was added during the Linux 2.5 kernel development effort. This adds some overhead to the RCU read-side primitives, as can be seen in Table 2. These primitives are still very light-weight, updating a field in the local `thread_info` structure, and, in the case of `rcu_read_unlock()`, also checking to see if the scheduler needs to be invoked, and invoking it if so. However, the fact that preemption is disabled for the duration of each RCU read-side critical section does not help real-time response.

This situation lead to the development of Preemptable RCU.

3.2 Preemptable RCU

The read-side primitives for preemptable RCU are shown in Table 3. These primitives have been described fully elsewhere [15], so we will instead simply count the number of atomic instructions and memory barriers, which can be as many as two each for both `rcu_read_lock()` and `rcu_read_unlock()`, for a total of up to four for a given read-side critical section. To add insult to injury, both primitives also disable interrupts.

Table 4 shows the current mainline implementation of the RCU read-side primitives. Again, these have been described in detail elsewhere [7], so we will instead count expensive operations. There are

no atomic instructions or memory barriers, but these primitives still disable interrupts. In addition, there are nine primitives that constrain the compiler and quite a bit more code. So while the current mainline preemptable-RCU implementation is a decided improvement over the early implementation, it is still quite heavyweight compared to the non-preemptable implementations shown above.

It turns out to be possible to further reduce read-side overhead, as shown in Table 5. This implementation does not use atomic instructions, memory barriers, or interrupt disabling, and has only five constraints on the compiler. Furthermore, this experimental implementation executes only four lines of C code in the common case, many fewer than the earlier implementations. The uncommon case is quite heavyweight, but is invoked only in exceptional circumstances, such as when the read-side critical section actually did block or was preempted—in which case the overhead of the resulting context switches dominates the overhead [12]. The overhead of this variant of `rcu_read_lock()` and `rcu_read_unlock()` can be expected to be roughly that of the desktop-class variant shown in Table 2.

3.3 Catalog of User-Level RCU Implementations

Of course, a real-time RCU implementation in an operating-system kernel is all well and good, but it does not provide much help to real-time applications.

<pre>1 void rcu_read_lock(void) 2 { 3 int f; 4 unsigned long oldirq; 5 struct task_struct *t = current; 6 7 raw_local_irq_save(oldirq); 8 if (t->rcu_read_lock_nesting++ == 0) { 9 f = rcu_ctrlblk.completed & 1; 10 smp_read_barrier_depends(); 11 t->rcu_flipctr1 = 12 &(__get_cpu_var(rcu_flipctr)[f]); 13 atomic_inc(t->rcu_flipctr1); 14 smp_mb__after_atomic_inc(); 15 if (f != (rcu_ctrlblk.completed & 1)) { 16 t->rcu_flipctr2 = 17 &(__get_cpu_var(rcu_flipctr)[!f]); 18 atomic_inc(t->rcu_flipctr2); 19 smp_mb__after_atomic_inc(); 20 } 21 } 22 raw_local_irq_restore(oldirq); 23 }</pre>	<pre>1 void rcu_read_unlock(void) 2 { 3 unsigned long oldirq; 4 struct task_struct *t = current; 5 6 raw_local_irq_save(oldirq); 7 if (--t->rcu_read_lock_nesting == 0) { 8 smp_mb__before_atomic_dec(); 9 atomic_dec(t->rcu_flipctr1); 10 t->rcu_flipctr1 = NULL; 11 if (t->rcu_flipctr2 != NULL) { 12 atomic_dec(t->rcu_flipctr2); 13 t->rcu_flipctr2 = NULL; 14 } 15 } 16 raw_local_irq_restore(oldirq); 17 }</pre>
---	---

Table 3: Early Preemptable RCU Read-Side Primitives

<pre>1 void __rcu_read_lock(void) 2 { 3 int idx; 4 struct task_struct *t = current; 5 int nesting; 6 7 nesting = ACCESS_ONCE(t->rcu_read_lock_nesting); 8 if (nesting != 0) { 9 t->rcu_read_lock_nesting = nesting + 1; 10 } else { 11 unsigned long flags; 12 13 local_irq_save(flags); 14 idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1; 15 ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])++; 16 ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1; 17 ACCESS_ONCE(t->rcu_flipctr_idx) = idx; 18 local_irq_restore(flags); 19 } 20 }</pre>	<pre>1 void __rcu_read_unlock(void) 2 { 3 int idx; 4 struct task_struct *t = current; 5 int nesting; 6 7 nesting = ACCESS_ONCE(t->rcu_read_lock_nesting); 8 if (nesting > 1) { 9 t->rcu_read_lock_nesting = nesting - 1; 10 } else { 11 unsigned long flags; 12 13 local_irq_save(flags); 14 idx = ACCESS_ONCE(t->rcu_flipctr_idx); 15 ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1; 16 ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])--; 17 local_irq_restore(flags); 18 } 19 }</pre>
---	---

Table 4: Current Mainline Preemptable RCU Read-Side Primitives

```

1 void __rcu_read_lock(void)
2 {
3     ACCESS_ONCE(current->rcu_read_lock_nesting)++;
4     barrier();
5 }
6
7 void __rcu_read_unlock(void)
8 {
9     struct task_struct *t = current;
10
11     barrier();
12     if (--ACCESS_ONCE(t->rcu_read_lock_nesting) == 0 &&
13         unlikely(ACCESS_ONCE(t->rcu_read_unlock_special)))
14         rcu_read_unlock_special(t);
15 }

```

Table 5: Experimental Preemptable RCU Read-Side Primitives

Fortunately, there have been a number of user-level RCU algorithms put forward over the past five years.

In 2006, Hart et al. [3] used a user-level variant of Classic RCU for benchmarking purposes. This implementation is sufficient for real-time applications whose architectures guarantee that each thread will periodically pass through a quiescent state.

In 2008, McKenney put forward a number of user-level RCU implementations, primarily for the purpose of debugging Linux kernel code at user level and to serve as an introduction to the more-complex implementations in the Linux kernel [8, 13]. These algorithms, though simple, all either impose restrictions on the application or suffer from high read-side overhead.

In 2009, Desnoyers adapted preemptable RCU for user-level use by substituting POSIX signals for the scheduling-clock interrupt [2]. This approach allows good read-side overhead combined with minimal restrictions on the application (namely, that it be willing to give up a signal).

Given this recent progress, it seems safe to say that RCU is ready to take on user-level applications, including real-time applications.

4 Where to Use RCU

Details on the use of RCU has been discussed at length [1, 9, 10], although continued progress indicates that there is still much more to be learned. This section will instead focus on the situations to which RCU is best applied, as summarized by Figure 5.

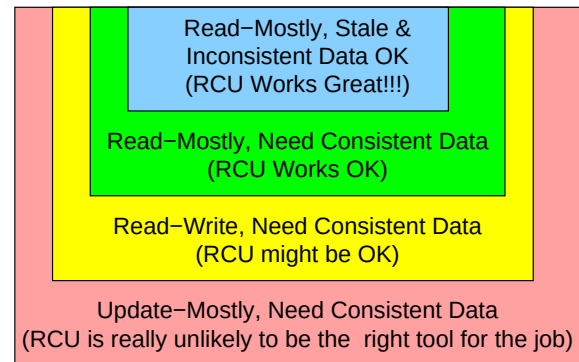


FIGURE 5: RCU Areas of Applicability

The situation most favorable to RCU is for read-mostly data structures where stale and inconsistent data can be tolerated. The prototypical use is routing tables [14], which maintain in-memory state that represents outside-of-system connectivity. Changes in outside-of-system connectivity result in routing-table updates, but these changes often take significant time to propagate to the system. This propagation delay means that the system has been mis-routing for some tens of seconds or even minutes, so that the additional few milliseconds required to retract the old data is not significant. The benefit is that readers gain deterministic access to the data.

RCU can also be quite helpful for read-mostly data structures where consistent and fresh data is required. For example, in the Linux kernel's System V semaphore implementation, RCU protects the data structures mapping from the user-mode `semid` to the in-kernel `sem_array` structure, but the structure itself is protected by a spinlock in the enclosed

`kern_ipc_perm` structure [1]. This permits deterministic mapping from `semid` to `sem_array`, while the spinlock reintroduces freshness and consistency, but at the level of an individual `sem_array` structure rather than globally.

RCU is less likely to be helpful for read-write data structures where consistent and fresh data is required. That said, the dcache system is one such example, where RCU is used to map from a pathname segment and parent dentry to the child dentry. Per-dentry locking and a global seqlock are used to impose freshness and consistency to pathname lookups [5, 16].

Finally, if the data structure is updated more often than it is read, and if consistent and fresh data are required, RCU is quite unlikely to be the right tool for the job.

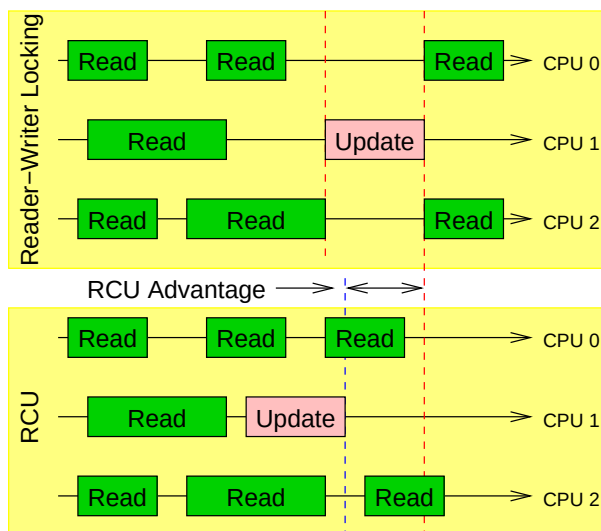


FIGURE 6: *Staleness for Reader-Writer Locking and RCU*

A surprising number of algorithms tolerate access to stale data. A key question to ask is “what defines stale?” It is natural to assume that the data is by definition fresh when it first arrives at the system, however, communication latencies often invalidate this assumption. When data is stale upon arrival, as it was in the routing-table example above, RCU can often provide greater freshness than can other synchronization primitives such as reader-writer locking. This can be seen in Figure 6, where reader-writer locking delays the update, thus resulting in the update happening later than for RCU. Therefore, if the freshness of the data is defined externally, use of reader-writer locking can result in data being more stale than for RCU.

Additional information on uses of RCU from the

Linux kernel may be found in McKenney’s dissertation [6, Chapter 6].

5 Conclusions

The past year has seen a number of user-level RCU implementations that promise to be useful in real-time applications, and also the beginnings of a much simpler and faster implementation of in-kernel preemptable (real-time) RCU. These developments promise to bring new options for real-time applications running on multi-core hardware.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Dietrich, K.R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. We all owe Darren Hart a debt of gratitude for his efforts to render this document human-readable. Finally, I owe thanks to Kathy Bennett for her support of this effort.

Legal Statement

This work represents the views of the authors and does not necessarily represent the view of IBM.

Linux is a copyright of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

References

- [1] ARCANGELI, A., CAO, M., MCKENNEY, P. E., AND SARMA, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310. Available: <http://www.rdrop.com/users/paulmck/RCU/rcu.FREENIX.2003.06.14.pdf> [Viewed November 21, 2007].
- [2] DESNOYERS, M. [RFC git tree] userspace RCU (urcu) for Linux. Available: <http://lkml.org/lkml/2009/2/5/572> git://ltnng.

- org/userspace-rcu.git [Viewed February 20, 2009], February 2009.
- [3] HART, T. E., MCKENNEY, P. E., AND BROWN, A. D. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium* (Rhodes, Greece, April 2006). Available: http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf [Viewed April 28, 2008].
 - [4] KUNG, H. T., AND LEHMAN, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems* 5, 3 (September 1980), 354–382. Available: <http://portal.acm.org/citation.cfm?id=320619&dl=GUIDE>, [Viewed December 3, 2007].
 - [5] LINDER, H., SARMA, D., AND SONI, M. Scalability of the directory entry cache. In *Ottawa Linux Symposium* (June 2002), pp. 289–300.
 - [6] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: <http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf> [Viewed October 15, 2004].
 - [7] MCKENNEY, P. E. The design of preemptible read-copy-update. Available: <http://lwn.net/Articles/253651/> [Viewed October 25, 2007], October 2007.
 - [8] MCKENNEY, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2008. Available: [git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git](http://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git) [Viewed August 15, 2009].
 - [9] MCKENNEY, P. E. RCU part 3: the RCU API. Available: <http://lwn.net/Articles/264090/> [Viewed January 10, 2008], January 2008.
 - [10] MCKENNEY, P. E. What is RCU? part 2: Usage. Available: <http://lwn.net/Articles/263130/> [Viewed January 4, 2008], January 2008.
 - [11] MCKENNEY, P. E. [PATCH -tip 0/3] expedited 'big hammer' RCU grace periods. Available: <http://lkml.org/lkml/2009/6/25/306> [Viewed August 16, 2009], June 2009.
 - [12] MCKENNEY, P. E. [PATCH RFC -tip 0/4] RCU cleanups and simplified preemptable RCU. Available: <http://lkml.org/lkml/2009/7/23/294> [Viewed August 15, 2009], July 2009.
 - [13] MCKENNEY, P. E. Using a malicious user-level RCU to torture RCU-based algorithms. In *linux.conf.au 2009* (Hobart, Australia, January 2009). Available: <http://www.rdrop.com/users/paulmck/RCU/urcutorture.2009.01.22a.pdf> [Viewed February 2, 2009].
 - [14] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: <http://www.linuxsymposium.org/2001/abstracts/readcopy.php> http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf [Viewed June 23, 2004].
 - [15] MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. Extending rcu for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184 <http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf> [Viewed January 1, 2007].
 - [16] MCKENNEY, P. E., SARMA, D., AND SONI, M. Scaling dcache with RCU. *Linux Journal* 1, 118 (January 2004), 38–46.
 - [17] MCKENNEY, P. E., AND WALPOLE, J. What is RCU, fundamentally? Available: <http://lwn.net/Articles/262464/> [Viewed December 27, 2007], December 2007.

The ARM Fast Context Switch Extension for Linux

Gilles Chanteperdrix

gilles.chanteperdrix@xenomai.org

Richard Cochran

OMICRON electronics GmbH

Oberes Ried 1, A-6833 Klaus

richard.cochran@omicron.at

Abstract

The ARMv5 CPUs are inexpensive, low power, 32 bit processors widely used in embedded systems. Because of these processors' cache implementation, using memory protection on these systems incurs a performance penalty too large for many real time applications. By implementing the Fast Context Switch Extension, we achieved improved cache performance while retaining memory protection under the Linux 2.6 kernel. We briefly discuss the problem, explain the necessary changes to the Linux memory management system, and present performance measurements taken from artificial and real world applications.

1 Introduction

The ARMv5 architecture specifies virtually indexed, virtually tagged (VIVT) cache units. This means that the central processing unit (CPU) accesses the memory cache by means of virtual addresses. When the CPU issues a load or store instruction, the desired virtual address is presented to the memory management unit. If this address is present in the cache, then the unit immediately returns the cached data to the CPU. In a memory protected, multi-tasking operating system like Linux, each process has its own virtual address space. Consequently, the cache must be invalidated during a context switch in order to ensure that each process can only access its own data.

As a result of the cache invalidation, each time a process is scheduled, it must reload all its working data from the main memory. Reading data from main memory is an expensive operation. Depending on the particular CPU type, the memory speed, and the program's data access pattern, the cost of the cache invalidation during a context switch can be on the order of 200 microseconds. This performance penalty is great enough to preclude using such CPU types for certain time critical applications, for example when the reaction time to external events is required to be less than one millisecond.

One approach to address this problem, used for example in vxWorks or μ Clinux, is to use one "flat"

address space, shared by the operating system kernel and all user processes. This avoids the context switch penalty while sacrificing memory protection. The obvious drawback under this scenario is that any one misbehaving program can corrupt the kernel or other programs.

Another solution is to use the Fast Context Switch Extension (FCSE) available on ARMv5 processors. The FCSE may be understood as a workaround to the VIVT cache, where a single virtual address space is divided up among several processes. This is done in such a way that no two processes share the same range of virtual addresses, thus obviating the need to invalidate the cache during a context switch.

The work described in this paper implements the FCSE under Linux kernel 2.6, removing some of the performance problems associated with the VIVT cache. Our goal was to create a minimally intrusive solution that would be viable for acceptance into the main line kernel.

2 Previous Work

Earlier work has highlighted the context switch performance problem running Linux on ARMv5 computers. Hyok-Sung Choi and Hee-Chul Yun [1] report dramatically improved performance when moving from plain Linux to μ Clinux. David, Carlyle,

and Campbell [2] instrumented a Linux 2.6 kernel to investigate the direct and indirect latency caused by context switches while running four different algorithms. They found that 99 context switches during a three to four second test add between 0.17% and 0.25% to the running time.

Our work was not the first attempt to implement the ARM FCSE for Linux. As part of his student work at the University of New South Wales, Adam Wiggins implemented the FCSE for Linux kernel 2.4. The changes in Wiggins' Linux patch [3] are considerably more extensive than just adding the FCSE. To avoid changing the page directories during a context switch, Wiggins employed the ARMv5 domain mechanism to create a single, global Translation Lookaside Buffer (TLB) in software. This approach can avoid invalidating the TLB during a context switch, in addition to avoiding cache invalidation. However, the performance gain comes at the cost of some complexity in the ARM Linux kernel code. Wiggins termed the combination of using ARM domains and FCSE as Fast Address Space Switching (FASS), and he had previously [4] implemented FASS for the L4 microkernel.

Van Schaik and Heiser [5] describe a paravirtualized Linux 2.6 kernel called "Wombat" running under the L4 microkernel. They report Wombat having greatly reduced context switching times compared with native ARM Linux, under the restriction of a 32 MB virtual memory space for user programs.

Sebastian Smolorz [6] attempted a port of Wiggins' FASS work to Linux kernel 2.6. Smolorz's work helps to highlight the ARMv5 context switch performance problems, and it presents one possible approach toward a solution. Although we did copy one useful header file from this source, in our view this port suffers from two major drawbacks. First, the FASS changes are far too intrusive to ever be accepted into the main line kernel, since they make fairly complex use of ARM domains, introducing a completely new concept into the ARM Linux memory system. Second, the port is not fully working and is no longer being developed.

3 The ARM FCSE

A simplified schematic of the ARMv5 Memory Management Unit (MMU) is depicted in Figure 1. The CPU issues a virtual address (marked VA in the figure) to the instruction cache (I-cache), data cache (D-cache), and the TLB. The TLB is a cache of virtual to physical address mappings. If the translation is not cached, then a hardware page table walker obtains the physical address (marked PA in the figure).

The cache unit returns the desired data to the CPU if the requested address is in the cache.

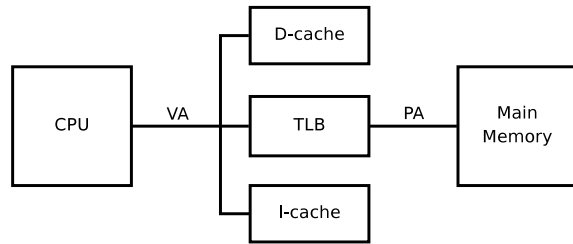


FIGURE 1: *Memory Management Unit*

Not shown in the figure is the fact that the cache units consult permission bits stored within the TLB. Since Linux uses the same address space layout for every user program, this arrangement of the MMU requires invalidating the caches and the TLB during every context switch.

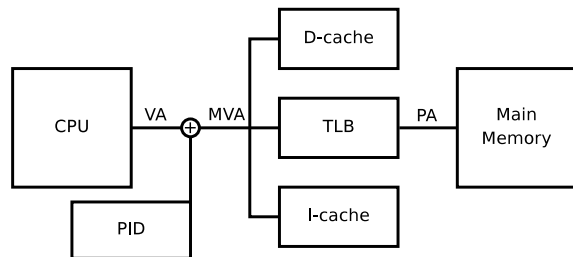


FIGURE 2: *FCSE PID Relocation*

The FCSE provides a "Process ID" register (PID) for sharing a single 32 bit virtual address space. This register contains a 7 bit process identifier that can be combined with a 25 bit virtual address, as shown in Figure 2. When the PID is set to a non-zero value and the 7 high order bits of a virtual address are cleared, a modified virtual address (shown as MVA) is generated by adding the PID value to the virtual address according to the following formula:

$$MVA = VA + (PID \ll 25)$$

Using this method allows 128 distinct 32 MB address spaces, so that up to 128 processes may safely coexist without having to flush the caches at each context switch. However, we note that it is still necessary to invalidate the TLB during a context switch in order to enforce memory protection. The resulting limitation of 128 processes and 32 MB address space is acceptable for some embedded systems.

4 Implementation

The Linux memory management subsystem¹ is not neatly abstracted into a few, water tight source files.

¹Mel Gorman's book [7] offers a good, if somewhat dated, introduction to the Linux memory management system.

On the contrary, the memory management code is spread among many files, some of which are architecture independent, and some architecture specific. Moreover, the actual logic can be quite tricky and contains a fair amount of “black magic.”

We endeavored to implement the FCSE in a minimally intrusive way, without requiring major changes in the existing memory management code. The current patch [8] changes 23 files and affects about 300 lines in the kernel. In addition, the patch adds two new, FCSE-specific files with about 260 lines total. Our implementation affects only the ARM specific architecture sources. No changes in the main Linux memory management subsystem were necessary.

4.1 Guaranteed Mode

The virtual memory layout for ARM Linux is fixed in such a way that all virtual addresses above `TASK_SIZE` (3 GB) are always interpreted as kernel addresses. Adapting the FCSE to this model thus required reducing the number of useable PIDs to 95. When the FCSE is active, it effectively partitions a single virtual memory space into one kernel and 95 user regions, as shown in Table 1.

Virtual Address	Use
⋮	
0xC000.0000	KERNEL
0xBE00.0000	reserved
0xBC00.0000	PID 94
⋮	
0x0400.0000	PID 2
0x0200.0000	PID 1
0x0000.0000	PID 0

TABLE 1: *The FCSE memory layout*

Although the original intent of the ARMv5 PID register appears to have been to allow the operating system to keep the current process ID in this register, our implementation decouples the ARM PID from the Linux process ID. Linux gives each thread its own process ID, but all the threads within one process share the same virtual memory space. Therefore, we associate one ARM PID with the `mm_struct` of each process.

With FCSE enabled, the CPU automatically converts a virtual address in the range from zero to `0x20000000` into a modified virtual address within one of the 95 possible slots. This creates situations where the kernel expects an unmodified VA, but the hardware MMU expects a MVA. The patch provides a

pair of functions² to convert between a VA and a MVA. A good deal of the effort in implementing the FCSE was simply to identify the spots where such a conversion is required.

The only place where a MVA to VA conversion occurs is in the fault handler. In this case, the CPU has issued an instruction that has triggered a fault. The hardware gives the MVA as the source of the fault, but the kernel requires the original VA in order to properly handle the fault.

In contrast, the VA to MVA conversion occurs several times, and for two different reasons. When flushing the cache and the TLB, the kernel has a VA in its Virtual Memory Area (VMA) data structures, but the cache and TLB contain the MVA that the CPU generated. Thus the cache and TLB flushing code must present the MVA to the hardware. These cases account for almost all of the VA to MVA conversions.

The one other important VA to MVA conversion occurs in the `pgd_offset()` macro. This macro is used by kernel code that needs to access the page tables. By modifying the VA, this macro now ensures that page tables will be created or adjusted using a correctly relocated MVA.

Besides accounting for the VA to MVA conversions, the FCSE code must also enforce the 32 MB limit on the virtual address space and reprogram the PID register during a context switch. Limiting the address space is accomplished in the `arch_get_unmapped_area` function, where any mapping that would cause a task’s space to exceed 32 MB is denied. Reprogramming the PID register amounts to a one line addition to the `switch_mm` function. In order to actually benefit from the FCSE, we also must disable the cache invalidation code normally executed during a context switch. The current patch implements this for Xscale, ARM920, and ARM926, by adding appropriate `#ifdefs` to the machine specific assembler files under `arch/arm/mm`.

4.2 Best Effort Mode

After presenting initial versions of the FCSE patch to the ARM Linux kernel mailing list, one repeated criticism was that allowing only 95 processes, each with a 32 MB virtual address space, is an unacceptable limitation. There was a desire for a way to enable the FCSE but, at the same time, to fall back to the standard, unlimited memory management scheme dynamically. To fulfill this requirement, the current version of the patch offers a compilation option to enable either a “guaranteed” mode, or a “best effort” mode. The guaranteed mode operates as de-

²The functions are defined in such a way that, if the FCSE option is not selected at compile time, they do nothing at all.

scribed above, with the mentioned limitations, and never flushes the caches during a context switch.

When the best effort mode is enabled, there are no artificial limits to the number of processes or the virtual address space. If the run time condition of the user space programs permits, then the system will benefit from the performance advantage that the FCSE offers. Under the best effort mode, the cache flush during a context switch is only performed when necessary to ensure the integrity of the virtual memory system.

For the best effort mode, we overcome the 95 processes limitation by reusing the FCSE PIDs. Each Linux process is assigned to one of 95 groups, and each group shares one PID. A context switch between processes belonging to two different groups does not necessarily trigger a cache flush. Instead, each group has a “dirty” bit which is set whenever a process in that group is scheduled. Also, each group remembers which of its processes was last scheduled. A cache flush is needed when the new process’s group is “dirty” and a different process was last scheduled in that group.

In order to overcome the 32 MB virtual memory limitation, the best effort mode reserves the zero PID for those processes which require a larger space. Once a process requests more that 32 MB, it is relocated from its initial PID to PID zero by coping its page tables. Flushing the cache is required every time such a process is scheduled or preempted.

5 Performance Evaluation

In order to measure the effectiveness of the FCSE, we benchmarked a recent Linux kernel both with and without the FCSE. This section presents results of performance measurements on artificial and real world applications. These measurements were performed on an Intel IXP425 clocked at 400 MHz, with 128 MB main memory. The IXP425 has separate data and instruction caches, each 32 KB in size.

5.1 Artificial Applications

5.1.1 `cyclicttest`

Thomas Gleixner’s `cyclicttest` program [9] was developed to measure the scheduling latency of the Linux kernel. The program repeatedly attempts to sleep (yield the CPU) for a certain time interval, and measures the actual duration of the sleep to infer the latency. We ran `cyclicttest` first under plain Linux, and then under FCSE Linux in both guaranteed and best effort mode, with the following command line arguments.

```
cyclicttest -D 60 -p 50 -q -h 1000
```

In order to create a load on the system, we ran the following program in the background before running the `cyclicttest` program.

```
while [ 1 ]; do
    find / ;
    cat /dev/mem > /dev/null ;
done
```

Table 2 shows the measured scheduling latencies for each of the three test cases. On average, the guaranteed FCSE mode had 100 μ s less latency than Linux without the FCSE, while the best effort mode showed about 50 μ s better average performance than plain Linux. The worst measured latency exceeded 550 μ s for all three test cases, but the FCSE test cases outperformed plain Linux here as well.

Kernel	Min	Ave	Max
Plain Linux	97	194	682
Guaranteed FCSE	26	94	611
Best Effort FCSE	37	143	569

TABLE 2: *Scheduling Latency (μ s)*

Figures 3 and 4 show the histograms of the `cyclicttest` latency measurements. In Figure 3 the results from the guaranteed FCSE mode, on the left hand side, are presented along with plain Linux, on the right hand side.

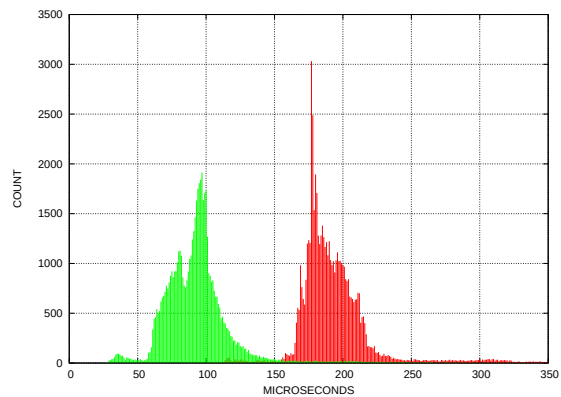
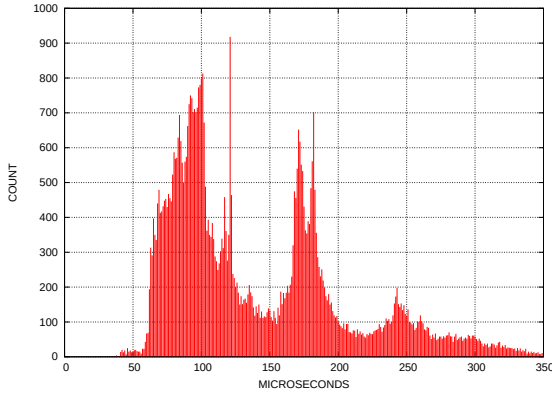


FIGURE 3: *FCSE versus Plain Linux*

Figure 4 shows the results from the best effort FCSE mode. Here one can recognize that the best effort mode appears as a combination of plain Linux and FCSE Linux, as it were. It is interesting to notice an additional latency region at about 250 μ s. Although the average performance is clearly improved over plain Linux, the best effort mode appears to have increased latencies in some code paths.


 FIGURE 4: *Best Effort FCSE*

5.1.2 lmbench

Larry McVoy’s `lmbench` [10] comprises a suite of programs that measure various system performance benchmarks. In our view, `lmbench` is not as useful as other tools for measuring latency since it reports only an average value, rather than a set of statistics or a histogram. However, since both Choi [1] and Van Schaik [5] report results obtained using `lmbench`, we also include measurements made with that program.

Since neither of those two papers mention using any kind of system load, we assume that their `lmbench` measurements were conducted with an otherwise idle system. Thus, in contrast to the `cyclictst` measurements, we ran the tests without any system load, with the command line as follows.

```
for s in 0 1 4 16; do
    lat_ctx -s $s 2 4 6 8 10 12 14 16;
done
```

We compared `lmbench` performance for plain Linux and FCSE Linux in both guaranteed and best effort mode. For each variety of Linux, we ran the `lat_ctx` test four times, using a different work buffer size each time, and varying the number of processes participating in the test. Following Choi, [1] we used work buffer sizes of 0, 1, 4, and 16 KB. A larger work buffer causes greater pressure on the data cache. For each buffer size, the test connects a number of processes in a ring of Unix pipes. Each process reads a token from its pipe and then writes the token to the next process. In the graphs, the x-axis shows the number of processes, the y-axis shows the measured latency. The result for each work buffer size appears as one line.

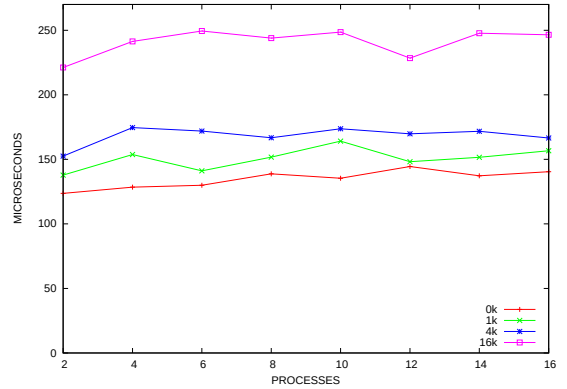

 FIGURE 5: *Plain Linux - lmbench*

Figure 5 shows the results of the `lmbench` measurement for plain Linux. For this test case, the scheduling latency stayed well above 100 μ s in every case. The fact that the number of processes made little difference in the latency comes as no surprise, since, by flushing the cache during a context switch, plain Linux always provokes the worst case performance.

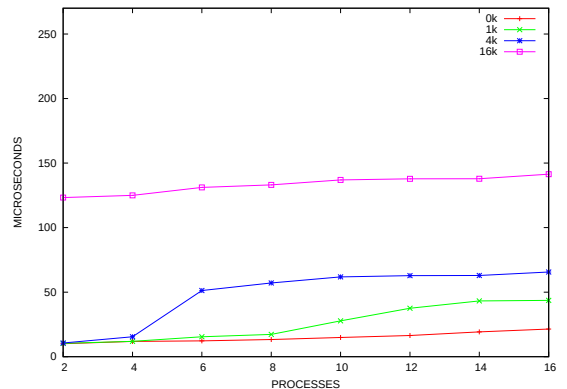
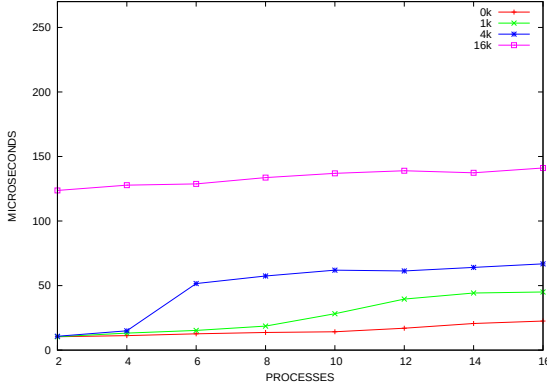

 FIGURE 6: *FCSE Linux - lmbench*

Figure 6 shows the results for FCSE Linux in guaranteed mode, and Figure 7 shows the best effort mode. For these test cases, the scheduling latency never exceeded 150 μ s. Here we notice that increasing the number of processes did indeed affect the latency. It is also interesting that the two FCSE modes yielded almost identical performance for this test. We expect that running the test under a loaded system, as we did for the `cyclictst`, would better highlight the difference between the two modes.


 FIGURE 7: *Best Effort FCSE - lmbench*

5.2 Real World Applications

Our original motivation for implementing the FCSE was to improve the performance of ARMv5 machines on real world applications. One recent trend in various industries is to run time critical applications over standard Ethernet. For example, Ethernet networks are being used as field bus replacements (like EtherCAT), for distributed real time computing, and for audio/video streaming. We present performance results for two real world applications that send or receive time critical messages in the form of Ethernet packets.

We used the Beckhoff ET200 Industrial Ethernet Multichannel Probe in order to obtain impartial results, external to the device under test. The ET2000 is able to time stamp Ethernet packets with a resolution of 10 nanoseconds. Since we measure the interval with an Ethernet tap outside of the device under test, the measured time includes all of the delays from both the hardware and software of the system under test.

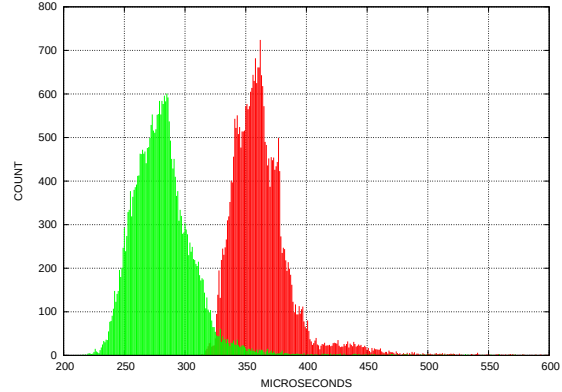
5.2.1 Packet Message Reply

In this application, the program expects incoming Ethernet packets containing control commands. After receiving a command packet, the application updates its internal state and sends a status reply packet. It is important that the program sends the reply as soon as possible. We measured the interval from the time the command packet appears on the wire to the time the reply packet appears.

Kernel	Min	Ave	Max	Std Dev
Plain Linux	313	364	771	29.46
Guaranteed FCSE	211	283	840	30.50
Best Effort FCSE	210	284	718	31.59

 TABLE 3: *Packet Reply Latency (μ s)*

We tested this application under plain Linux and FCSE Linux in both guaranteed and best effort mode. During these tests we created a system load using the method described in Section 5.1.1. For this test, the best effort FCSE mode performed nearly as well as the guaranteed mode, as shown in Table 3. Both of the FCSE modes outperformed plain Linux by 80 μ s on average. Figure 8 presents a histogram of the reply times using guaranteed FCSE mode, on the left hand side, along with the times using plain Linux, on the right hand side.


 FIGURE 8: *FCSE vs Plain Linux*

5.2.2 Periodic Packet Transmission

This application periodically transmits a single Ethernet packet containing sampled data. The program attempts to send one packet every 500 μ s exactly. On the test platform, keeping such an extremely short interval is not possible using a standard Linux kernel, so we used Xenomai [11] to achieve better real time performance. The program under test creates one high priority Xenomai thread which runs a periodic timer set at 500 μ s. In order to demonstrate the effect of the FCSE, we also modified the `trivial-periodic` example program to run its timer in a one millisecond period. This second Xenomai thread was executed with a lower priority than the program under test.

Xenomai Kernel	Min	Ave	Max	Std Dev
Plain Linux	458	500	542	17.157
Guaranteed FCSE	456	500	534	1.969

 TABLE 4: *Packet Period (μ s)*

Xenomai Linux is able to deliver the correct average period both with and without the FCSE, as the results in Table 4 show. However, the standard deviation is noticeably higher without the FCSE.

Figure 9 shows the histogram of the packet transmission period when using plain Xenomai. Notice that the y-axis is logarithmically scaled in order to

highlight the worst case performance. The symmetrical form of this graph demonstrates how the Xenomai scheduler is able to keep a periodic timer on track. Whenever the timer overshoots a deadline, the induced delay is always compensated in the next period. This graph has two maxima around the average interval, offset about $15\ \mu\text{s}$. In addition, there are two local maxima offset about $40\ \mu\text{s}$ from the center.

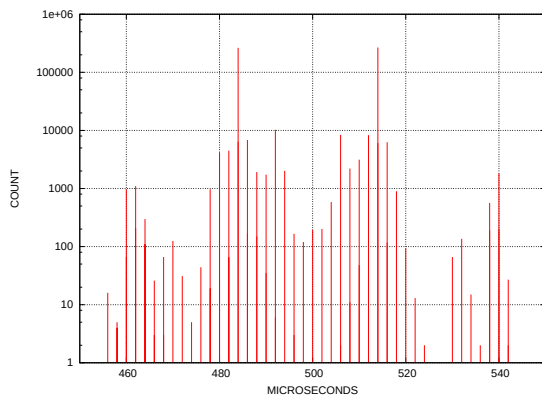


FIGURE 9: *Periodic Task - Plain*

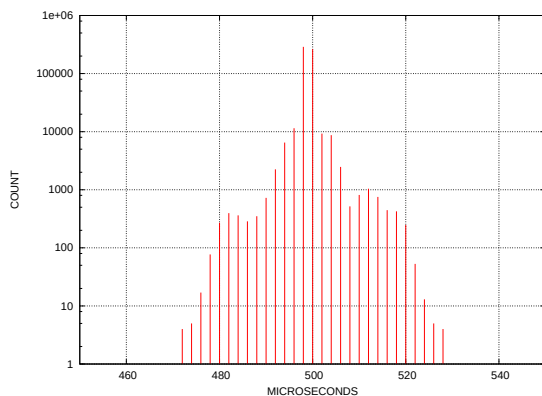


FIGURE 10: *Periodic Task - FCSE*

Figure 10 shows the histogram when the FCSE is enabled. In contrast to the previous graph, here the histogram has a wedge shape with only one maximum in the center. Comparing the shapes of Figures 9 and 10 illustrates the consequence of the heavy handed cache flushes which are necessary when the FCSE is not enabled.

6 Conclusions and Future Work

The ARM FCSE offers a method for avoiding costly cache flushing during context switches. We presented a minimally intrusive implementation of the FCSE for the Linux kernel. Experimental evidence from artificial and real world tests shows that enabling the FCSE can reduce the time to schedule processes by dozens or even hundreds of microseconds. This performance gain is significant enough to bring certain real time applications within reach of Linux running on ARMv5 processors.

The FCSE imposes fairly restrictive limitations on programs. However, for systems which cannot accept the FCSE limitations, the benefits of the FCSE can still be exploited by using our patch in the dynamic, best effort mode.

It is the authors' wish to have the FCSE merged into the mainstream ARM Linux kernel. The patch has previously been presented on the mailing list, and it will be resubmitted in small, more easily digestible pieces in the near future. Some further work needs to be done, namely identifying the other ARMv5 machines supported by Linux and adapting their machine specific assembler files to benefit from the FCSE. One interesting possibility not covered here would be to combine the FCSE with the PREEMPT_RT patch and gauge its effectiveness in that configuration.

References

- [1] Hyok-Sung Choi and Hee-Chul Yun.
Context switching and IPC performance comparison between uClinux and Linux on the ARM9 based processor.
Technical report, Samsung Electronics, 2004.
http://opnsrc.sec.samsung.com/document/uc-linux-04_sait.pdf.
- [2] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell.
Context switch overheads for Linux on ARM platforms.
ExpCS, 13-14, June 2007.
- [3] Adam Wiggins.
Fast address-space switching for ARM Linux kernels.
<http://www.ertos.nicta.com.au/software/fass>, March 2004.

- [4] Adam Wiggins.
The design and implementation of the L4 microkernel on the StrongARM SA-1100.
Bachelor thesis, University of New South Wales, November 1999.
- [5] Carl van Schaik and Gernot Heiser.
High-performance microkernels and virtualisation on ARM and segmented architectures.
Technical report, University of New South Wales, 2007.
<http://www.ertos.nicta.com.au/software/fass>.
- [6] Sebastian Smolorz.
linux-2.6 FASS.
<http://git.opensource.emlix.com/cgi-bin/gitweb.cgi?p=kernel/fass/linux-2.6.git>, November 2007.
- [7] Mel Gorman.
Understanding the Linux Virtual Memory Manager.
Prentice Hall, 2004.
<http://www.kernel.org/doc/gorman>.
- [8] Gilles Chanteperdrix.
FCSE for Linux.
<http://git.denx.de/?p=ipipe-2.6.git;a=commitdiff;h=607cdaaa4447068ac81e04ad79af3feb04858fd5>,
May 2009.
- [9] Thomas Gleixner.
Cyclctest.
<http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests>, August 2009.
- [10] Larry McVoy.
Lmbench - tools for performance analysis.
<http://www.bitmover.com/lmbench>, August 2009.
- [11] Philippe Gerum.
Xenomai: Real-time framework for Linux.
<http://www.xenomai.org>, August 2009.

XtratuM: a Hypervisor for Safety Critical Embedded Systems

M. Masmano, I. Ripoll, and A. Crespo

Instituto de Informática Industrial, Universidad Politécnica de Valencia (Spain)

{mmasmano, iripoll, alfons}@ai2.upv.es

J.J. Metge

CNES (Toulouse, France)

jean-jacques.Metge@cnes.fr

Abstract

XtratuM is an hypervisor designed to meet safety critical requirements. Initially designed for x86 architectures (version 2.0), it has been strongly redesigned for SPARC v8 architecture and specially for the LEON2 processor. Current version 2.2, includes all the functionalities required to build safety critical systems based on ARINC 653, AUTOSTAR and other standards. Although XtratuM does not provides a compliant API with these standards, partitions can offer easily the appropriated API to the applications. XtratuM is being used by the aerospace sector to build software building blocks of future generic on board software dedicated to payloads management units in aerospace.

XtratuM provides ARINC 653 scheduling policy, partition management, inter-partition communications, health monitoring, logbooks, traces, and other services to easily been adapted to the ARINC standard. The configuration of the system is specified in a configuration file (XML format) and it is compiled to achieve a static configuration of the final container (XtratuM and the partition's code) to be deployed to the hardware board. As far as we know, XtratuM is the first hypervisor for the SPARC v8 architecture.

In this paper, the main design aspects are discussed and the internal architecture described. An evaluation of the most significant metrics is also provided. This evaluation permits to affirm that the overhead of a hypervisor is lower than 3% if the slot duration is higher than 1 millisecond.

1 Introduction

Although virtualisation has been used in mainframe systems since 60's; the advances in the processing power of the desktop processors in the middle of the 90's, opened the possibility to use it in the PC market. The embedded market is now ready to take advantage of this promising technology. Most of the recent advances on virtualization have been done in the desktop systems, and transferring these results to embedded systems is not as direct as it may seem.

The current state of the visualizing technology is the result of a convergence of several technologies: operating system design, compilers, interpreters, hardware support, etc. This heterogeneous origin, jointly with the fast evolution, has caused a confusion on the terminology. The same term is used to refer to different ideas and the same concept is differently named depending on the engineer background.

A virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. **Hypervisor** (also known as virtual machine monitor VMM [7]) is a layer of software (or a combination of software/hardware) that allows to run several independent execution environments¹ in a single computer. The key difference between hypervisor technology and other kind of virtualizations (such as java virtual machine or software emulation) is the performance. Hypervisor solutions have to introduce a very low overhead; the throughput of the virtual machines has to be very close to that of the native hardware.

Hypervisor is a new and promising technology, but has to be adapted and customized to the requirements of the target application. As far as we know, there are no previous experiences with hypervisors for spatial systems.

¹We will use the terms: guest, virtual machine and partition as synonyms.

When a hypervisor is designed for real-time embedded systems, the main issues that have to be considered are:

- Temporal and spatial isolation.
- Basic resource virtualisation: clock and timers, interrupts, memory, cpu time, serial i/o.
- Real-time scheduling policy for partition scheduling.
- Efficient context switch for partitions.
- Deterministic hypervisor system calls.
- Efficient inter-partition communication.
- Low overhead.
- Low footprint.

In this paper, we present the design, implementation and evaluation of XtratuM for the LEON2 processor. Although XtratuM was initially implemented for x86 architectures, its porting to LEON2 has implied a strong effort in redesign and implementation due to the architecture constraints.

2 Virtualising technologies overview

Attending to the resources used by the hypervisor there are two classes of hypervisors called type 1 and type 2. The type 1 hypervisors run directly on the native hardware (also named *native* or *bare-metal* hypervisors); the second type of hypervisors are executed on top of an operating system. The native operating system is called host operating system and the operating systems that are executed in the virtual environment are called guest operating systems.

Although the basic idea of *virtualizing*[5] is widely understood: “any way to recreate an execution environment, which is not the original (native) one”; there are substantial differences between the different technological approaches used to achieve this goal.

Virtualizing is a very active area, several competing technologies are actively developed. There is still not a clear solution, or a winner technology over the rest. Some virtualizing technologies are better than other for a given target. For example, on desktop systems, para-virtualization is the best choice if the source code of the virtualized environment is available, otherwise full-virtualization is the only possible solution.

A detailed description and analysis of the techniques and the existing solutions is beyond the scope of this report (the reader is referred to the document “Virtualization: State of the Art” [14]). Just to summarise the current available solutions for the real-time embedded systems:

Separation kernel: Also known as operating system-level virtualization. In this approach the operating system is extended (or improved) to enforce a stronger isolation between processes or groups of processes. Each group of isolated group of processes is considered a partition. In this solution, all the partitions must use the same operating system. It is like if several instances of the same O.S. were executed in the same hardware.

Micro-kernel: This was originally an architectonic solution for developing large and complex operating systems. The idea was to separate the core kernel services from the rest of more complex and “baroque” services. The core kernel services are implemented by a layer of code called *micro-kernel*, and consist of: context switch, basic memory management, and simple communication services (IPC). Although the microkernel technology was developed as a paradigm to implement a single operating system, the services provided by the micro-kernel can be used to build several different operating systems, resulting in a virtualized system.

The micro-kernel approach started with the March micro-kernel [8]. The most representative implementation of a micro-kernel is the L4 [10, 9].

Bare-metal hypervisor: It is a thin layer of software that virtualizes the critical hardware devices to create several isolated partitions. The hypervisor also provides other virtual services: inter-partition communication or partition control services.

The hypervisor does not define an abstract virtual machine but tries to reuse and adapt to the underlying hardware as much as possible to reduce the virtualization overhead. In other words, the virtual machine will be close to the native hardware in order to directly use the native hardware as much as possible without jeopardizing the temporal and spatial isolation.

2.1 Para-virtualization

The para-virtualization (term coined in the Xen [6] project) technique consist in replacing the conflicting instructions² explicitly by functions provided by the hypervisor. In this case, the partition code has to be aware of the limitations of the virtual environment and use the hypervisor services. Those services are provided through a set of *hypercalls*.

The hypervisor is still in charge of managing the hardware resources of the systems, and enforce the spatial and temporal isolation of the guests. Direct access to the native hardware is not allowed.

The para-virtualization is the technique that better fits the requirements of embedded systems: Faster, simpler, smaller and the customization (para-virtualization) of the guest operating system is not a problem because the source code is available. Also, this technique does not requires special processor features that may increase the cost of the product.

2.2 Dedicated devices

In the server and desktop segments, the virtualizer provides a complete (or full) virtualized environment for each virtual machine. That is, the each virtual machine is fully isolated from the native hardware, and has no direct access to any peripheral.

Some virtualizers allows a virtual machine to access directly some parts of the native hardware. This concept is known as partial virtualization. This technique is widely used in embedded systems when a device is not shared among several partitions, or when the complexity of the driver is that high that it does not worth including it in the virtualizer layer. In some cases, when the policy for sharing a peripheral is user specific (or when the user requires a fine grain control over the peripheral), it is better to allocate the native peripheral to a designated manager virtual machine.

In this case, the virtualizer is not aware of the exact operation of the peripheral, but enforces that only the allowed partition uses it. This technique is frequently used in embedded systems due to the use of home designed (or customised) peripherals.

3 XtratuM Overview

XtratuM 1.0 [11, 12] was designed initially as a substitution of the RTLinux [15, 4] to achieve temporal and spatial requirements. XtratuM was designed as a nanokernel which virtualises the essential hardware devices to execute concurrently several OSes, being at least one of these OSes a RTOS. The other hardware devices (including booting) were left to a special domain, named root domain. The use of this approach let us speed up the implementation of a first working prototype. Like RTLinux before, XtratuM was implemented as a LKM which, once loaded, takes over the box; Linux was executed as the root domain. XtratuM and the root domain still shared the same memory space. Nevertheless, the rest of the domains were run in different memory maps, providing *partial* space isolation.

After this experience, it was redesigned to be independent of Linux and bootable. The result of this is XtratuM 2.2 [13]. In the rest of this paper, the term XtratuM will refer to this second version. This version is being used to build a TSP-based solution for payload on-board software, highly generic and reusable, project named LVCUGEN [2]. TSP (Time and Space Partitioning) based architecture has been identified as the best solution to ease and secure reuse, enabling a strong decoupling of the generic features to be developed, validated and maintained in mission specific data processing [3].

XtratuM is a type 1 hypervisor that uses para-virtualization. The para-virtualized operations are as close to the hardware as possible. Therefore, porting an operating system that already works on the native system is a simple task: replace some parts of the operating system HAL (Hardware Abstraction Layer) with the corresponding hypercalls.

The ARINC-653 [1] standard specifies the baseline operating environment for application software used within Integrated Modular Avionics (IMA), based on a partitioned architecture. Although not explicitly stated in the standard, it was developed considering that the underlying technology used to implement the partitions is the separation kernel. Although it is not an hypervisor standard, some parts of the APEX model of ARINC-653 are very close to the functionality provided by an hypervisor. For this reason, it was used as

²Conflicting instructions: instructions that operate directly on the native hardware and may break the isolation.

a reference in the design of XtratuM. It is not our intention to convert XtratuM in an ARINC-653 compliant system.

It defines both, the API and operation of the partitions, and also how the threads or processes are managed inside each partition.

In an hypervisor, and in particular in XtratuM, a partition is a *virtual computer* rather than a group of strongly isolated processes. When multi-threading (or tasking) support is needed in a partition, then an operating system or a run-time support library has to provide support to the application threads. In fact, it is possible to run a different operating system on each XtratuM partition.

XtratuM was designed to meet safety critical real-time requirements. The most relevant features are:

- Bare hypervisor.
- Employs para-virtualisation techniques.
- An hypervisor designed for embedded systems: some devices can be directly managed by a designated partition.
- Strong temporal isolation: fixed cyclic scheduler.
- Strong spatial isolation: all partitions are executed in processor user mode, and do not share memory.
- Fine grain hardware resource allocation via a configuration file.
- Robust communication mechanisms (XtratuM sampling and queuing ports).

4 Architecture and design

Figure 1 shows the complete system architecture.

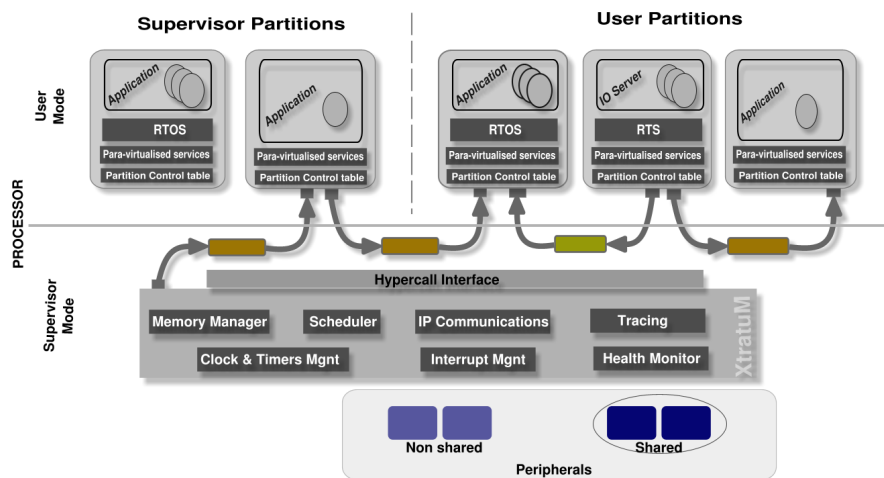


Figure 1: System architecture.

The main components of this architecture are:

Hypervisor XtratuM is in charge of virtualisation services to partitions. It is executed in supervisor processor mode and virtualises the cpu, memory, interrupts and some specific peripherals. The internal XtratuM architecture includes: memory management, scheduling (fixed cyclic scheduling), interrupt management, clock and timers management, partition communication management (ARINC-653 communication model) and health monitoring. A more detailed explanation of each component will be detailed in the full paper.

Partitions A partition is an execution environment managed by the hypervisor which uses the virtualised services. Each partition consists of one or more concurrent processes (implemented by the operating system of each partition), sharing access to processor resources based upon the requirements of the application. The partition code can be:

- An application compiled to be executed on a bare-machine (bare-application).

A real-time operating system (or runtime support) and its applications.

- A general purpose operating system and its applications.

Partitions need to be *virtualised* to be executed on top of an hypervisor. Depending on the type of execution environment, the virtualisation implications in each case can be summarised as:

Bare application The application has to be virtualised using the services provided by XtratuM. The application is designed to run directly on the hardware and it has to be aware about it.

Operating system application When the application runs on top of a (real-time) operating system, it uses the services provided by the operating system and does not need to be virtualised. But the operating system has to deal with the virtualisation. The operating system has to be virtualised (ported on top of XtratuM).

Two different type of partitions can be defined: supervisor () and user ().

Partitions can send/receive messages to/from other partitions. The basic mechanisms provided are sampling a queuing ports as defined in ARINC-653.

4.1 Design issues

XtratuM has designed specifically to meet real-time constraints and be as efficient as possible. The main decisions involving these requirements are:

XtratuM has been designed specifically to meet real-time constraints and to be as efficient as possible. The main decisions involving these requirements are:

Data structures are static: All data structures are pre-defined at build time from the configuration file; therefore: 1) more efficient algorithms can be used; 2) the exact resources used by XtratuM are known.

XtratuM code is non-preemptive: ³ Although this feature is desirable in most operating systems, there is no benefits in the case of a small hypervisor. The code is simpler (no fine grain critical sections) and faster.

All services (hypercalls) are deterministic and fast: XtratuM provides the minimal services to guarantee the temporal and spatial isolation of partitions.

Peripherals are managed by partitions. XtratuM only supervises the access to IO ports as defined in the configuration file.

Interrupt occurrence isolation: When a partition is under execution, only the interrupts managed by this partition are enabled, which minimizes inter-partition interferences through hardware.

Full control of the resources used by XtratuM and the partitions. All the system resources allocated to partitions are specified in a configuration file.

One-shot timer: It provides a 1 microsecond resolution both for timers and clocks with a very low overhead.

4.2 LEON2 virtualisation issues

The SPARC v8 architecture does not provide any kind of virtualization support. It implements the classical two privilege levels: supervisor and user; used by the operating system to control user applications. In order to guarantee the isolation, partition code has to be executed in user mode; only XtratuM can be executed in supervisor mode.

Additionally, the design of XtratuM for LEON2 processors introduce some additional aspects as the register window mechanism and the MPU (LEON2 without MMU).

³Note that XtratuM is **non-preemptive**, but it is prepared to be **re-entrant**, which allows multiple processors to execute concurrently XtratuM code.

5 System configuration and deployment

The integrator, jointly with the partition developers, have to define the resources allocated to each partition. The configuration file that contains all the information allocated to each partition as well as specific XtratuM parameters is called `XM_CF.xml`. It contains the information as: memory requirements, processor sharing, peripherals, health monitoring actions, etc.

Memory requirements: The amount of physical memory available in the board and the memory allocated to each partition.

Processor sharing: How the processor is allocated to each partition: the scheduling plan.

Native peripherals: Those peripherals not managed by XtratuM can be used by one partition. The I/O port ranges and the interrupt line if any.

Health monitoring: How the detected error are managed: direct action, delivered to the offending partition, create a log entry, etc.

Inter-partition communication: The ports that each partition can use and the channels that link the source and destination ports.

Since `XM_CF.xml` defines the resources allocated to each partition, this file represents a **contract between the integrator and the partition developers**. A partner (the integrator or any of the partition developers) should not change the contents of the configuration file on its own. All the partners should be aware of the changes and should agree in the new configuration in order to avoid problems later during the integration phase.

In order to reduce the complexity of the XtratuM hypervisor, the `XM_CF.xml` is parsed and translated into a binary representation that can be directly used by XtratuM code. This process is performed by two tools `xmcparser` and `xmcbuilder`.

In order to ensure that each partition does not depend on or affect other partitions or the hypervisor due to shared symbols. The partition binary is not an ELF file. It is a raw binary file which contains the machine code and the initialized data.

The system image is a **single file** which contains all the code, data and configuration information that will be loaded in the target board. The tool `xmpack` is a program that reads all the executable images and the configuration files and produces the system image. The final system image also contains the necessary code to boot the system. The system image file can be written into the ROM of the board.

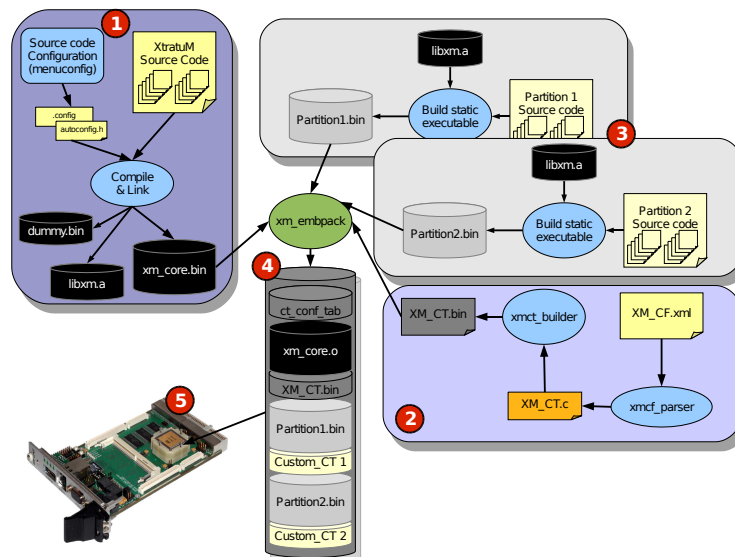


Figure 2: The big picture of building a XtratuM system.

6 Performance Evaluation

In this section we provide the initial evaluation of XtratuM for LEON2 processor. A development board (GR-CPCI-AT697 LEON2-FT 80MHz with 8Mb flash PROM and 4 Mb RAM, 33 MHz 32-bit PCI bus) has been used during the evaluation.

The following metrics have been measured:

- Partition context switch: Time needed by the hypervisor to switch between partitions. Three main activities can be identified:
 1. Save the context
 2. Timer interrupt management
 3. Scheduling decision
 4. Load the context of the new partition
- Clock interrupt management: Time needed to process a clock interrupt
- Effective slot duration: Time where the partition is executing partition code during a slot.
- Partition performance loss: This measurement provides a measure of the overhead introduced by XtratuM. It is measured at partition level.
- Hypercall cost: Time spent in some of the hypercalls

6.1 Partition context switch

The measures involving internal activity of XtratuM have been measured adding breakpoints at the beginning and end of the code to be measured. Next table presents the activities involved in a partition context switch and the cost measures (in microseconds).

Internal operation	Time (microseconds)
Partition Context switch	27
1. Save the context	5
2. Timer interrupt management	9
3. Scheduler	8
4. Load the context	5
Effective slot duration	Slot duration - 27

Table 1: Partition context switch measurement.

6.2 Performance evaluation

In order to evaluate the partition performance loss a scenario consisting in 3 partitions with the same code which increments a counter. They write in a sampling port the address of the counter. A forth partition reads the ports and prints the counter value of the each partition.

In this scenario, several plans are built:

- Case 1** Partitions 1,2 and 3 are executed sequentially with a slot duration of 1 second. The scheduling plan consists in the execution of p1; p2; p3; p4. When p4 is executed, it reads the final counter of the partitions executed during 1 second. This case is taken as reference.
- Case 2** Partitions 1,2 and 3 are executed sequentially with a slot duration of 0.1 second. The scheduling plan consists in the execution of 10 sequences of p1; p2; p3;... (10 times)....; p4. When p4 is executed, it reads the final counter of the partitions executed 10 times (1 second).
- Case 3** Partitions 1,2 and 3 are executed sequentially with a slot duration of 0.2 second. The scheduling plan consists in the execution of 5 sequences of p1; p2; p3;... (5 times)....; p4. When p4 is executed, it reads the final counter of the partitions executed 5 times (1 second).
- Case 4** Partitions 1,2 and 3 are executed sequentially with a slot duration of 0.2 second. The scheduling plan consists in the execution of 5 sequences of p1; gap; p2; gap; p3;gap ... (5 times)....; p4. When p4 is executed, it reads the final counter of the partitions executed 5 times (1 second).

	case 1	case 2	case 3
Average	11428202	11426656	11424764
Difference	0	1546	3438
Performance lost	0	0,01%	0,03%

Table 2: Case 1,2 and 3 measurements

Next table compares the results achieved by the three first cases. The values shown are the counter achieved each major frame.

Next table compares the results of cases 2 and 4. The difference is a gap introduced each time a slot is planned. As it can see in the table the difference is not significative.

	case 2	case 4
Average	11426656	11426715
Difference	59	0
Loss of performance	0,0001%	0,00

Table 3: Case 2 and 4 measurements

6.3 Performance lost due to the number of partitions.

In order to evaluate the effect of the number of partitions, two cases has been defined.

Case 5 3 Partitions are executed sequentially with a slot duration of 100 miliseconds. The scheduling plan consists in the execution of p1; p2; p3; p4. When p4 is executed, it reads the final counter of the partitions executed during 1 second.

Case 6 8 Partitions are executed sequentially with a slot duration of 100 miliseconds. The scheduling plan consists in the execution of p1; p2; p3; p4. When p4 is executed, it reads the final counter of the partitions executed during 1 second.

Next tables show the results in each case and a comparison of both cases.

Case 5	Partition 1	Partition 2	Partition 3
Avg	1142484	1142484	1142474
Max	1142500	1142488	1142476
Min	1142468	1142474	1142468
Stdev	3,49	4,98	4,09

Table 4: Case 5: 3 partitions measurements

Case 6	Part. 1	Part. 2	Part. 3	Part. 4	Part. 5	Part. 6	Part. 7	Part. 8
Avg	1142477	1142484	1142475	1142474	1142474	1142477	1142475	1142477
Max	1142488	1142487	1142487	1142476	1142476	1142477	1142477	1142477
Min	1142481	1142473	1142475	1142473	1142444	1142477	1142464	1142477
Stdev.	3,12	3,93	1,71	0,49	4,57	0,00	0,42	0,00

Table 5: Case 5: 8 partitions measurements

	Case 5	Case 6	Difference (C5 - C6)
Average	1142484	1142477	4
Avg	1142484	1142484	0
Max	1142500	1142488	12
Min	1142468	1142474	6

Table 6: Cases 5 and 6 comparison

6.4 Hypercall cost

Several tests have been designed to measure the cost of hypercalls. All hypercalls except those that perform a copy of the data sent or receive (read or write in ports) should have a constant cost. read and write operation on ports perform a copy of the data stream to the kernel space.

The cost measured of some of the hypercalls are shown in the next table.

Hypercall	Time	Hypercall	Time
XM_get_time(XM_HW_CLOCK)	5	XM_hm_open	31
XM_get_time(XM_EXEC_CLOCK)	7	XM_hm_status	14
XM_set_timer(XM_HW_CLOCK)	13	XM_trace_open	66
XM_set_timer(XM_EXEC_CLOCK)	13	XM_trace_status	5
XM_enable_irqs	5	XM_trace_event	13
XM_disable_irqs	5	XM_unmask_irq	5
XM_create_sampling_port	65	XM_create_queuing_message	68
XM_write_sampling_port (32)	15	XM_send_queuing_message (32)	15
XM_write_sampling_port (256)	17	XM_send_queuing_message (256)	20
XM_write_sampling_port (1024)	32	XM_send_queuing_message (1024)	30
XM_write_sampling_port (4096)	87	XM_send_queuing_message (4096)	81
XM_read_sampling_port (32)	16	XM_receive_queuing_message (32)	15
XM_read_sampling_port (256)	18	XM_receive_queuing_message (256)	19
XM_read_sampling_port (1024)	30	XM_receive_queuing_message (1024)	32
XM_read_sampling_port (4096)	83	XM_receive_queuing_message (4096)	92

Table 7: Hypercalls measurement

In read/write operation on ports numbers in parenthesis indicate the message length in bytes. Time units in microseconds.

7 Conclusions and Future Work

XtratuM 2.2 is the first implementation of an hypervisor for the LEON2 processor. The initial versions of XtratuM⁴ were designed for conventional real-time systems: dynamic partition loading, fixed priority scheduling of partitions, Linux in a partition, etc.

This version of XtratuM, besides of the porting to the SPARC v8 architecture, is a major redesign to meet highly critical requirements: health monitoring services, cyclic plan scheduling, strict resource allocation via a configuration file, etc.

The resources allocated to each partition (processor time, memory space, I/O ports, and interrupts, communication ports, monitoring events, etc.) are defined in a configuration file (with XML syntax). A tool to analyze and compile the configuration file has also been developed.

Two issues of the initial implementation of XtratuM have not ported to LEON2: the MMU and the multiprocessor support. The effort of porting the MMU support (for LEON3 with MMU) can be relatively low and will permit to step up the spatial isolation of partitions. XtratuM 2.2 code has been designed to be multiprocessor (the x86 version was multiprocessor). Therefore, XtratuM may be the faster and safer way to use a multiprocessor system (SMP⁵) in a highly critical environment.

This work was initially planned as a prototype development, however, the achieved result, in efficiency and performance, is closer to a product than a proof of concept. Next step is the formal model of the hypervisor as first step to the certification.

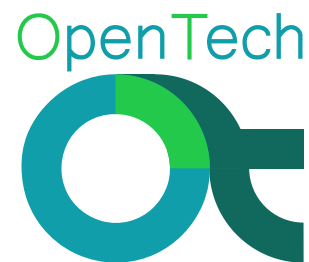
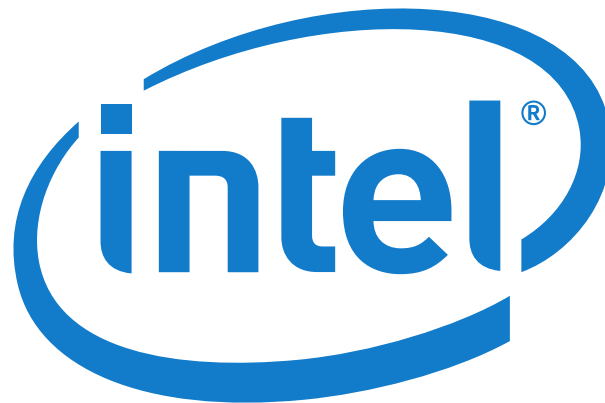
Partitioned based scheduling tools is one of the weak areas in the integration of partitioned systems with real-time constraints. The improvement of the scheduling tools to optimise the scheduling plan in another important activity to be done in the next months.

⁴XtratuM 1.2 and 2.0 were implemented in the x86 architecture only.

⁵SMP: Symmetric Multi-Processor

References

- [1] Airlines Electronic Engineering Committee, 2551 Riva Road, Annapolis, Maryland 21401-7435. *Avionics Application Software Standard Interface (ARINC-653)*, March 1996.
- [2] P. Arberet, J.-J. Metge, O. Gras, and A. Crespo. Tsp-based generic payload on-board software. In *DASIA 2009. DATA Systems In Aerospace.*, May. Istanbul 2009.
- [3] P. Arberet and J. Miro. Ima for space : status and considerations. In *ERTS 2008. Embedded Real-Time Software.*, January. Toulouse. France 2008.
- [4] M. Barabanov. A Linux-Based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, June 1997.
- [5] IBM Corporation. IBM systems virtualization. Version 2 Release 1 (2005). <http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf>.
- [6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [7] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.
- [8] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. Unix as an application program. In *USENIX Summer*, pages 87–95, 1990.
- [9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ kernel-based systems. In *SOSP*, pages 66–77, 1997.
- [10] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.
- [11] M. Masmano, I. Ripoll, and A. Crespo. Introduction to XtratuM. 2005.
- [12] M. Masmano, I. Ripoll, and A. Crespo. An overview of the XtratuM nanokernel. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPRT)*, 2005.
- [13] M. Masmano, I. Ripoll, A. Crespo, J.J. Metge, and P. Arberet. Xtratum: An open source hypervisor for tsp embedded systems in aerospace. In *DASIA 2009. DATA Systems In Aerospace.*, May. Istanbul 2009.
- [14] SCOPE Promoting Open Carrier Grade Base Platforms. *Virtualization: State of the Art*, April 2008. <http://www.scope-alliance.org>.
- [15] V. Yodaiken. The RTLinux manifesto. http://www.fsmlabs.com/developers/white_papers/rtmanifesto.pdf.



Eleventh Real Time Linux Workshop

September 28 to 30, 2009

