XTRATUM: AN OPEN SOURCE HYPERVISOR FOR TSP EMBEDDED SYSTEMS IN AEROSPACE

A. Crespo[1], **I. Ripoll**[1], **M. Masmano**[1], **P. Arberet**[2], **and J.J. Metge**[2]

[1]*Instituto de Informática Industrial, Universidad Politécnica de Valencia, Spain*
[2]*CNES, France*

ABSTRACT

XtratuM is an hypervisor designed to meet safety critical requirements. XtratuM 2.1.0 is a redesign of the former version XtratuM 2.0 (for x86 architectures) to meet safety critical requirements. It has been ported to SPARC v8 arquitecture and specially to the to the LEON2 processor, which is the reference platform for the spatial sector. Adaptation involves a strong effort in redesign to be closer to the ARINC-653 standards. 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 initial evaluation of the most significant metrics is also provided.

Key words: Partitioning systems, hypervisors,microkernels.

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 [Gol74]) 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.

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, deterministic hypervisor system calls, efficient inter-partition communication, efficient context switch, low overhead and 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. VIRTUALISATION TECHNOLOGIES

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*[Cor] is widely understood: "any way to recreate an execution environment, which is not the original (native) one"; there are

[1]We will use the terms:guest, virtual machine and partition as synonyms.

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" [SCO08]). Just to summarise the current available solutions for the real-time embedded systems:

2.1. 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.

An important disadvantage of the this solution is the large amount of code and the high complexity (the whole O.S.) of the virtualizer.

2.2. 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).

Only the micro-kernel code is executed in processor privileged-mode, the rest of the operating system subsystems (scheduler, user process management, memory services, filesystem, network stack, etc.) are implemented as micro-kernel thread. The operating system itself is designed as a set of cooperating servers; each server is a thread with its own protected memory space and communicated with other servers via the micro-kernel IPC facilities. The micro-kernel implements only the mechanisms; the policies are implemented by the micro-kernel threads at user level (even the scheduling policy).

> *"The microkernel implements only the mechanism to select one of the time partitions as active foreground partition. The policy part of deciding which of the time partitions is activated when is left to the user level. ..."*

Micro-kernel was initially presented as a solution (the right way to do) to the supposed problems of the *monolithic* kernels[2]. This strong (and enforced by the micro-kernel) isolation between the components of the operating system prevents that an error on a component affects the behavior of the rest of the system. 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. Currently the main drawback is the non negligible overhead introduced by the communication mechanism, and the high abstraction level of the processor. The virtualized operating system must be highly modified (ported) to meet the micro-kernel API and philosophy. The micro-kernel approach started with the March micro-kernel. The most representative implementation of a micro-kernel is the L4 [Lie95].

2.3. 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. Several hypervisors are available for general purpose operating systems.

2.4. Para-virtualization

The para-virtualization (term coined in the Xen [DFH+03] project) technique consist in replacing the conflicting instructions[3] 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]Linux is a monolithic kernel.

[3]Conflicting instructions: instructions that operate directly on the native hardware and may break the isolation.

2.5. 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 [MRC05a, MRC05b] was designed initally as a substitution of the RTLinux [Yod, Bar97] 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 2.0. In the rest of this paper, the term XtratuM will refer to this second version.

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 [Air96] 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 a reference in the design of XtratuM.

ARINC-653 relays on the idea of a "*separation kernel*", which basically consists in extending and enforcing the isolation between a process or a group of processes. It defines both, the API and operation of the partitions, and also how the threads or processes are managed inside each partition. From this point of view, XtratuM will cover the ARINC-653 specification related to the low level layer.

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)
- Full management of exceptions and errors via Health Monitor

4. ARCHITECTURE AND DESIGN

Figure 1 shows the complete system architecture. The main components of this architecture are:

4.1. 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), health monitoring and tracing facilities. Three layers can be identified:

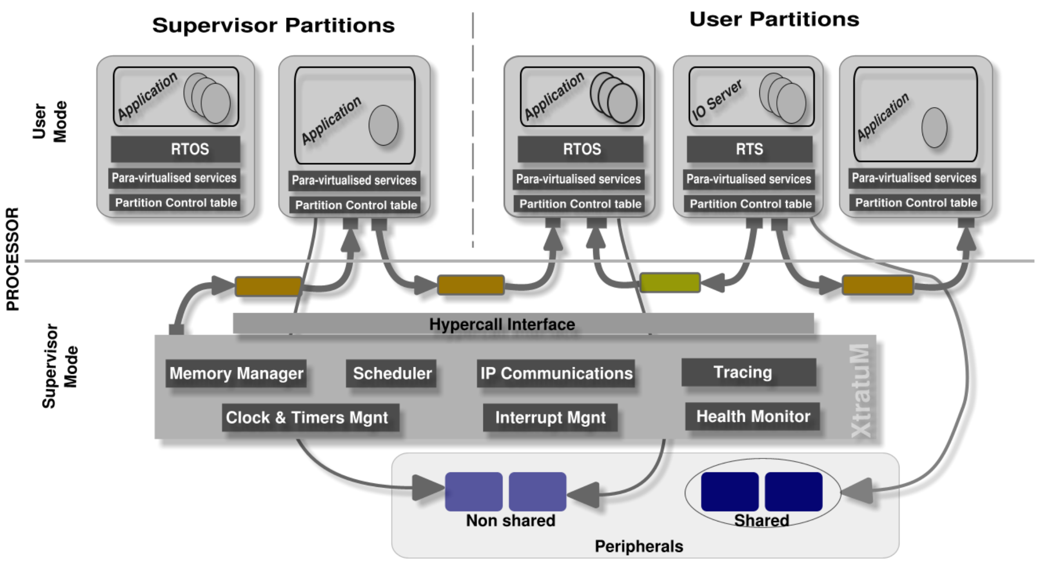

Figure 1. System architecture.

Hardware-dependent layer It implements the set of drivers required to manage the strictly necessary hardware: processor, interrupts, hardware clocks, hardware timers, paging, etc. This layer is isolated from the rest through the Hardware Abstraction Layer (HAL). Thus, the HAL hides the complexity of the underlying hardware by offering a high-level abstraction of it (for instance, a ktimer is the common abstraction to all the hardware timers).

Internal-service layer Those services are not available to the partitions. This layer includes a minimal C library which provides the strictly required set of standard C functions (e.g. strcpy, memcpy, sprintf) and a bundle of data structures. The system boot is also part of the internal services.

Virtualization-service layer It provides the services required to support the para-virtualisation services, which are provided via the hypercall mechanism to partitions. Some of these services are also used from other XtratuM modules.

Four strategies have been used to prevent partitions to pervert the temporal isolation:

- Partitions cannot disable native traps. Partitions are executed in user mode, thus guaranteeing that they have not access to control registers. Additionally,a partition can not interact with native traps.
- A partition can not mask those hardware interrupts not allocated to the partition.
- Partitions have no access to the trap table. Thus, partitions are unable to install their own trap handlers. All traps are rstly handled by XtratuM and, when required, propagated to partitions.
- Partitions can not access to the system hardware timer, but virtual timers. Partitions can set virtual timers based on native hardware clock or the execution partition clock.

4.2. Trap and interrupt management

A trap is the mechanism provided by the SPARC v8 processor to implement the asynchronous transfer of control. When a trap occurs, the processor switches to supervisor mode and jumps unconditionally into a predened handler. XtratuM extends the concept of processor traps by adding a 32 additional interrupt numbers. This new range is used to implement the concept of asynchronous event. Those asynchronous events are used to indicate to the partitions the need for attention on the occurrence of an event.

The trap mechanism is used to handle hardware interrupts, software traps and processor exceptions.

Although in a fully virtualised environment, a partition should not need to manage native interrupt hardware; XtratuM only virtualises the hardware peripherals that may endanger the isolation, but leaves to the partitions to directly manage non-critical devices. A hardware interrupt can only be allocated to one partition (in the conguration le).

A partition, using exclusively a device (peripheral), can access the device through the device driver implemented in the partition. The partition is in charge of handling properly the device. The conguration le has to specify the IO ports and the interrupt lines that will be used by each partition. Two partitions cannot use the same IO port or the same interrupt line. When a device is used by several partitions, a specic IO server partition will be in charge of the device management. An IO server partition is a specic partition which accesses and controls the devices attached to it, and exports a set of services via the inter-partitions communication mechanisms, enabling the rest of partitions to make use of the managed peripherals.

4.3. Inter-partition communication

Inter-partition communication is related with the communications between two partitions or between a partition and the hypervisor. XtratuM implements a message passing model which highly resembles the one dened in the ARINC-653. A communication channel is the logical path between one source and one or more destinations. Two basic transfer modes are provided: sampling and queuing. Partitions can access to channels through access points named ports. The hypervisor is responsible for encapsulating and transporting messages. At the partition level, messages are atomic entities. Partition developers are responsible for agreeing on the format (data types, endianess, padding, etc.). Channels, ports and the maximum message sizes and maximum number of messages (queuing ports) are entirely dened in the conguration les.

4.4. Health Monitor

The health monitor is the part of XtratuM that detects and reacts to anomalous events or states. The purpose of the HM is to discover the errors at an early stage and try to solve or conne the faulting subsystem in order to avoid or reduce the possible consequences. As as result of enforcing the isolation of the partitions, XtratuM contains a lot of consistency and security checks; therefore, it can detect a large number of errors. Errors are grouped by categories. Once an error is detected, XtratuM reacts to the error providing a simple set of predened actions to be done when it is detected.

XtratuM HM subsystem is composed by four logical components:

HM event detection : to detect abnormal states, using logical probes in the XtratuM code.

HM actions : a set of predened actions to recover the fault or conne the error.

HM conguration : to bind the occurence of each HM event with the appropriate HM action.

HM notication : to report the occurrence of the HM events.

Once an HM event is raised, XtratuM performs an action that is specified in the configuration file. Next table shows the list of HM events and the predefined default action at hypervisor and partition level.

	XtratuM scope		**Partition scope**	
Event name	Def. action	Log	Def. Action	Log
Processor triggered				
WRITE_ERROR	Warm_Reset	Y	Propagate	N
INST_ACC_EXCEPTION	Warm_Reset	Y	Propagate	Y
ILLEGAL_INST	Warm_Reset	Y	Propagate	N
PRIVILEGED_INST	Warm_Reset	Y	Propagate	Y
FP_DISABLED	Warm_Reset	Y	Propagate	N
CP_DISABLED	Warm_Reset	Y	Propagate	N
REG_HW_ERROR	Warm_Reset	Y	Suspend	Y
MEM_ADDR_NOT_ALIG	Warm_Reset	Y	Propagate	N
FP_EXCEPTION	Warm_Reset	Y	Propagate	N
DATA_ACC_EXCEPTION	Warm_Reset	Y	Propagate	Y
TAG_OVERFLOW	Warm_Reset	Y	Propagate	N
DIVIDE_EXCEPTION	Warm_Reset	Y	Propagate	N
User triggered				
PARTITION_ERROR			Halt	Y
XtratuM triggered				
MEM_PROTECTION			Suspend	Y
PARTITION_INTEGRITY			Suspend	Y
PARTITION_UNRECOV.			Halt	Y
OVERRUN	Ignore	Y		
SCHED_ERROR	Ignore	Y		
INTERNAL_ERROR	Warm_Reset	Y		
UNEXPECTED_TRAP	Ignore	Y		

Table 1. Health monitoring events and actions

4.5. Tracing

XtratuM provides a mechanism to store and retrieve the traces generated by partitions and XtratuM itself. Traces can be used for debugging, during the development phase of the application, but also to log relevant events or states during the production phase. In order to enforce resource isolation, each partition (as well as XtratuM) has a dedicated trace log stream to store its own trace messages. Trace streams str stored in buffers (RAM or FLASH). Only supervisor partitions can read from a trace stream.

4.6. 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).

XtratuM defines two types of partitions: normal and supervisor. Supervisor partitions are allowed to manage and monitor the state of the system and other partitions. Some hypercalls cannot be called by a normal partition or have restricted functionality. Note that supervisor rights are related to the capability to manage the system, and not to the ca- pability to access directly to the native hardware or to break the isolation: a supervisor partition is scheduled as a normal partition; and it can only use the resources allocated to it in the conguration le.

4.7. Design issues

XtratuM has designed specifically to meet real-time constraints and be as efficient as possible. The main decissions involving these requirements are:

Data structures are static: All data structures are predefined 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: [4] 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 interpartition interferences though hardware.

Extended management of traps, interrups and errors. All these events are handled by a module (Health Monitor) and handled properly according the configuration table. A trace system provides the mechanisms to log messages.

Full control of the resources. All the system resources allocated to partitions are specified in a configuration file.

[4]Note that XtratuM is **non-preemptive**, but it is prepared to be **re-entrant**, which allows multiple processors to execute concurrently XtratuM code.

One-shot timer. It provides a 1 microsecond resolution both for timers and clocks with a very low overhead.

4.8. 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).

LEON2 has not MMU . In a processor without MMU, the most simple and convenient way to manage system memory is with a xed partition memory scheme. The required amount of memory is allocated to each partition at boot (or load) time. Partition memory can not grow or shrink.

LEON2 provides a register window mechanism . A very particular feature of the LEON (sparcv8) processor is the register window. The register window is a mechanism used to replace the standard stack operation by a large set of internal processor registers. Partitions have full access to the 32 general purpose registers. XtratuM guarantees their consistence when the partition context-switch is performed.

XtratuM provides a transparent management of the stack. Stack overflow and underflow is directly managed by XtratuM without the intervention of the partition. The partition code must load the stack register with valid values.

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 `xmct_parser` and `xmct_builder`.

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 `xm_embpack` 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.

6. 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

The measures involving internal activity of XtratuM have been perfomed adding breakpoints at the begining and end of the code to be measured.

Internal operation	**Time (microseconds)**
Partition Context switch	25
1. Save the context	5
2. Timer interrupt management	9
3. Scheduler	7
4. Load the context	5
Clock interrupt management	2
Effective slot duration	Slot duration - 25

Table 2. Context switch evaluation

6.1. 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 0.2 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 3 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; 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).

Next table summarises the result obtained.

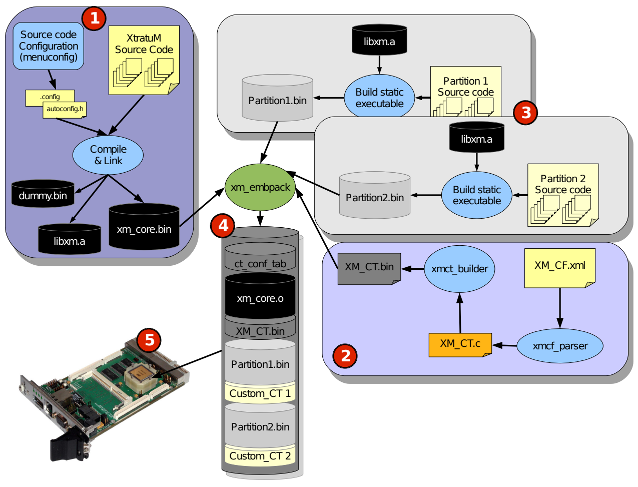

Figure 2. The big picture of building a XtratuM system.

Case	Counter	Performance (loss)
1	7272518	
2	7271682	0.011 %
3	7270643	0.026 %
4	7271706	-0.0003 %

Table 3. Performance loss

Case 4 is compared with Case 2. These results provide an idea of the overhead (or performance loss) introduced by XtratuM when a partition is split in several time slots.

In general, the overhead can be modeled with two components: the effect of the clock interrupt which occurs once each two seconds and the partition context switch which.

Next table summarizes the overhead depending on the partition slot duration. Two overhead are provided: the measured in the experiments and the theoretical. Slot duration is in milliseconds.

Slot duration	Measured (%)	Model (%)
1000	0.00	0.00
200	0.01	0.01
100	0.03	0.03
50	0.05	0.06
20	0.14	0.14
10	0.29	0.28
5	0.57	0.56
1	2.67	2.68

Table 4. Overhead

6.2. Hypercall cost

A set of test 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 are (unit microseconds)

Hypercall	Time
`XM_get_time`	9
`XM_set_time`	9
`XM_enable_irqs`	10
`XM_unmask_event`	1
`XM_create_sampling_port`	23
`XM_write_sampling_port` (16bytes)	19
`XM_write_sampling_port` (1024bytes)	52
`XM_write_sampling_port` (4096bytes)	153
`XM_read_sampling_port` (16bytes)	19
`XM_read_sampling_port` (1024bytes)	53
`XM_read_sampling_port` (4096bytes)	154

Table 5. Hypercall measurement

Read and write operations are implemented in a naive way: performing one byte at once. The results obtained are coherent with the implementation. There is a large marging of improvement on this implementation. An optimisation of these and other hypercalls will be done in the next version of XtratuM.

7. CONCLUSIONS

XtratuM 2.1.0 is the first implementation of an hypervisor for the LEON2 processor. The initial versions of XtratuM[5] 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,

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

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 2.1.0 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[6]) in a highly critical environment.

7.1. Future work

Although this work was planned as a prototype development, the result achieved, in efficency and performance, is closer to a product than a proof of concept. Additional steps are being performed to obtain a product in terms of procedures (following the appropriated standars), documentation and code optimisation.

The roadmap of XtratuM includes the porting to LEON3 processor and the consideration of several aspects as multi-plan, multi-core and other scheduling policies. Additionally, the design and implementation of a minimal run-time following the ARINC-653standard for partitions is under study.

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.

REFERENCES

[Air96] Airlines Electronic Engineering Committee, 2551 Riva Road, Annapolis, Maryland 21401-7435. *Avionics Application Software Standard Interface (ARINC-653)*, March 1996.

[Bar97] M. Barabanov. A Linux-Based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, June 1997.

[Cor] IBM Corporation. IBM systems virtualization. Version 2 Release 1 (2005). http://publib.boulder.ibm.com/infocenter/-eserver/v1r2/topic/eicay/eicay.pdf.

[DFH$^+$03] 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.

[Gol74] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[Lie95] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[MRC05a] M. Masmano, I. Ripoll, and A. Crespo. Introduction to XtratuM. 2005.

[MRC05b] 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.

[SCO08] SCOPE Promoting Open Carrier Grade Base Platforms. *Virtualization: State of the Art*, April 2008. http://www.scopealliance.org.

[Yod] V. Yodaiken. The RTLinux manifesto. http://www.fsmlabs.com/developers/white_papers/rtmanifesto.pdf.

[6]SMP: Symmetric Muli-Processor