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 arquitecture and specially for the to 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 XtratuMdoes 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 arquitecture.

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 environments[1] 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.

[1] We 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 instructions[2] 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 thought 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 initally 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 sapce. 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 arquitecture. Although not explicitly stated in the standard, it was developed considering that the underlaying technology used to implement the partitions is the separation kernel. Athough 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

[2]Conflicting 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.

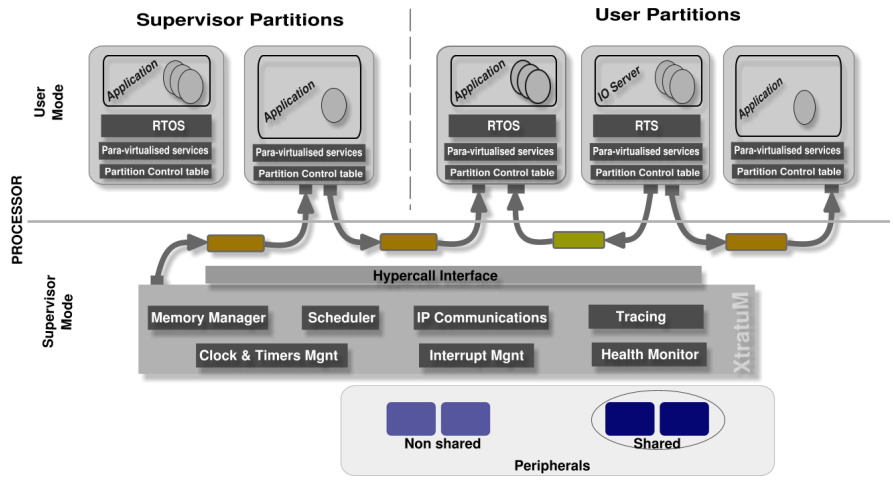

Figure 1: System architecture.

The main components of this architecture are:

Hypervisor XtratuM is in charge of virtalisation 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 decissions involving these requirements are:

XtratuM has been designed specifically to meet real-time constraints and to be as efficient as possible. The main decissions 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 though 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).

[3] Note 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.

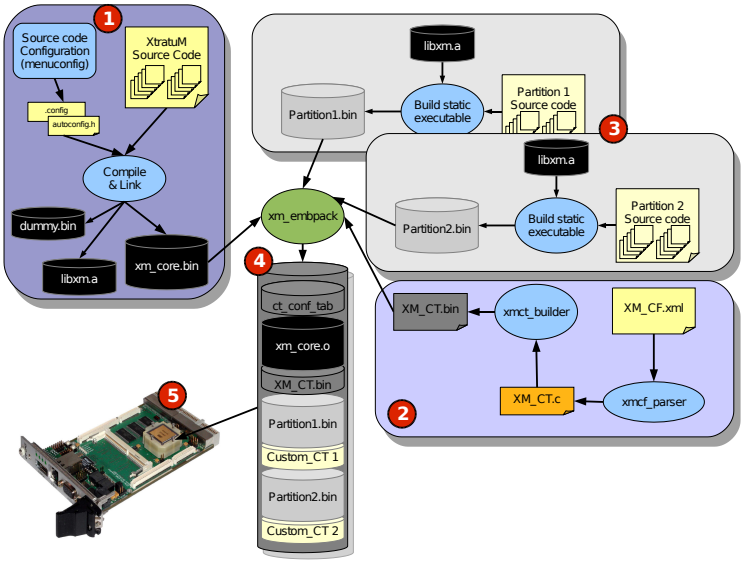

Figure 2: The big picture of building a XtratuM system.

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 begining 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 has 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 XtratuM[4] 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 (SMP[5]) in a highly critical environment.

This work was initially planned as a prototype development, however, the achieved result, in efficency 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 contraints. The improvement of the scheduling tools to optimise the scheduling plan in another important activity to be done in the next months.

[4]XtratuM 1.2 and 2.0 were implemented in the x86 architecture only.

[5]SMP: Symmetric Muli-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.*, Jannuary. 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 μkernel-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 (OSPERT)*, 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.