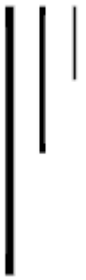


University of Piraeus - Department of Informatics
Postgraduate Programme
"Advanced Information Systems"

Postgraduate Thesis

Thesis Title	**Evaluation of a hypervisor for real-time embedded systems** **Αξιολόγηση ενός υπερ-επόπτη για ενσωματωμένα συστήματα πραγματικού χρόνου**
Student Name	**Georgios V. Pikoulis**
Registration number	**MPSP/12061**
Supervisor	**Mihalis Psarakis, Assistant Professor**

Submission Date **October 2016**

Examination Committee

(signature)	(signature)	(signature)
Mihalis Psarakis (supervisor)	Charalampos Konstantopoulos	Aggelos Pikrakis

Contents

List of Figures

List of Tables

Acknowledgements

The following thesis assignment could not have been realized without the inestimable help of my supervisor, Assistant Professor Michalis Psarakis, whose guidance was instrumental to the completion thereof.

Special mention is also due to assistant supervisor, Dr. Aitzan Sarri, who sacrificed precious time from his own research to collaborate with me.

Additionally, I offer my sincere thanks to Alexandros Karmiris, for his valuable assistance in the grammatical correctness of this document.

Finally, I would like to extend my deepest gratitude to my loving mother and father, for their boundless support during my studies.

Abstract

Hypervisors offer a way to achieve software functionality isolation, by partitioning system elements. While providing isolation, it is unavoidable that each partitioned element will have slower execution times compared to a standalone element. Depending on the application, such delays might not be acceptable, so it is vital to have prior knowledge of those limitations and design accordingly.

Embedded systems can make use of a hypervisor to partition their functionality, but they can be particularly restricted in their timing deadlines. Delays introduced by the hypervisors have to be observed and taken into account in the system's task schedule.

The mission of this study is to evaluate the performance of such partitions, when managed by the XtratuM, a freely distributed hypervisor. We chose this approach in order to study the use and performance of a hypervisor in real-time embedded systems.

To evaluate the performance of the XtratuM, we will measure the execution time of a test routine in various scenarios. To present a meaningful comparison, we will juxtapose the results with the execution time of the test routine when run without the intervention of the hypervisor.

Οι υπερ-επόπτες είναι ένας τρόπος ώστε να επιτευχθεί απομόνωση των λειτουργιών ενός συστήματος και το επιτυγχάνουν αυτό διαχωρίζοντας το σύστημα σε διακριτά κομμάτια. Αν και παρέχεται αυτή η απομόνωση, είναι αναπόφευκτο κάθε διαχωρισμένο κομμάτι να παρουσιάζει πιο αργούς χρόνους εκτέλεσης σε σύγκριση με ένα ενιαίο σύστημα. Αναλόγως με την εφαρμογή, αυτές οι καθυστερήσεις μπορεί να μην είναι αποδεκτές, γι αύτό και είναι πολύ σημαντικό να υπάρχει πρότερη γνώση των περιορισμών και αναλόγως να πραγματοποιηθεί η σχεδίαση.

Στα ενσωματωμένα συστήματα μπορεί να χρησιμοποιηθεί ένας υπερ-επόπτης ώστε να απομονωθούν στοιχεία της λειτουργικότητάς τους, αλλά είναι πιθανό αυτά τα συστήματα να πρέπει να τηρούν ιδιαίτερα αυστηρές χρονικές προθεσμίες. Οι καθυστερήσεις που εισήγαγε ο υπερ-επόπτης πρέπει να είναι γνωστές, ώστε να λαμβάνονται υπόψη στο χρονοδιάγραμμα εργασιών του συστήματος.

Ο στόχος της παρούσας μελέτης είναι η αξιολόγηση των επιδόσεων διαχωρισμένων κομματιών, διαχειριζόμενα από τον δωρεάν διανομής υπερ-επόπτη XtratuM. Επιλέξαμε αυτή την κατεύθυνση με σκοπό την μελέτη της χρήσης και επιδόσεων ενός υπερ-επόπτη σε ενσωματωμένο σύστημα πραγματικού χρόνου.

Για να αξιολογηθούν οι επιδόσεις του XtratuM, θα μετρήσουμε το χρόνο εκτέλεσης διάφορων σεναρίων. Για να μπορέσουμε να παρουσιάσουμε μια ουσιαστική σύγκριση, θα αντιπαραβάλουμε τα αποτελέσματα με το χρόνο εκτέλεσης των σεναρίων όταν εκτελούνται χωρίς την παρέμβαση του υπερ-επόπτη.

1 Introduction

From the advent of information technology we have relied on computers to perform various tasks with speed and repeatability that no human could hope to achieve. However computers are not unerring. Due to various circumstances, be it software that was not designed to overcome every possible situation or electrical faults in memory caused by ionizing radiation, a computer can indeed arrive at a state of error. Depending on the task given to complete, this could be from being just irritating to even fatal. Let's take for example the automobile ECU[1] ecosystem. Audio artifacts through the media system can annoy the driver and passengers, but it would be an enormous overstatement to claim that this could cause an accident. On the other hand, a malfunctioning driving assistance system like the ABS[2] or ESP[3], could very well lead to multiple injuries or deaths. As such, it would make sense to segregate the computer systems, so that critical and non-critical parts do not interfere with each other. Indeed, this is standard practice in industrial sectors where computers execute critical tasks, like automotive, medical, aerospace, military, nuclear and such [Sto96].

An approach to achieve system segregation, is to isolate mission critical functions in autonomous hardware. While this solution achieves physical isolation, since each hardware device has its own processor and memories, it increases the system cost and complexity, and still needs to be interconnected with other computers on site. As such, it makes sense to limit the functionality of such systems to the bare minimum, in order to avoid needless complexity, which could introduce entropy and errors. Those purpose build, hardware constrained, relatively simple systems are called *embedded systems* and have dominated all areas where mission critical computers are needed.

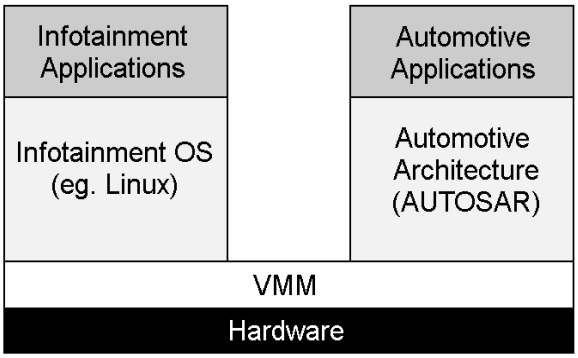


Figure 1: Hypervisors enable usage of diferent OSes on the same hardware device [NDB10].

As computer performance kept on increasing over the years, embedded systems were uplifted too. Whereas in earlier years *microcontrollers*[4] were mostly utilized as deeply embedded processors running OS-less *assembler* and *C* code, they caught up with RTOSes and in recent years with fully stacked OSes like *Linux*. Moreover additional features were added, like virtualization extensions, memory protection and supervisor modes, features that simplify virtualization. The above advancements made possible for embedded systems to make use of hypervisors. Acting as a layer between the hardware and the user application, a hypervisor can achieve functionality isolation, while allowing critical and non-critical functions to run on the same processor [For10], in different segments called *partitions*.

Virtualization in embedded systems introduces an array of desirable features in mission critical environments [Hei08][NDB10].

- Reduced system complexity by reduction of devices in lieu of fewer, more powerful devices. Interconnection between those devices has also fewer physical interfaces and signals, but greater bandwidth.
- Re-use of legacy systems made easier by abstracting the older or even discontinued hardware in the hypervisor layer and keeping the application code mostly the same.
- Running different OSes on the same physical machine, allows systems designers to more easily choose the best devices for each use case. Critical code can still run in bare metal partitions, if needed.
- Partition isolation allows use of non-critical, non-secure software in devices which share resources with critical systems. In case of failure or malicious intrusion, the critical systems are not vulnerable to attacks via the affected systems.

[1] Electronic Control Unit is a generic term for any embedded system that controls one or more of the electrical system or subsystems in a transport vehicle.
[2] Anti-lock braking system.
[3] Electronic stability control.
[4] A microcontroller is a processor with all the memories and peripherals required, in order to act as a standalone computer.

- Monitoring multiple partitions on the same system, allows for higher level detection of fault and error patterns.
- Power management in clusters, by moving virtual machine partitions off lightly loaded hypervisors, which can then be shut down.

We can recognize that the computer industry has not remained idle on the matter. More than ten commercial hypervisors are actively used in embedded systems, as we explore on the next chapter, showing that this technology is not any more just a matter of academic research. Even so, performance evaluation for hypervisors in embedded systems has not yet been thoroughly explored. While bibliography of research on hypervisors for personal computers and servers is quite rich, with Barham et al., 2003 [Bar+03] publishing an excellent evaluation of the *Xen* hypervisor's performance, embedded systems hypervisors are lacking such publications. Thus on the scope if this dissertation, we are motivated to evaluate and analyze the performance of an hypervisor designed for embedded systems.

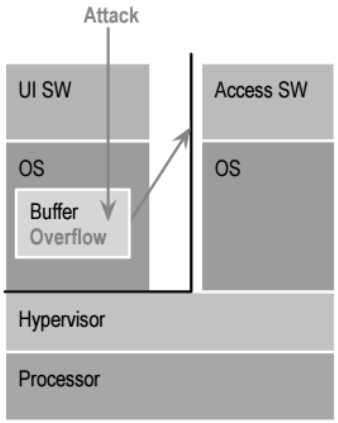


Figure 2: Standard security use case: A user or networkfacing OS is compromised, but encapsulated in a VM, which protects the rest of the system from the exploit [Hei08].

One such hypervisor is the XtratuM[5], brought by the Instituto de Automática e Informática Industrial[6] of the Universitat Politècnica de València in Spain[7]. We chose to evaluate the XtratuM on basis of its mission critical orientation, applying the philosophy of the ARINC-653 [inc03] software specification and support of RTOSes designed for embedded systems. XtratuM has been deployed, among others, in projects of Thales Alenia Space[8] and Astrium[9] for the aerospace industry. Moreover, XtratuM offers a free academic version for the x86 architecture, along with the necessary SDK[10] for ease.

On this thesis we utilize a set of tests, in order to evaluate the performance of the aforementioned hypervisor in various scenarios. We will measure the execution time of a test program written in the *C* language, while running on a partition managed by the hypervisor. We will then compare the results and describe what knowledge we can extract from the measurements.

[5]http://www.xtratum.org/
[6]http://www.ai2.upv.es/en/index.php
[7]http://www.upv.es/
[8]http://www.thalesgroup.com/Markets/Space/Home/
[9]http://www.astrium.eads.net/
[10]Software Development Kit.

2 Hypervisors

2.1 Virtualization and hypervisors

The term virtualization dates back to 1967, where the IBM Thomas J. Watson Research Center[11] developed an experimental computer system, the *M44/44X* to explore among others, the virtual machine concept. The system was based on an IBM 7044 (the 'M44') and simulated multiple 7044 *virtual machines* (the '44X') [ONe67].

To better explain what a *hypervisor* or *VMM* is, the concept of the virtual machine must be described. A virtual machine is taken to be *an efficient, isolated duplicate* of the real machine [PG74]. The process of duplicating the isolated machine is called *virtualization*. We can only achieve virtualization by means of a software called *virtual machine monitor* (VMM). The VMM shall provide an environment for programs

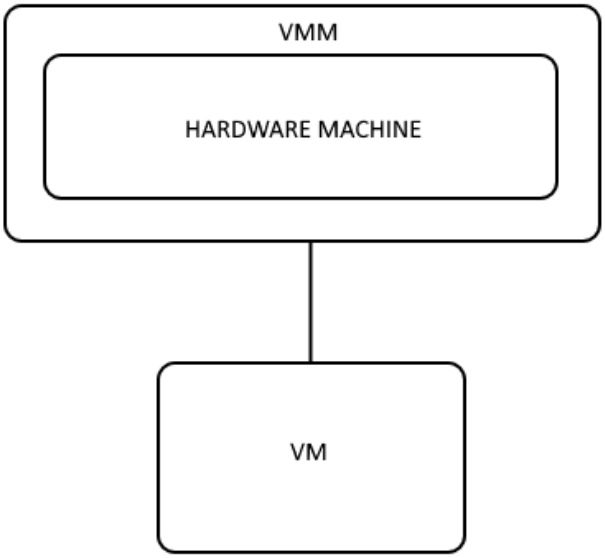


Figure 3: The virtual machine monitor

essentially identical to the real machine, programs shall exhibit the lowest possible decrease of execution speed and the VMM shall be in total control of all system resources available to the programs. The VMM shall be able to handle multiple virtual machines.

By an essentially identical environment we mean that any program run within the VMM, shall exhibit an effect identical to the one when run in the original machine. It is most possible and acceptable that there will be differences caused by the availability of system resources. This could be because it is desirable to allocate part of the total machine memory and processing cores to the virtual machine. Moreover, there can exist differences caused by timing dependencies, because of the intervening levels of software between machine and program or because of other virtual machines running concurrently.

In order to exhibit the lowest possible speed decrease and achieve high efficiency, the virtual machine's processor instructions shall be mostly run on the real processor. This means that the virtual machines cannot be expected to run cross platform; the programs have to be written for the architecture of the real machine. As such, emulators and simulators cannot be called VMMs.

Finally, system resource control is the feature that enables the VMM to disallow the programs access to cores, memories, peripherals and the like, *without explicitly being allocated to*. It is also possible that, if needed, the VMM will be able to disallow access to resources previously allocated.

2.1.1 Classification

Virtual machine monitors or *hypervisors* can be classified in to distinct types, as described by Robin in *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor* [RI00]:

- Type-1, native or bare-metal hypervisors. These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems. For this reason, they are sometimes called bare metal hypervisors. A guest operating system runs as a process on the host.

[11] The Thomas J. Watson Research Center is the headquarters for IBM Research. The center comprises two sites, with its main laboratory in Yorktown Heights, New York, U.S., 38 miles north of New York City and offices in Cambridge, Massachusetts.

- Type-2, hosted hypervisors. These hypervisors run on a conventional operating system just as other computer programs do. Type-2 hypervisors abstract guest operating systems from the host operating system.

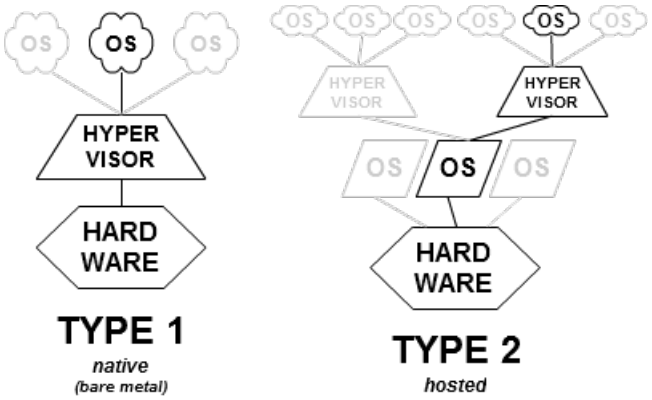


Figure 4: Type-1 and type-2 hypervisors [Wik11]

Robin describes in more detail the requirements of each hypervisor type, based on Goldberg's dissertation [Gol73].

A Type-1 hypervisor runs directly on the machine hardware, so it must act as an operating system or kernel that has mechanisms to support virtual machines. A Type-1 hypervisor must perform scheduling and resource allocation for all virtual machines in the system. This means that a Type-1 hypervisor may be much larger than Type-2 hypervisor because of the extra code needed to implement these features. Furthermore, a Type-1 hypervisor requires drivers for hardware peripherals. Goldberg develops a set of rules to determine if processor hardware is capable of supporting virtual machines and thus could be a host for a Type-1 hypervisor. His three requirements for virtualization are:

- The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. For example, a processor can not use an additional bit in an instruction word or in the address portion of an instruction when in privileged mode.
- There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.
- There must be a way to automatically signal the hypervisor when a VM attempts to execute a sensitive instruction. It must also be possible for the hypervisor to simulate the effect of the instruction. Sensitive instructions include:
  - Instructions that attempt to change or reference the mode of the VM or the state of the machine.
  - Instructions that read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.
  - Instructions that reference the storage protection system, memory system, or address relocation system. This class includes instructions that would allow the VM to access any location that is not in its virtual memory.
  - All I/O instructions.

A Type-2 hypervisor runs as an application under a host operating system. A Type-2 hypervisor should be simpler than a Type-1 hypervisor because the memory management, processor scheduling, resource allocation, and hardware drivers of the host operating system are used in its implementation. A Type-2 hypervisor provides only virtualization support services. The Type-2 hypervisor virtualizes the real machine even though the hypervisor is running as an application in the host OS. To support a Type-2 virtual machine a processor must meet all of the hardware requirements for the Type-1 hypervisor listed above. However, in addition to these requirements, there are software requirements for the host operating system that a Type-2 hypervisor runs on. The host OS requirements are:

- The host OS can not do anything to invalidate the requirement that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.

- There must be primitives available in the host OS to protect the hypervisor and other VMs from the active virtual machine. Examples of this primitive include a protection primitive, address translation primitive, or a sub-process primitive. When the virtual machine traps[12] because it attempted to execute a sensitive instruction, the host OS must direct the signal to the hypervisor. Therefore, the host OS needs a primitive to perform this action. The host OS also needs a mechanism to allow a hypervisor to run the virtual machine as a sub-process. The hypervisor must still be able to simulate sensitive instructions.

The most efficient type of hypervisor to use depends on the use case, as both excel in some areas, while lacking in others. Type-1 hypervisors provide higher performance, availability, and security than Type-2 hypervisors, due to bypassing the OS layer, becoming a type of OS or kernel themselves. One disadvantage, which is a Type-2 advantage, is that Type-1 hypervisors' hardware support is limited to exactly what the hypervisor was designed for, while Type-2 hypervisors use the OS drivers to abstract the hardware. Ease of use is also an issue, as Type-2 hypervisors run on a familiar user interface, that of the underlying OS.

2.1.2 Types of virtualization

Virtual machines can communicate with the underlying hardware with the hypervisor mediating the access. Depending on the way this intervention occurs, different virtualization methods can be described:

- Full Virtualization
- Hardware-assisted virtualization
- Paravirtualization

In *Full Virtualization* the virtual machine can access all the hardware that are required by the *guest operating system*, in such a way that the OS is unmodified and unaware that it is being virtualized. Hardware request including the full instruction set, input/output operations, interrupts and memory access are routed to hypervisor drivers that take the place of the original hardware drivers. With this approach, critical instructions are discovered, *statically or dynamically at run-time*, and replaced with traps into the VMM to be emulated in software. Full virtualization offers the best isolation and security for virtual machines, and simplifies migration and portability as the same guest OS instance can run on a VMM or on native hardware. On the other hand a key challenge for full virtualization is the interception and simulation of operations, that either access or affect state information that is outside the virtual machine, like hardware I/O operations. Virtualizing each and every such instruction in not a trivial task during VMM development. Full virtualization translation can incur a large performance overhead in comparison to a virtual machine running on natively virtualized architectures, since the VMM has to mediate every hardware request.

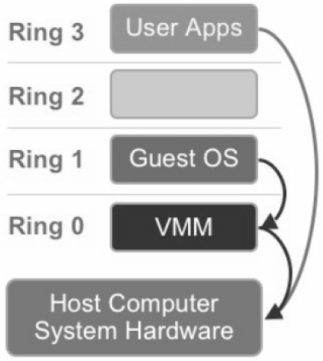


Figure 5: Full Virtualization [Hua12].

Hardware-assisted virtualization is similar to full virtualization, but the host processor on the real machine provides *special instructions* to aid the virtualization. In the traditional CPU architecture, OS kernels expect direct CPU access, directly on the most privileged level. With virtualization the guest OS cannot run on this level, as the VMM operates there, meaning that the guest OS will run in a less privileged environment. Several CPU instructions are only accessible at the highest privilege level, which means the OS has to be provisioned somehow in order to execute those. One such provision is the introduction of an even more privileged level, available only for virtualization. This way The VMM can still run at a the most privileged level, while the hosted OS can execute most CPU instructions directly on the hardware, while being monitored by the VMM.

[12]In computing and operating systems, a trap, also known as an exception or a fault, is typically a type of synchronous interrupt typically caused by an exceptional condition (e.g. division by zero, invalid memory access).

Thus, direct CPU instructions can be called by the virtual machine, without the VMM intervention and trap overhead. Moreover, no changes on the virtual machine OSes are needed, as long as they support the CPU's hardware extensions via their drivers. The disadvantage of this method is that it can be applied only to certain CPUs that support it, meaning migration from one architecture to the other, or even different version of the same architecture can be problematic.

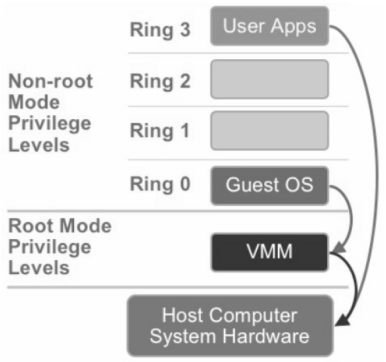


Figure 6: Hardware-assisted Virtualization [Hua12].

Another way to provision a virtual machine's access to CPU instructions is *Paravirtualization*. This method allows the virtual machine OS to be aware of the virtualization process and use special drivers and API to access the VMM instead of the real machine. To achieve this, the virtual machine OS has to be modified by *replacing processor requests with VMM requests*. With paravirtualization the CPU is not required to support virtualization and neither CPU calls are required to be fully virtualized, but are still needed to be trapped so that hardware is still isolated from the virtual machine. The downside of this method is that there still exists the overhead of trapping processor instructions, albeit smaller than full virtualization, and that migration to another OS is impossible, unless the new OS is patched for paravirtualization. However, it is worth to mention that components may be available that enable many of the significant performance advantages of paravirtualization. For example, the Xen Windows GPLPV[13] project provides a kit of paravirtualization-aware device drivers, licensed under the terms of the GPL, that are intended to be installed into a Microsoft Windows virtual-guest running on the Xen hypervisor.

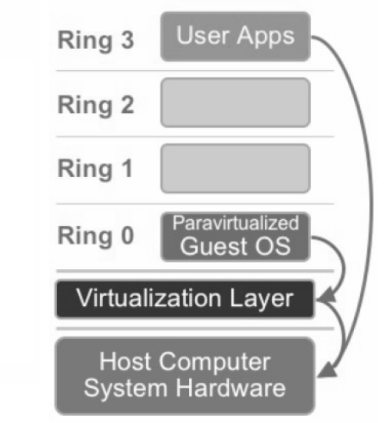


Figure 7: Paravirtualization [Hua12].

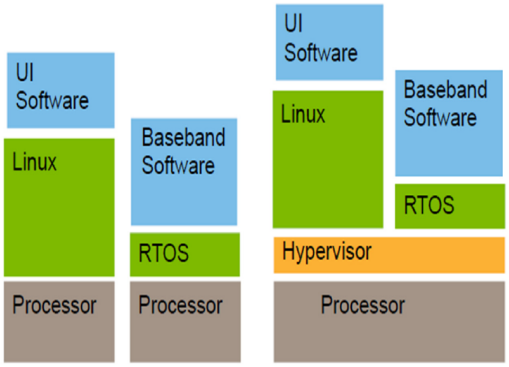
2.2 Virtualization in embedded systems

One of the most common applications for hypervisors in embedded systems today is in mobile phones, where trusted and secure applications (baseband management) share the platform with third-party and untrusted applications. The isolation that the hypervisor provides is a key attribute to their success in this domain. But this is not the only advantage of using a VMM in a mobile phone [Maz16].

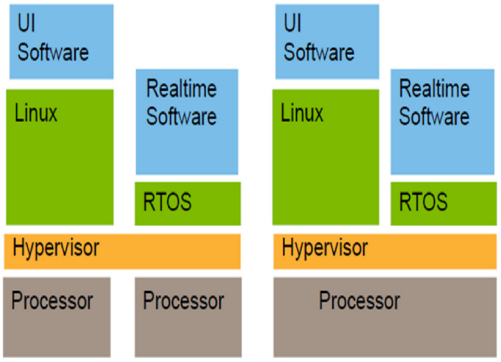
- Processor consolidation where applications are run on a higher level OS and baseband management is run on an RTOS.

[13] http://wiki.xen.org/wiki/XenWindowsGplPv

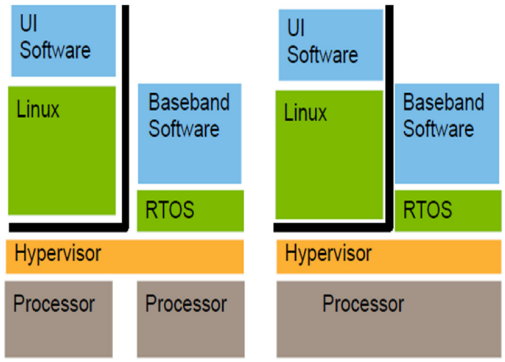
- Software and hardware architecture abstraction allows range of related products with varying capabilities, that use of same parts of software on high and low end devices.
- Users can be allowed to configure the OS at their needs, without compromising the device's main use of telephony.



(a) Processor consolidation.



(b) Software and hardware architecture abstraction.



(c) User configurable OS.

Figure 8: Virtualization advantages for mobile devices [Maz16].

Applications for hypervisors are growing outside of handsets and tablets. Deeply embedded avionics and automotive applications are also benefiting by taking advantage of the isolation and reliability aspects of hypervisors. Systems with a focus on security, survivability, or high configurability are finding applications for the technology.

In the automotive market 90% of new functions use software and electronics evaluate 40% of total vehicle costs [NDB10]. The huge complexity of 80 ECUs, 2500 signals, 6 networks, multi-layered run-time environment, multi-source software, multi-core CPUs, etc is a reality for car electronics engineers. Strong costs, safety concerns, reliability issues, time-to-market pressure, and reusability of resources are a driving force for the use of virtualization.

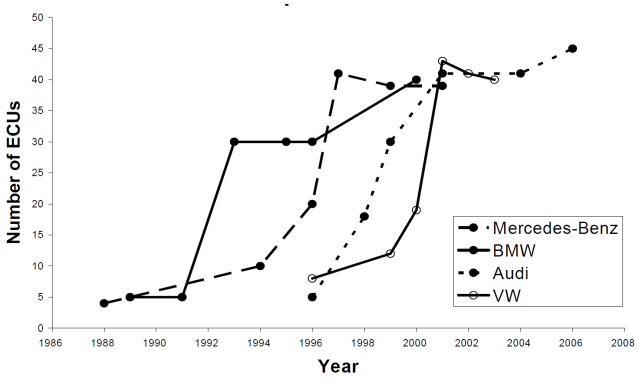


Figure 9: Proliferation of ECUs in the automotive sector increases system complexity [NDB10].

Using a lower number of high end processors instead of a multitude of low end ECUs, complexity can be greatly reduced, along with the interconnections. Software for security-critical sub-systems can run on the AUTOSAR[14] environment while non critical systems can run on OSes. Updates can be applied over the air, without the need of specialized hardware, as the critical parts will be isolated and the VMM will intercept any faults caused from updates. Software developed for older processor architectures, like parking sensors, can be reused by abstracting the hardware at the hypervisor level.

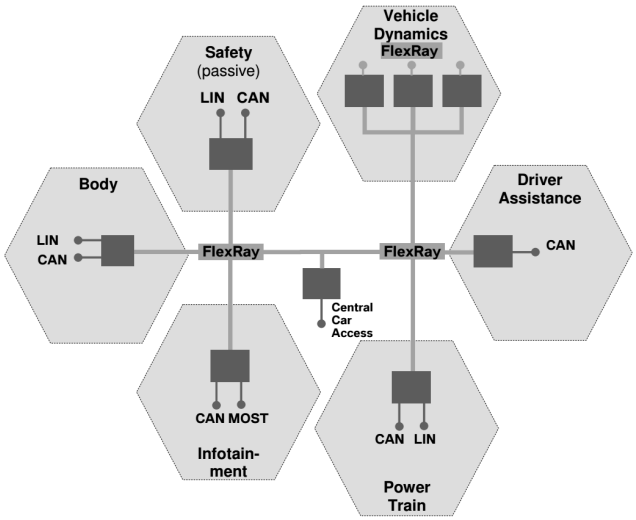


Figure 10: Faster ECUs with hypervisors, faster interconnects and lower complexity is what BMW foresees for the automotive market [Sch07].

Although the benefits derived from virtualization are similar for general and *aerospace/defense* use, some of the requirements differ dramatically. Compared to a consumer device, a chassis for an aerospace/defense application needs to power on/off gracefully and rather quickly without issues or corruption of data, meeting hard requirements for startup and shutdown times. This may require careful scripting of the platform management software to ensure the different hardware and software components are shut down and brought up in the correct order. Another difference is *A/D* requirements for ease of serviceability are often more rigorous than consumer systems, given that soldiers in the field may have less experience and fewer available resources than specialized engineers. While soldiers operating virtualized systems are highly trained, they typically have the same skill level as a network planner with 10 years of experience [Cor11]. Therefore, A/D systems should have relatively simple user interfaces for start up and shut down. Meeting the above example requirements, hypervisors can offer substantial benefits in aerospace and defence applications.

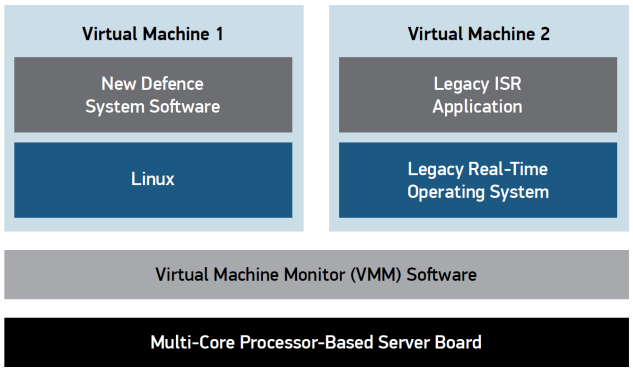


Figure 11: Running an ISR Application Unmodified on a New Platform. [Cor11].

When it comes to legacy code in many military applications, a common perspective is, "If it ain't broke, don't fix it." Unfortunately, it's not always that simple. When adopting new hardware, many times it is necessary to migrate to a new operating system. This will require the legacy applications to be ported, which often necessitates rewriting and retesting the code. This rework may even be required when using the same OS

[14] AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of automotive interested parties founded in 2003. It pursues the objective of creating and establishing an open and standardized software architecture for automotive electronic control units excluding infotainment. www.autosar.org/

vendor whose new OS isn't 100% backward compatible with earlier versions. Further complicating migration, legacy code written in assembly language is likely to be single threaded and therefore unable to take advantage of the performance improvements of multi-core processors. Overcoming these challenges, virtualization allows systems to execute legacy software with little or no modification. Legacy code runs on its native OS in an isolated VM, which also protects it from unintended interactions with other applications. Furthermore, multi-core processors can be fully utilized since the VMM uses all the available processor cores to support the VMs. Take for example a legacy intelligence, surveillance and reconnaissance (ISR) application, running on a near real-time operating system (RTOS), which acquires and processes radar images. The software is part of a new platform that is Linux-based. Use of virtualization enables the legacy ISR application to run unmodified on its own in a VM.

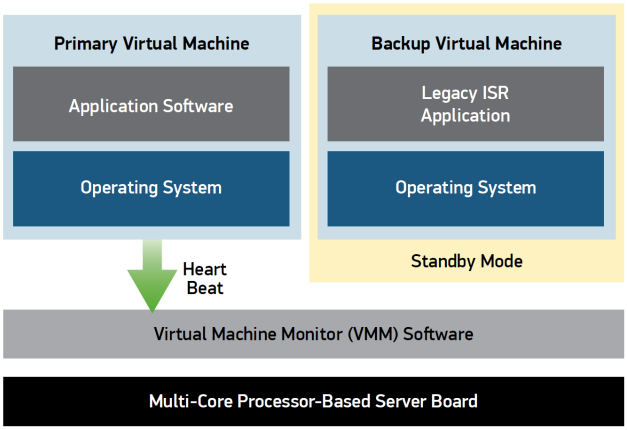


Figure 12: Primary and Backup VMs Provide Robust Failover. [Cor11].

When the operating system or application crashes, the standard remedy is a hardware reboot, which takes the system offline for a period of time. With virtualization, a failing operating system or application doesn't have to be catastrophic since it is isolated in a VM. One option is to restart the software in the failing VM without impacting the other VMs; however, this approach does not address a corrupted software image. Therefore, a more robust solution is to maintain a backup VM (e.g., identical software) in standby mode that is ready take over in a matter of seconds. Working behind the scenes, the VMM provides high availability and management functions that are critical when building highly reliable systems. One of its key functions is to detect faults and manage failover to ensure continuity of service. The VMM monitors the application, and if it detects deteriorating performance, it can take action such as provisioning a new virtual machine for the application to run, even on separate hardware if required. Hypervisors can support a fault tolerant model where a secondary VM runs in lockstep with the primary VM. If the performance of the primary VM degrades, the virtualization software can failover to the secondary VM.

2.3 Available Type-1 hypervisors

Below we introduce some of the type-1 hypervisors available on July 2016. The list is non-exhaustive.

Xen Project[15] originated as a research project at the University of Cambridge, led by Ian Pratt, senior lecturer at Cambridge who co-founded XenSource, Inc. with Simon Crosby also of Cambridge University. The first public release of Xen was made in 2003. Xen Project has been supported originally by XenSource Inc., and since the acquisition of XenSource by Citrix in October 2007.

Xen supports five different approaches to running the guest operating system: HVM (hardware virtual machine), HVM with PV drivers, PVHVM (HVM with PVHVM drivers), PVH (PV in an HVM container) and PV (paravirtualization). The Xen hypervisor has been ported to a number of processor families.

- Intel: IA-32, IA-64 (before version 4.2), x86-64.
- PowerPC: previously supported under the XenPPC project, no longer active after Xen 3.2
- ARM: previously supported under the XenARM project for older versions of ARM without virtualization extensions, such as the Cortex-A9. Currently supported since Xen 4.3 for newer versions of the ARM with virtualization extensions, such as the Cortex-A15.
- MIPS: XLP832 experimental port.

[15] www.xenproject.org/
[16] wiki.xen.org/wiki/Xen_Project_Software_Overview/

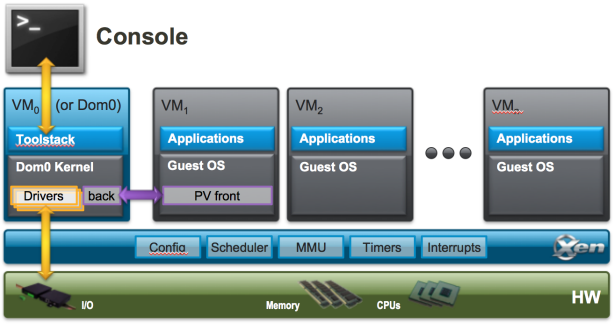


Figure 13: Xen project architecture[16].

Using the Xen project, various implementation built upon the GNU General Public Licence free core, to produce a commercial version, adding proprietary additions:

- Citrix XenServer
- Huawei FusionSphere
- Oracle VM Server for x86
- Thinsy Corporation

Proxmox Virtual Environment[17], or Proxmox VE, is an open-source server virtualization environment. It is a Debian-based Linux distribution with a modified Red Hat Enterprise Linux kernel and allows to deploy and manage virtual machines and containers.

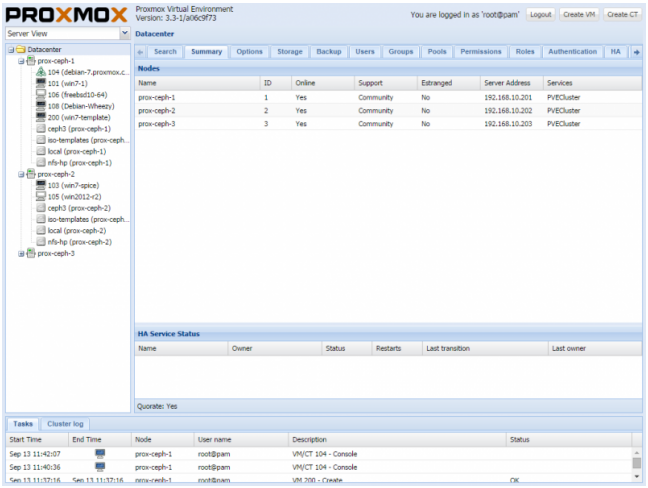


Figure 14: Proxmox startpage with 3 Cluster nodes.

Proxmox VE includes a Web console and command-line tools, and provides a REST[18] API for third-party tools. Two types of virtualization are supported: container-based with LXC and full virtualization with KVM. It comes with a bare-metal installer and includes a Web-based management interface.

Nuxis[19] is an hypervisor with resources and tools to streamline the management of IT services, such email, file server, backup, printers, databases, Firewall, VoIP, ERP.

Nuxis supports multiple operating systems on 32bit and 64bit architectures, Linux and Windows. As for virtualization paravirtualized and fully virtualized (with HVM support) host are supported. Management is available via a web interface.

L4, was created by German computer scientist Jochen Liedtke [Lie96] as a response to the poor performance of earlier microkernel-based operating systems. Liedtke felt that a system designed from the start for high performance, rather than other goals, could produce a microkernel of practical use. His original implementation in hand-coded Intel i386-specific assembly language code in 1993 sparked off intense interest in the computer

[17] pve.proxmox.com/wiki/Main_Page/
[18] Representational state transfer
[19] http://www.nuxis.com/

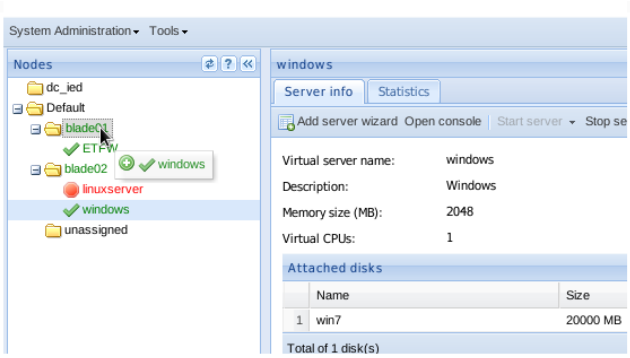


Figure 15: Live migration of a VM on Nuxis.

industry. Since its introduction, L4 has been developed for platform independence and also in improving security, isolation, and robustness. There have been various re-implementations of the original binary L4 kernel interface (ABI) and its successors, including L4Ka::Pistachio (Uni Karlsruhe), L4/MIPS (UNSW) and Fiasco (TU Dresden).

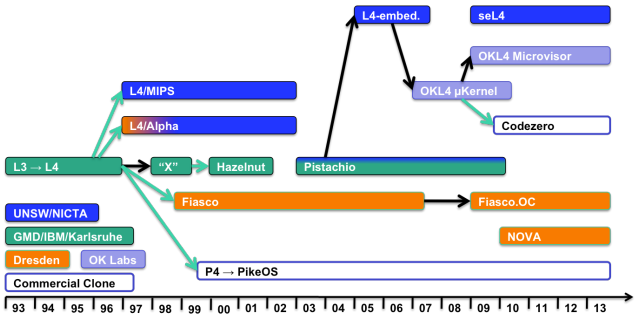


Figure 16: L4 branching to to various implementations.

For this reason, the name L4 has been generalized and no longer only refers to Liedtke's original implementation. It now applies to the whole microkernel family including the L4 kernel interface and its different versions. L4 is widely deployed. Depending on the implementation L4 implementation can support all types of virtualization, numerous processor architectures and even be formally verified for high assurance (seL4) or used in deeply embedded ARM Cortex M3/M4 devices (F9 microkernel). Examples of industry applications based on L4 include the following:

- Apple's iOS secure coprocessor based on L4 shipped and estimated 310 million devices for year 2015.
- OKL4 on more than 1.5 billion handsets from 2007 and unreleased amount of Bosch car infotainment systems.
- Under the DARPA High-Assurance Cyber Military Systems (HACMS) program, NICTA together with project partners Rockwell Collins, Galois Inc, the University of Minnesota and Boeing are developing a high-assurance drone based on seL4, with planned technology transfer onto the optionally piloted autonomous Unmanned Little Bird helicopter under development by Boeing.

XtratuM is a bare-metal hypervisor specially designed for embedded real-time systems available for the instruction sets of x86, LEON2 and LEON3 (SPARC v8), ARM Cortex-R4F processors and supports apart from XAL for bare-C applications, Partikle RTOS, LITHOS RTOS, uLITHOS runtime, Ada Ravenscar profile ORK+, RTEMS and Linux (x86 architectures) in paravirtualized mode.

We cover XtratuM in detail on the next chapter.

VMware ESX runs on bare metal. It includes its own kernel: A Linux kernel is started first, and is then used to load a variety of specialized virtualization components, including ESX, which is otherwise known as the *vmkernel* component. The Linux kernel is the primary virtual machine; it is invoked by the service console. At normal run-time, the vmkernel is running on the bare computer, and the Linux-based service console runs as the first virtual machine. The vmkernel is a microkernel with three interfaces: hardware, guest systems, and the service console (Console OS). The vmkernel handles CPU and memory directly, using scan-before-execution to intercept and fully virtualize special or privileged CPU instructions and a system resource allocation table to track allocated memory, but also support paravirtualization for select operating systems. Access to

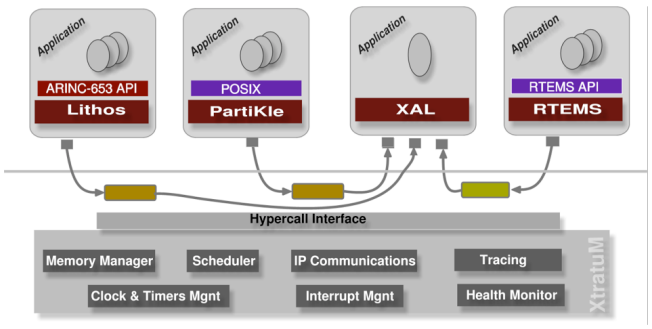


Figure 17: XtratuM running various execution environments.

other hardware (such as network or storage devices) takes place using modules. At least some of the modules derive from modules used in the Linux kernel. To access these modules, an additional module called *vmklinux* implements the Linux module interface. VMware ESXi, a smaller-footprint version of ESX, does not include the service console. It is available - without the need to purchase a vCenter license - as a free download from VMware, with some features disabled. VMware ESXi originated as a compact version of VMware ESX that allowed for a smaller 32 MB disk footprint on the host. With a simple configuration console for mostly network configuration and remote based VMware Infrastructure Client Interface, this allows for more resources to be dedicated to the guest environments.

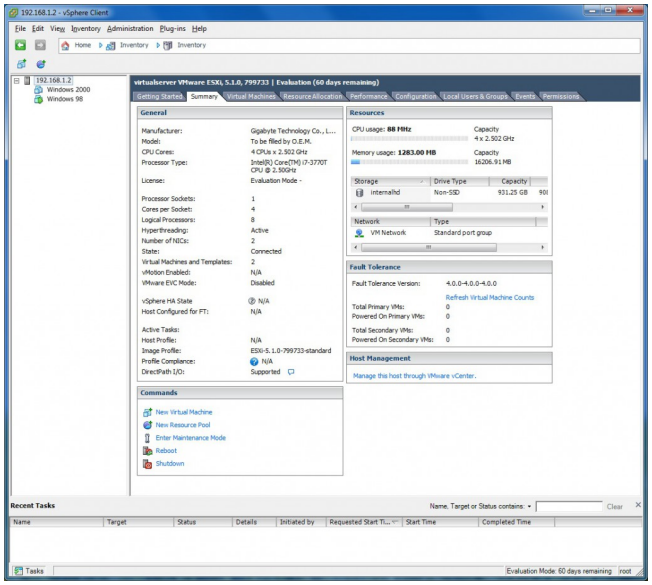


Figure 18: ESXi service console running two instances of Windows OS[20].

Two variations depending on the type and size of the target installation media of ESXi exist:

- VMware ESXi Installable, on a hard disk (or iSCSI/SAN/FCoE partition) that has a size of at least 5 GB
- VMware ESXi Embedded Edition, on a USB key drive, or SD card, or target media (no matter what type) smaller than 5 GB

Microsoft Hyper-V, codenamed Viridian and formerly known as Windows Server Virtualization, can create virtual machines on x86-64 systems running Windows. A server computer running Hyper-V can be configured to expose individual virtual machines to one or more networks. Hyper-V was first released alongside Windows Server 2008, and has been available without charge for all the Windows Server and some client operating systems since. A hypervisor instance has to have at least one parent partition, running a supported version of Windows Server (2008 and later).

The virtualization stack runs in the parent partition and has direct access to the hardware devices. The parent partition then creates the child partitions which host the guest OSs. A parent partition creates child partitions using the hypercall API, which is the application programming interface exposed by Hyper-V. A child partition does not have access to the physical processor, nor does it handle its real interrupts. Instead, it

[20] matthill.eu/

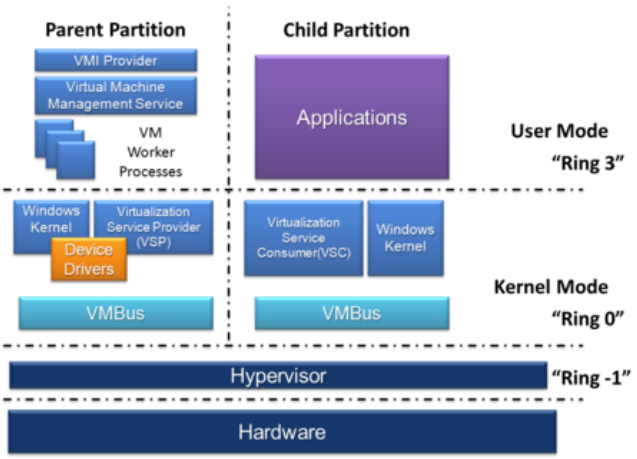


Figure 19: HyperV architecture supported by Intel's VT-x ring -1 hardware extension.

has a virtual view of the processor and runs in Guest Virtual Address, which, depending on the configuration of the hypervisor, might not necessarily be the entire virtual address space. Depending on VM configuration, Hyper-V may expose only a subset of the processors to each partition. The hypervisor handles the interrupts to the processor, and redirects them to the respective partition using a logical Synthetic Interrupt Controller (SynIC). Hyper-V can hardware accelerate the address translation of Guest Virtual Address-spaces by using second level address translation provided by the CPU, referred to as EPT on Intel and RVI on AMD. Child partitions do not have direct access to hardware resources, but instead have a virtual view of the resources, in terms of virtual devices. Any request to the virtual devices is redirected via the VMBus to the devices in the parent partition, which will manage the requests. The VMBus is a logical channel which enables inter-partition communication. The response is also redirected via the VMBus. If the devices in the parent partition are also virtual devices, it will be redirected further until it reaches the parent partition, where it will gain access to the physical devices. Parent partitions run a Virtualization Service Provider (VSP), which connects to the VMBus and handles device access requests from child partitions. Child partition virtual devices internally run a Virtualization Service Client (VSC), which redirect the request to VSPs in the parent partition via the VMBus. This entire process is transparent to the guest OS.

Bellow follows an aggregated table listing the hypervisors described, along with some non-mentioned. Types of virtualization supported and known use in embedded applications are noted.

Hypervisor	Virtualization			Embedded
	Full	Hardware	Para	
Xen Hypervisor	✓	✓	✓	✓
Proxmox VE	✓			
nuxis	✓	✓	✓	
L4[21]	✓	✓	✓	✓
XtratuM			✓	✓
VMware ESXi	✓	✓	✓	
MS Hyper-V	✓	✓		
Parallels Bare Metal	✓	✓		
Windriver Linux	✓	✓	✓	✓
LynxSecure	✓	✓	✓	✓
IBM PowerVM	✓			
Vembu VMBackup	✓	✓		

Table 1: Type-1 Hypervisors with virtulization and embedded categorization.

[21] L4Ka, L4/MIPS, Fiasco, OKL4, seL4, F9

3 The XtratuM Hypervisor

The term XtratuM derives from the word *stratum*. In geology and related fields it means:

> *Layer of rock or soil with internally consistent characteristics that distinguishes it from contiguous layers.*

In order to stress the tight relation with Linux and the open source the *"S"* was replaced by *"X"*. XtratuM would be the first layer of software (the one closer to the hardware), which provides a rock solid basis for the rest of the system. [Sol13]

XtratuM is a type-1, or *native* hypervisor, targeted for highly critical systems. The philosophy of the ARINC-653[22] software specification has been employed when applicable, even though it defines time partitioning systems, but not hypervisors explicitly. This influence is present in XtratuM's API and partition management.

XtratuM's partitions are treated as *virtual computers* that can run various operating systems, as long as their kernels are modified to support *para-virtualization*. At May 2016 the following execution environments and operating systems where supported, depending on the underlying hardware:

- XAL (XtratuM Abstraction Layer): a mininal layer to execute bare-C partitions
- PartiKle: a real-time operating system with an POSIX PSE52 API
- Lithos: a real-time operating system ARINC-653 compliant
- RTEMS: Real-Time Executive for Military Systems, an RTOS designed for embedded systems
- Linux: a Unix-like and mostly POSIX-compliant OS

XtratuM's licence is covered by the GPLv2[23].

3.1 Architecture

The main components of XtratuM's architecture are:

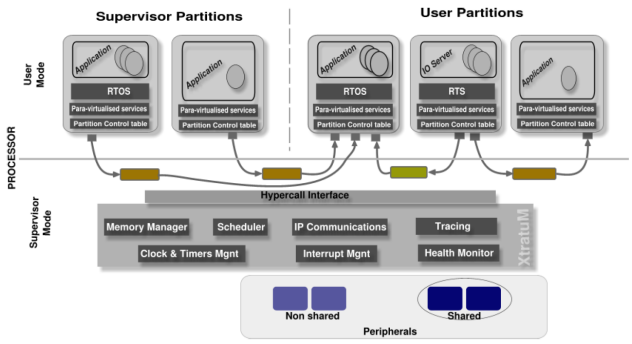


Figure 20: XtratuM architecture. [Sol13]

- Hypervisor: XtratuM provides 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 the following components:
  - Memory management: XtratuM provides a memory model for the partitions enforcing the spatial isolation. It uses the hardware mechanisms to guarantee the isolation.
  - Scheduling: Partitions are scheduled using a cyclic scheduling policy.
  - Interrupt management: Interrupts are handled by XtratuM and, depending on the interrupt nature, propagated to the partitions. XtratuM provides a interrupt model to the partitions that extends the concept of processor interrupts by adding a 32 additional interrupt numbers.
  - Clock and timer management.

[22] ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in safety-critical avionics real-time operating systems (RTOS). It allows the hosting of multiple applications of different software levels on the same hardware in the context of an Integrated Modular Avionics architecture. It is part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators. [inc03]

[23] http://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html

  - IP 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 defined 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.
  - 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 confine the faulting subsystem in order to avoid or reduce the possible consequences.
  - Tracing facilities: 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.

- API: Defines the para-virtualised services provided by XtratuM. The access to these services is provided through hypercalls.

- 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), that share 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; a real-time operating system (or runtime support) and its applications; or a general purpose operating system and its applications. Partitions need to be virtualised to be executed on top of a 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 the hardware must be aware of this fact.
  - 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. However, the operating system has to deal with the virtualisation and be virtualised (ported on top of XtratuM).

3.2 Developing process

XtratuM is a layer of software that extends the capabilities of the native hardware. There are important differences between a classical system and an hypervisor based one. The simplest scenario is composed of two actors: the integrator and a partition developer or partition supplier. There shall be only one integrator team and one or more partition developer teams.

The tasks to be done by the integrator are:

- Configure the XtratuM source code (jointly with the resident software). Customise it for the target board (processor model, etc.) and a miscellaneous set of code options and limits (debugging, identifiers length, etc).

- Build XtratuM: hypervisor binary, user libraries and tools.

- Distribute the resulting binaries to the partition developers. All partition developers shall use the same binary version of XtratuM.

- By creating the *XM_CF* configuration file, allocate the available system resources to the partitions, according to the resources required to execute each partition:

  - memory areas where each partition will be executed or can use,
  - design the scheduling plan,
  - communication ports between partitions,
  - the virtual devices and physical peripherals allocated to each partition,
  - configure the health monitoring,

- Gather the partition images and customisation files from partition developers.

- Pack all the files (resident software, XtratuM binary, partitions, and configuration files) into the final system image.

The partition developer activity:

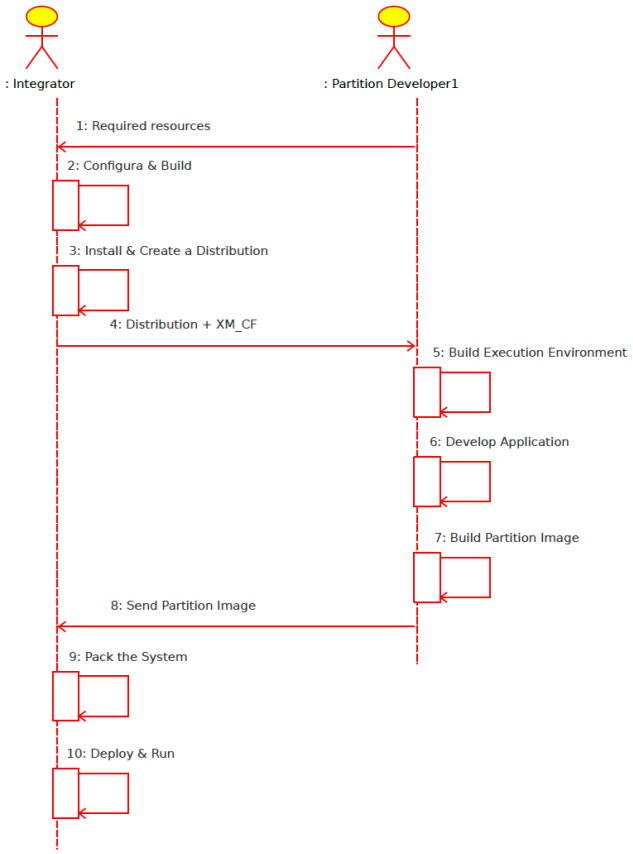


Figure 21: System integrator and partition developer cooperation. [Sol13]

- Define the resources required by its application, and send it to the integrator.
- Prepare the development environment. Install the binary distribution created by the integrator.
- Develop the partition application, according to the system resources agreed by the integrator.
- Deliver to the integrator the resulting partition image and the required customisation files (if any).

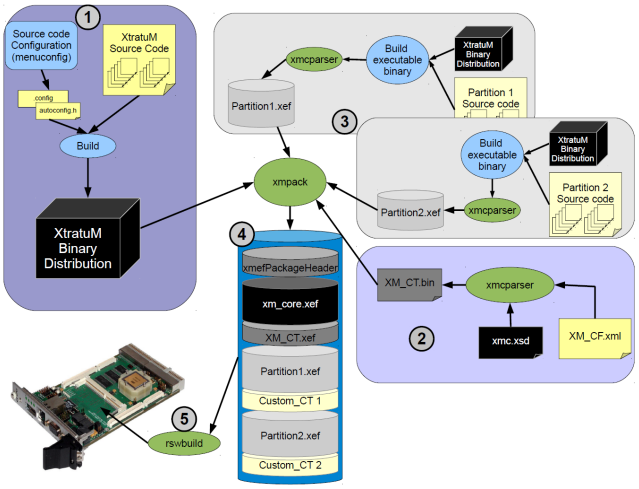


Figure 22: Buidling XtratuM. [Sol13]

There should be an agreement between the integrator and the partition developers on the resources allocated to each partition. The binaries, jointly with the *XM_CF* configuration file defines the partitioned system. All partition developers shall use exactly the same XtratuM binaries and configuration files during the development. Any change on the configuration shall be agreed with the integrator.

Since the development of the partitions may be carried out in parallel (or due to intellectual property restrictions), the binary image of some partitions may not be available to a partition developer team. In this case, it is advisable to use dummy partitions to replace those non-available, rather than changing the configuration file.

3.3 Configuration

The integrator, jointly with the partition developers, has to define the resources allocated to each partition, by creating the *XM_CF* file. The main information contained in the *XM_CF* file is:

- Memory: The amount of physical memory available in the board and the memory allocated to each partition.
- Processor: How the processor is allocated to each partition: the scheduling plan.
- 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 errors are managed by the partition and XtratuM: direct action, delivered to the offending partition, create a log entry, reset, etc.
- Inter-partition communication: The ports that each partition can use and the channels that link the source and destination ports.
- Tracing: Where to store trace messages and what messages shall be traced.

Since *XM_CF* 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* is parsed and translated into a binary format which can be directly used by XtratuM. The XML data is translated into a set of initialised data structures ready to be used by XtratuM. Otherwise, XtratuM will need to contain an XML parser to read the *XM_CF* information. The resulting configuration binary will be passed to XtratuM as a *customisation* file.

4 Performance Evaluation

In this chapter we evaluate the performance of the hypervisor by executing memory transfers and recording the delays. Due to technical constraints we were not able to generate comparable results on a non-hypervisor system. Instead, we used the single partition transfer delays as reference.

4.1 Obstacles and failures

This thesis was not absent of failures. While we put effort in producing results on specific topics, this was not always possible due to technical limitations, time constraints and unforeseeable events.

4.1.1 FPGA and LEON processor

The initial goal of this thesis was to evaluate the Xtratum hypervisor directly on an embedded platform. We chose the LEON[24] processor as it was available for VHDL synthesis. We were not aware that the publisher of the Xtratum hypervisor created a company, in order to monetize their work. While the x86 version was still freely available, the LEON version was no longer so. Unfortunately we had to discard the part of our work relevant to LEON's synthesis on a Xilinx FPGA[25].

4.1.2 Zybo Platform and Xen

After our previous failure an opportunity to compete in Xilinx Open Hardware 2015 competition arose. The ZYBO board[26] with the Zynq processor was the topic of the competition and the winners would have to present a system prototype, outlining the capabilities of the board. Our entry would be a ZYBO board with XEN hypervisor, demonstrating an automotive system. Unfortunately while there existed available material[27] for XEN on a Zynq Ultrascale[28], there weren't for the smaller Zynq-7000. Porting XEN for the Zynq-7000 proved to be a tedious procedure, which we abandoned to pursue research on Xtratum's free version.

4.1.3 Xtratum on bare metal

Our plans for evaluating Xtratum on x86, were to run the MiBench benchmarks[29] but this proved to be impossible due to XAL supporting only *stdio* and *string* C libraries. The benchmarks made use of other libraries and could not be compiled with the XAL resources.

While the SDK provides example systems with various combinations of partitions, there are no definitive instructions on makefile and xml schema creation. Many times partition generation failed or the final system was unresponsive. Some times there were clear indications as to what the fault was, but on other occasions we had to rely on trial and error. As it is, we believe that the free edition SDK is poorly documented for creating something that varies from the examples.

Moreover we were not able to compare the results directly on a x86 system, because Xtratum's *stdio* output did not provide any other options apart from "PcUart" in the xml file. No documentation on the output could be found, nor Xtratum's team could be reached via email. The only option in which we finally settled, was to virtualize the x86 system and use a Linux terminal as a redirected output.

4.2 Testing Environment

The structure of the testing environment was built like so:

- Intel i7 3770
- Windows 10 Pro v1607 (courtesy of University of Piraeus MSDNAA/Dreamspark Programme)
- Oracle VirtualBox v5.0.20
- Kubuntu/Ubuntu 12.04 LTS
- QEMU emulator v1.0
- XtratuM 2.6.0 GPL Edition
- PaRTiKle v2.0r0

[24] http://www.gaisler.com/index.php/products/processors/leon3
[25] https://www.xilinx.com/products/silicon-devices/fpga.html
[26] https://www.xilinx.com/products/boards-and-kits/1-4azfte.html
[27] http://dornerworks.com/services/xilinxxen
[28] https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html
[29] http://vhosts.eecs.umich.edu/mibench//source.html

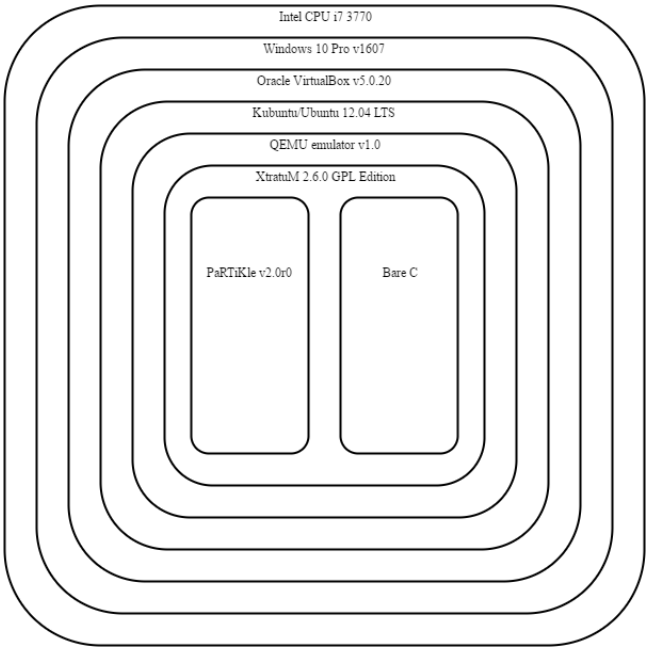


Figure 23: The testing environment system

4.2.1 Hardware specifications

The hardware of the system is based on an Intel i7 CPU, which supports the following features:

- *Intel Virtualization Technology (VT-x)* which is necessary to run virtualization with KVM/QEMU
- *Intel Virtualization Technology for Directed I/O (VT-d)* which makes direct access to a PCI device possible for guest systems with the help of the Input/Output Memory Management Unit (IOMMU) provided. This feature is not relevant with our evaluation but to virtualization in general.
- *Intel VT-x with Extended Page Tables (EPT)* provides a hardware assist to memory virtualization, which includes the partitioning and allocation of physical memory among VMs. This feature is not relevant with our evaluation but to virtualization in general.

4.2.2 Host OS and Virtual Machine

Windows 10 was the author's OS of choice due to the simplicity of use in multiple environments in academia, professional use and recreation. Oracle's VirtualBox type-2 hypervisor was chosen because of its availability in freeware, trust on the publisher's expertise and exploitation of the CPU's virtualization features.

4.2.3 QEMU

The QEMU type-2 hypervisor was chosen to host the Xtratum hypervisor, because in our experiments we did not manage to generate console or serial output of the performance tests in native environment. As such, we opted for QEMU redirection of Xtratum's system serial output, on Kubuntu's terminal window. The required linux command lines for this task are included in the appendix.

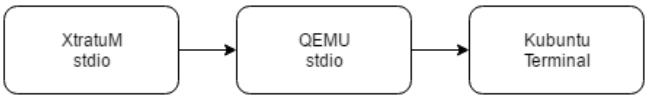


Figure 24: Flow of debugging messages during testing.

4.2.4 Bare Metal C

A Bare Metal C partition was selected as good reference of a partition that runs the minimum required code for our benchmarks.

4.2.5 Partikle OS

The Partikle OS was chosen for it's compatability on the Xtratum hypervisor and for the fact that it's a real time OS.

4.3 Setting up the Hypervisor and Partitions

The hypervisor files were generated following the instructions in the user manual's chapter 4 [Sol13]. The partitions required tweaking upon the existing examples provided in the SDK and in some cases significant effort, due to lack of more specific instructions. The relevant *makefiles* and *Xtratum xml schemas* can be found at the appendix of this document.

4.4 Data Transfers

Three methods of data transfers were tested:

- Shared memory using C library's *memcpy* function. This is the simplest type of memory transfer, akin to a *for loop* copying the data. This method was available for all test cases. The location of the shared memory space is pre-allocated when packing the systems in the *xml* file.
- Sampling messages, provided be Xtratum's XAL API. The *sender* partition uses the API to transmit data via a pre-allocated port. The data are available *sampling* by another partition, until overwritten by the *sender*. The *sender* and *sampler* ports are created when packing the system, by setting the relevant switches in the *xml* file.
- Queuing messages, provided be Xtratum's XAL API. The *sender* partition uses the API to transmit data via a pre-allocated port. The data are placed in a queue with predefined depth and can be accessed in a FIFO manner. The *sender* and *receiver* ports are created when packing the system, by setting the relevant switches in the *xml* file.

The measurements were performed using the XAL API's *XM_get_time*. This function returns the hypervisor's internal incremental clock in microsecond accuracy as a 64bit unsigned integer. The function was called before and after the transfer and the difference was used as a measure of the transfer's delay. The API's delay was not taken into account, due to our approach on result comparison being relative rather than absolute.

Each case was tested 16 times to measure the average delay and deviation among passes. All tests were written in the C language and the relevant files for each case are available in the appendix of this document.

	1 Partition Delay (us)		2 Partitions Delay (us)		
Pass #	Bare C	Partikle	Bare C	Partikle	Bare C / Partikle
1	42	33	36	33	19
2	44	35	21	26	21
3	42	31	38	22	19
4	45	35	21	23	20
5	41	34	21	26	20
6	42	29	21	24	21
7	42	32	19	23	20
8	47	34	17	23	20
9	43	33	19	22	19
10	40	34	21	42	20
11	43	33	45	23	29
12	45	32	21	24	20
13	40	34	20	23	21
14	43	36	27	39	20
15	41	34	21	22	21
16	43	32	20	22	20
Min	40	29	17	22	19
Average	42.6875	33.1875	24.25	26.0625	20.625
Max	47	36	45	42	29
Av. Deviation	1.4375	1.3125	6.125	4.4765625	1.234375

Table 2: Delays on shared memory *memcpy*.

	2 Partition - Bare C		
	Sampling		
Pass #	Send Delay (us)	Read Delay (us)	Total Delay (us)
1	164	184	348
2	160	169	329
3	100	186	286
4	161	171	332
5	135	58	193
6	171	305	476
7	182	177	359
8	143	147	290
9	224	248	472
10	159	145	304
11	148	153	301
12	150	320	470
13	277	199	476
14	176	220	396
15	177	170	347
16	201	190	391
Min	100	58	193
Average	170.5	190.125	360.625
Max	277	320	476
Av. Deviation	26.8125	42.671875	64.65625

Table 3: Delays on XAL port sampling.

	2 Partition - Bare C		
	Queuing		
Pass #	Send Delay (us)	Read Delay (us)	Total Delay (us)
1	335	491	826
2	275	381	656
3	297	264	561
4	301	322	623
5	235	375	610
6	271	594	865
7	277	274	551
8	237	211	448
9	276	774	1050
10	231	261	492
11	267	239	506
12	270	382	652
13	295	286	581
14	433	431	864
15	268	459	727
16	294	429	723
Min	231	211	448
Average	285.125	385.8125	670.9375
Max	433	774	1050
Av. Deviation	30.53125	107.890625	128.671875

Table 4: Delays on XAL port queuing.

4.4.1 Intra-partition shared memory

A single partition system was created and a 128 byte array was copied to another array using *memcpy*.

```
>> HwTimer [i8253 timer (1193Khz)]
[sched] using cyclic scheduler
1 Partition(s) created
P0 ("Partition1":0) cpu: 0 flags: [SV BOOT (0x2000000)]: [0x2000000 - 0x2100000]
[P0] Duration: 42us
[HYPERCALL] (0x0) Halted
```

Figure 25: Single partition Bare C intra-partition shared memory bash listing.

```
>> HwTimer [i8253 timer (1193Khz)]
[sched] using cyclic scheduler
1 Partition(s) created
P0 ("Partition1":0) cpu: 0 flags: [SV BOOT (0x800000)]: [0x800000 - 0x900000]
[P0] >> PaRTiKle Core (2.0 r0) <<
[P0] Detected 3401.306 MHz processor.
[P0] Setting up the dynamic memory manager (512 kbytes at 0x814d60)
[P0] Free system memory 768 Kb
[P0] PaRTiKle (853 Kb [.text=28 .data=2 .rodata=1 .bss=793] Kb)
[P0] App. (42 Kb [.text=26 .data=0 .rodata=6 .bss=9] Kb)
[P0]    - init console: ok
[P0]    - init root: ok
[P0]
[P0] Jumping to the user's code
[P0]
[P0] [P0] I'm t1: 0x859bdc
[P0] [P0] Duration: 33us
[P0]
[P0] System exit status: 0.
[P0]
[HYPERCALL] (0x0) Halted
```

Figure 26: Single partition Partikle intra-partition shared memory bash listing.

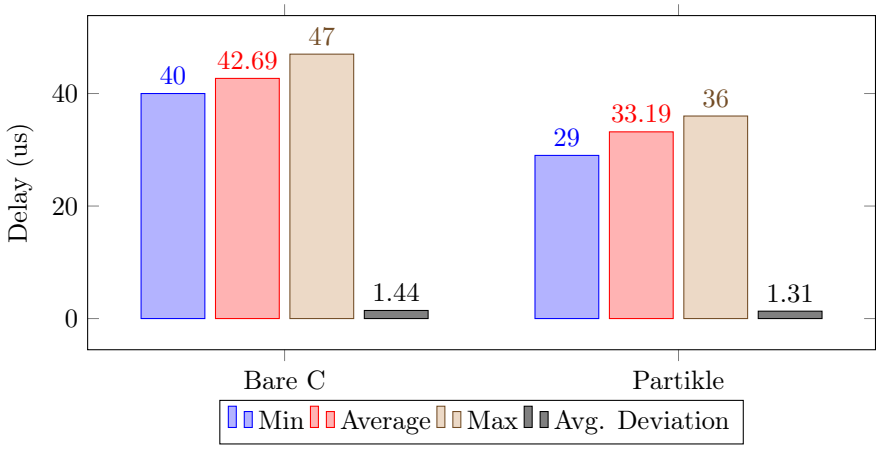


Figure 27: Single partition 128 byte *memcpy* transfers.

We can see that the Partikle partition was somewhat faster than the Bare C. While a definite conclusion as to the reason of this result cannot be drawn, due to Partikle's core and libraries being available as pre-compiled objects, it's safe to assume that the *memcpy* function has been optimized internally in Partikle's distribution.

This case was tested to serve as a reference for other cases.

4.4.2 Inter-partition shared memory

For the inter-partition transfers, we used *memcpy* again, with some added features. Each partition generates an 128 byte array, calculates the 8bit checksum of the array elements which is stored in shared space and sets a flag to report that the transfer is ready. Then continuously checks the other partition's completion flag, and

if it's been set, reads the array, calculates the 8bit checksum and compares it with the previously calculated checksum. Finally each partition generates a success or failure message, if the checksums are equal.

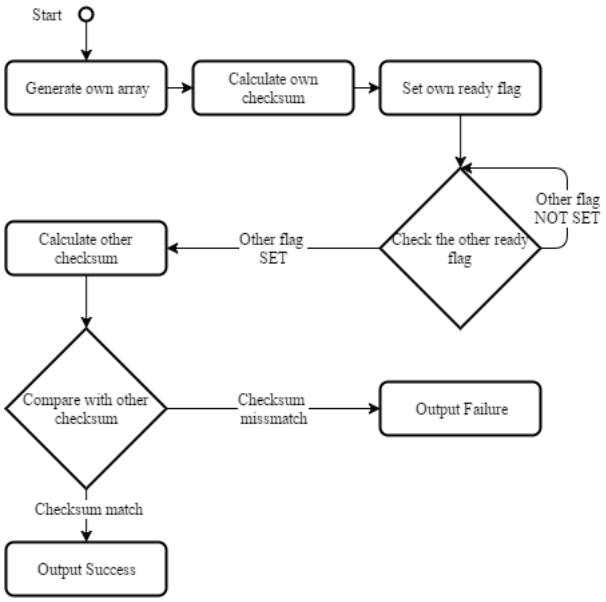


Figure 28: Finite state machine of inter-partition testing.

```
>> HwTimer [i8253 timer (1193Khz)]
[sched] using cyclic scheduler
2 Partition(s) created
P0 ("Partition1":0) cpu: 0 flags: [ BOOT (0x2000000) ]:
    [0x2000000 - 0x2100000]
    [0x2200000 - 0x2300000]
P1 ("Partition2":1) cpu: 0 flags: [ BOOT (0x2100000) ]:
    [0x2100000 - 0x2200000]
    [0x2200000 - 0x2300000]
[P0] Initialized shared memory
[P1] Initialized shared memory
[P0] Vector fill duration: 805us
[P0] Vector Checksum: 86
[P0] memcpy vector duration: 36us
[P1] Vector fill duration: 802us
[P1] Vector Checksum: 243
[P1] memcpy vector duration: 19us
[P1] Checksum updated
[P1] checksum calculation duration: 23us
[P1] calculated checksum: 86
[P1] shared checksum: 86
[P1] Checksum match
[HYPERCALL] (0x1) Halted
[P0] Checksum updated
[P0] checksum calculation duration: 24us
[P0] calculated checksum: 243
[P0] shared checksum: 243
[P0] Checksum match
[HYPERCALL] (0x0) Halted
```

Figure 29: Two partition Bare C shared memory bash listing.

All partitions performed almost equally, within a margin of 2.01us from their combined average or 8,52% on relative average deviation. It is worth to mention that while the mixed OS system performed best on average, this happens within a narrow statistical error margin. Even so, we can note that the use of mixed OSes does not incur any latency burden on memory transfers between partitions within shared memory.

4.4.3 Inter-partition messages

For the messaging test, only the Bare C environment was available, as the XAL API did not support messaging for Partikle. The queue depth was 16 messages deep. Message reading is occurring during an interrupt call,

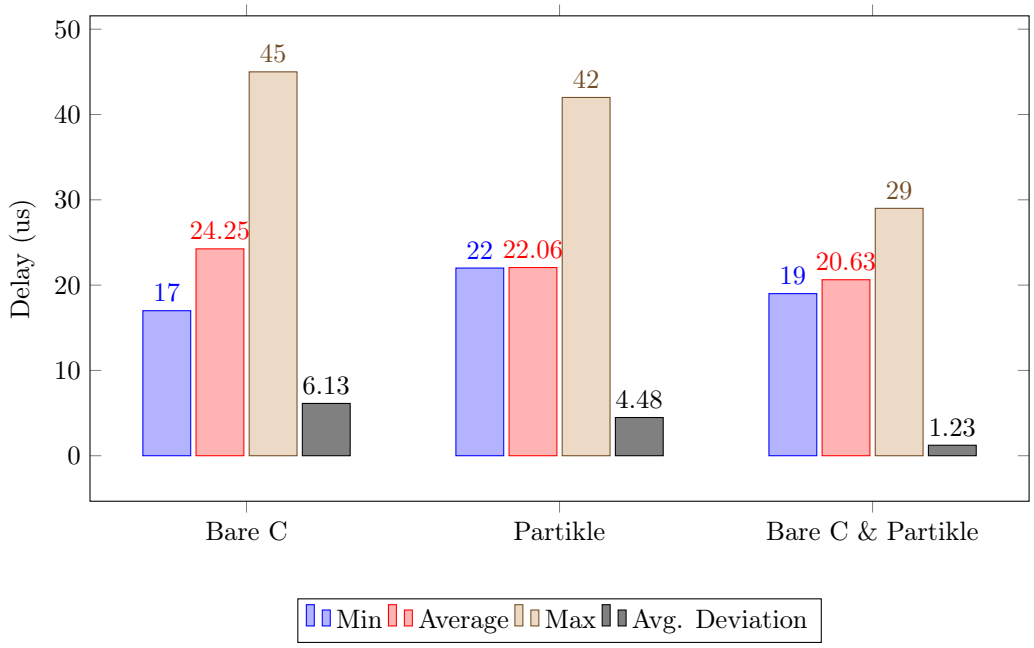


Figure 30: Two partition 128 byte *memcpy* transfers.

caused by the reception of the message. The relevant API hypercalls are the following:

- XM_write_sampling_message
- XM_read_sampling_message
- XM_send_queuing_message
- XM_receive_queuing_message

Internally the API calls the hypercalls *XM_write_object* and *XM_read_object*, which are not available to read in source code, so we can only assume the inner workings.

```
>> HwTimer [i8253 timer (1193Khz)]
[sched] using cyclic scheduler
2 Partition(s) created
P0 ("Partition1":0) cpu: 0 flags: [BOOT (0x2000000)]: [0x2000000 - 0x2100000]
P1 ("Partition2":1) cpu: 0 flags: [BOOT (0x2100000)]: [0x2100000 - 0x2200000]
[P0] Opening ports...
[P0] done
[P0] Generating messages...
[P0] SEND <<sampling message 0>>
[P0] Duration to send sampling message:  164us
[P1] Opening ports...
[P1] done
[P1] Waiting for messages
[P0] SEND <<queuing message 0>>
[P0] Duration to send queuing message:  335us
[P1] Duration to receive queuing message:  491us
[P1] RECEIVE <<queuing message 0>>
[P1] Duration to receive sampling message:  184us
[P1] RECEIVE <<sampling message 0>>
[P0] Done
[HYPERCALL] (0x0) Halted
```

Figure 31: Two partition Bare C sampling and queuing messaging bash listing.

We only measure the duration of writing (send) and reading (receive) in order to compare with the simple memory transfers of the previous tests. We then add the send and receive duration to compare it with the simple memory transfer, as with *memcpy* the data are readily available to the receiving partition on *memcopy*'s exit point.

As we do not have code access on the port messaging system, we can assume that the sampling procedure includes the following steps:

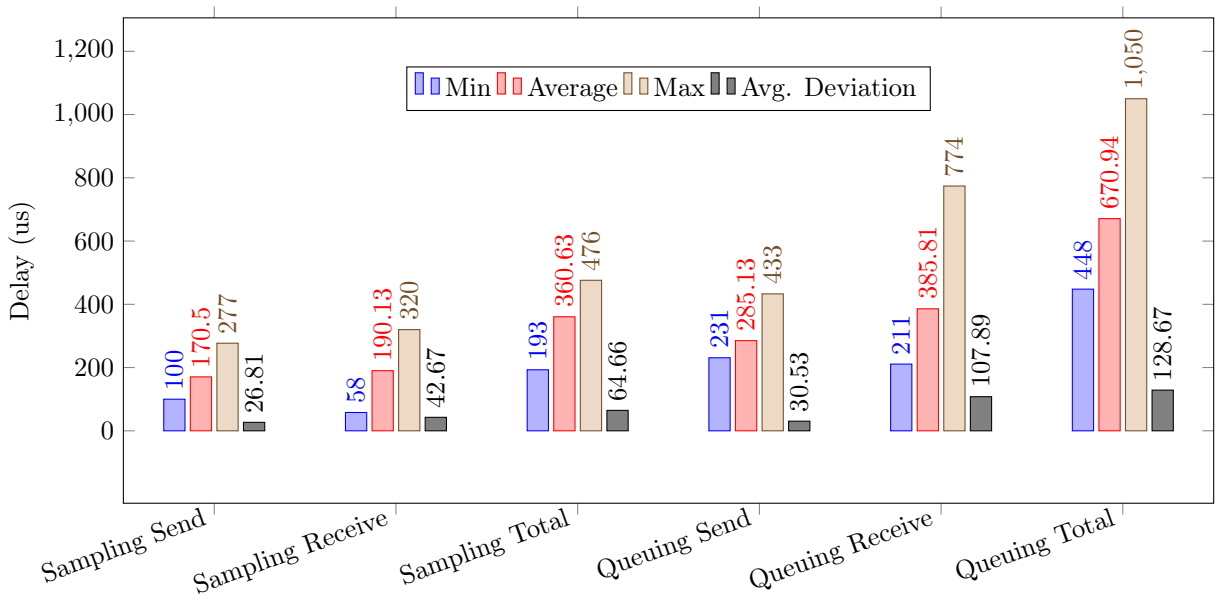


Figure 32: Two partition Bare C sampling and queuing 128 byte messaging.

- The sender is writing a message on common or hypervisor managed memory
- The hypervisor generates an interrupt to notify the recipient of the write
- The recipient enters the interrupt and can read the message from memory

Queuing should follow the steps:

- The sender is writing a message on common or hypervisor managed memory
- the hypervisor pushes the message at the bottom of the FIFO memory
- The hypervisor generates an interrupt to notify the recipient of the write
- The recipient enters the interrupt and can requests the message
- The hypervisor returns with the message at the top of the FIFO memory
- The hypervisor rearranges the FIFO or updates an internal indexing scheme or chain list

It is clear from the above that the queuing port shall be a slower method, since it accesses a FIFO than a memory location directly. This is confirmed from our results where the sampling mechanism is 53.75% faster on average.

While queuing is almost twice as slow, it can offer certain advantages over sampling when used correctly. One example can be reading from a partition that manages a sensor system. The sensor can transmit a number of measurements and the receiving partition can read them after the FIFO is full and use them for averaging or filtering purposes.

4.4.4 Aggregated Results

Here we present the total transaction delays for each type of transfer for comparison

As it is evident, a logarithmic scale was required to be used on the y axis, due to the magnitude of latency difference between shared memory and queue transfers. The API transfers dwarf the simple *memcpy* transfer in latency and this shows that the API has room for improvement.

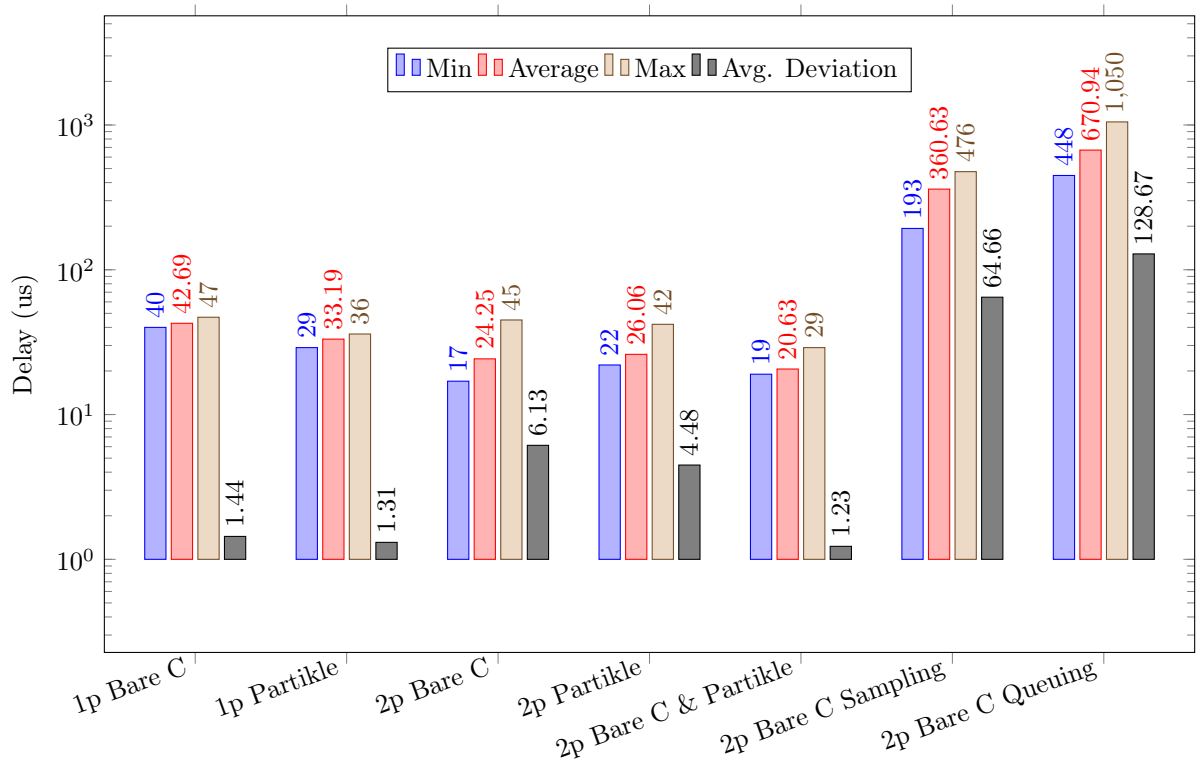


Figure 33: All test case delays for 128 byte transfers.

5 Summary and future work

5.1 Summary of this thesis

This thesis has described the use of virtualization in the form of hypervisors and detailed various methods of their usage in Chapters 1 and 2. In Chapter 3 the XtratuM hypervisor was introduced, along with its structure and building process. Finally, the results of the performance evaluation are presented in Chapter 4. New *makefiles* were written in order to build systems with two Partikle partitions and mixed Partikle - Bare C. Though many failures were encountered, we provided results that we aspire to be useful to future research on the subject of embedded systems virtualization and the Xtratum hypervisor.

5.2 Summary of evaluation

The XtratuM hypervisor's XAL API includes methods to perform inter-partitional memory transfers. While using the API is arguably more straightforward, caution must be used as it entails great latency, by at least an order of magnitude when compared to shared memory transfers.

We have shown that shared memory transfers can take 6.72% of the time that is needed for an equivalent sampling transfer on average, or 3.61% when using the API's queuing mechanism. Those parameters should be taken into account when designing a mission critical system, lest scheduling deadlines are missed with unpredictable consequences.

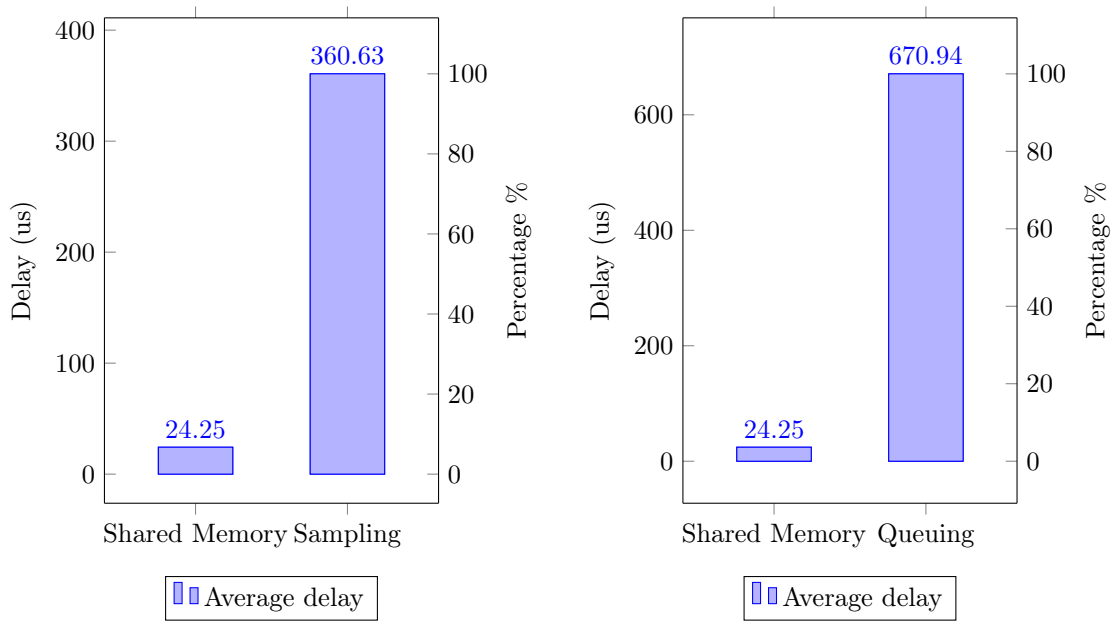
5.3 Suggestions for future work

As we were not able to perform the evaluation on a native x86 machine, the tests were run in a virtualized environment. While the results were compared in a relative manner, more accurate results could be acquired when run on the original hardware. This approach would also allow comparison of results from the hypervisor environment versus results from a native environment.

Moreover other types of partitions could be explored, such as Linux.

5.4 Conclusion

Hypervisors and virtualization can offer numerous advantages in mission critical and embedded in particular systems. While embedded hypervisors tend to be highly optimized for meeting critical schedules, their performance should not be overestimated. Memory transfers between hypervisor partitions are expected to be a



(a) Sampling versus shared memory transfer delay on average.

(b) Queuing versus shared memory transfer delay on average.

Figure 34: Percentage of average delay when using sampling and queuing transfers versus shared memory between Bare C partitions.

very common occurrence and as such, can create a bottleneck if not properly evaluated. It is the hypervisor designer's decision to choose between using mechanisms already provided or building new ones, to design a robust system.

References

[ONe67] Robert W. O'Neill. "Experience using a time-shared multi-programming system with dynamic address relocation hardware". In: *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*. 1967, pp. 611–621. DOI: 10.1145/1465482.1465581. URL: http://doi.acm.org/10.1145/1465482.1465581.

[Gol73] Robert P Goldberg. *Architectural principles for virtual computer systems*. Tech. rep. DTIC Document, 1973.

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073.

[Lie96] Jochen Liedtke. *L4 reference manual*. 1996.

[Sto96] Neil R. Storey. *Safety Critical Computer Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201427877.

[RI00] John Scott Robin and Cynthia E. Irvine. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor". In: *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*. SSYM'00. Denver, Colorado: USENIX Association, 2000, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251306.1251316.

[Bar+03] Paul Barham et al. "Xen and the Art of Virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: http://doi.acm.org/10.1145/1165389.945462.

[inc03] IHS inc. *ARINC 653, Avionics Application Software Standard Interface, 2003 Edition*. software specification W/D S/S BY ARINC 653 P1. Information Handling Services, Colorado, United States, Oct. 2003.

[Sch07] Anton Schedl. "Goals and Architecture of FlexRay at BMW". In: *Vector FlexRay Symposium Dr. Anton Schedl BMW Group 6th of March 2007*. FRS '07. Stuttgart, Germany, 2007. URL: http://vector.com/.

[Hei08] Gernot Heiser. "The Role of Virtualization in Embedded Systems". In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. Glasgow, Scotland: ACM, 2008, pp. 11–16. ISBN: 978-1-60558-126-2. DOI: 10.1145/1435458.1435461. URL: http://doi.acm.org/10.1145/1435458.1435461.

[For10] Johan Fornaeus. "Device Hypervisors". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 114–119. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837305. URL: http://doi.acm.org/10.1145/1837274.1837305.

[NDB10] Nicolas Navet, Bertrand Delord, and Markus Baumeister. "Virtualization in Automotive Embedded Systems : an Outlook". In: *Talk at RTS Embedded Systems 2010 Paris, 31/03/2010*. RTS '10. Paris, France, 2010. URL: http://nicolas.navet.eu/publications.html.

[Cor11] Radisys Corporation. *Leveraging Virtualization in Aerospace & Defense Applications*. Tech. rep. 5435 NE Dawson Creek Drive, Hillsboro, OR 97124 USA: Radisys Corporation, 2011. URL: www.radisys.com.

[Wik11] Scsami (Own work) [CC0] Wikimedia Commons. *The two types of hypervisors for virtualization*. [Online; accessed 17-April-2016]. 2011. URL: https://commons.wikimedia.org/wiki/File%3AHyperviseur.png.

[Hua12] Jim Huang. *Embedded Virtualization for Mobile Devices*. Tech. rep. 0xlab.org/, 2012.

[Maz16] Manolis Mazarakis. *Virtualization for Embedded Systems Lecture for the Embedded Systems Course CSD*. Tech. rep. University of Crete, 2016.

[Sol13] Fent Innovative Software Solutions. *XtratuM Hypervisor for x86: XtratuM User Manual*. Tech. rep. fnts-xm-um-41b. Fent Innovative Software Solutions, July, 2013.

Appendices

A Useful Linux Bash Commands

In the following section we list some useful bash command to ease XtratuM's installation and building.

A.1 Prerequisites installation in one command

Restart system after installing prerequisites.

```
> apt-get install gcc-4.4 libncurses5-dev gcc-4.8-multilib libxml2-dev grub2
  xorriso perl makeself qemu build-essential checkinstall cdbs devscripts
```

A.2 XtratuM hypervisor files setup

Navigate to *xm-src_v2.6* directory and prepare the hypervisor files:

```
> make distclean
> cp xmconfig.ia32 xmconfig
```

Find where gcc is and add its path to *xmconfig* file, *TARGET CCPREFIX entry*.

```
> which gcc
```

Add XtratuM's path to *xmconfig* file, *XTRATUM PATH entry*. Prepare the installation script by using the menu configuration tool. Leave defaults and save changes on all three occasions.

```
> make menuconfig
```

Make the hypervisor.

```
> make
```

Create a compressed tarball or an executable.

```
> make distro-tar
> make distro-run
```

A.3 Partition Building

Navigate to the *makefile* directory. Run each command separate when debugging build errors.

```
> make clean
> make
> make resident_sw.iso
> qemu-system-i386 -m 512 -serial stdio -hda resident_sw.iso
```

Or build and run the system with one line

```
> make clean;make;make resident_sw.iso;
  qemu-system-i386 -m 512 -serial stdio -hda resident_sw.iso
```

B XtratuM XML Schema files

In the following section we list each test case's schema file. Some of the following files are the original files provided by the SDK.

B.1 1 Partition - Bare C

```
<?xml version="1.0"?>
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    hello_world">
  <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
        <PhysicalMemoryAreas>
            <Area start="0x100000" size="3MB" />
        </PhysicalMemoryAreas>
        <HwDescription>
```

```
            <Processor id="0">
                <Sched>
                    <CyclicPlanTable>
                        <Plan id="0" majorFrame="200ms">
                            <Slot id="0" start="0ms" duration="200ms"
                                partitionId="0" />
                        </Plan>
                    </CyclicPlanTable>
                </Sched>
            </Processor>
            <MemoryLayout>
                <Region type="ram" start="0x0" size="128MB" />
            </MemoryLayout>
        </HwDescription>
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="Partition1" processor="0" flags="boot sv"
            loadPhysAddr="0x2000000" headerOffset="0x0" imageId="0x0"
             console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2000000" size="1MB" flags="mapped write" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>

    <Devices>
        <PcUart name="PcUart" />
    </Devices>
</SystemDescription>
```

B.2 1 Partition - Partikle

```
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    hello_world">
  <XMHypervisor console="PcUart" loadPhysAddr="0x100000">

    <PhysicalMemoryAreas>
      <Area start="0x100000" size="3MB" />
    </PhysicalMemoryAreas>

    <HwDescription>
      <Processor id="0">
                <Sched>
                <CyclicPlanTable>
                        <Plan id="0" majorFrame="1s">
                                <Slot id="0" start="0ms" duration="500ms"
                                    partitionId="0"/>
                        </Plan>
                </CyclicPlanTable>
                </Sched>
      </Processor>
      <MemoryLayout>
                <Region type="ram" start="0x0" size="32MB"/>
      </MemoryLayout>
    </HwDescription>
  </XMHypervisor>

  <PartitionTable>
    <Partition id="0" name="Partition1" processor="0"
                loadPhysAddr="0x800000" headerOffset="0x0" imageId="0x0" console
```

```
                    ="PcUart" flags="sv boot">
      <PhysicalMemoryAreas>
                <Area start="0x800000" size="1MB" flags="mapped write" />
      </PhysicalMemoryAreas>
    </Partition>
  </PartitionTable>
  <Devices>
    <PcUart name="PcUart" />
  </Devices>
</SystemDescription>
```

B.3 2 Partitions - Bare C - Shared Memory

```
<?xml version="1.0"?>
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    channels">
    <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
        <PhysicalMemoryAreas>
            <Area start="0x100000" size="3MB" />
        </PhysicalMemoryAreas>
        <HwDescription>
            <Processor id="0">
                <Sched>
                    <CyclicPlanTable>
                        <Plan id="0" majorFrame="1000ms">
                        <Slot id="0" start="0ms" duration="500ms" partitionId="0
                            " />
                        <Slot id="1" start="500ms" duration="500ms" partitionId=
                            "1" />
                    </Plan>
                    </CyclicPlanTable>
                </Sched>
            </Processor>
            <MemoryLayout>
                <Region type="ram" start="0x0" size="128MB" />
            </MemoryLayout>
        </HwDescription>
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="Partition1" processor="0" flags="boot"
           loadPhysAddr="0x2000000" headerOffset="0x0" imageId="0x0"
            console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2000000" size="1MB" flags="mapped write" />
                 <Area start="0x2200000" size="1MB" flags="mapped write shared"/
                    >
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="Partition2" processor="0" flags="boot"
           loadPhysAddr="0x2100000" headerOffset="0x0" imageId="0x1"
            console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2100000" size="1MB" flags="mapped write" />
                <Area start="0x2200000" size="1MB" flags="mapped write shared"/>
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>

    <Devices>
        <PcUart name="PcUart" />
```

```
    </Devices>

</SystemDescription>
```

B.4 2 Partitions - Partikle - Shared Memory

```
<?xml version="1.0"?>
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    channels">
    <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
        <PhysicalMemoryAreas>
            <Area start="0x100000" size="3MB" />
        </PhysicalMemoryAreas>
        <HwDescription>
            <Processor id="0">
                <Sched>
                    <CyclicPlanTable>
                        <Plan id="0" majorFrame="1000ms">
                        <Slot id="0" start="0ms" duration="500ms" partitionId="0
                            " />
                        <Slot id="1" start="500ms" duration="500ms" partitionId=
                            "1" />
                    </Plan>
                    </CyclicPlanTable>
                </Sched>
            </Processor>
            <MemoryLayout>
                <Region type="ram" start="0x0" size="128MB" />
            </MemoryLayout>
        </HwDescription>
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="Partition1" processor="0" flags="boot"
           loadPhysAddr="0x800000" headerOffset="0x0" imageId="0x0"
            console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x800000" size="1MB" flags="mapped␣write" />
                 <Area start="0x1000000" size="1MB" flags="mapped␣write␣shared"/
                    >
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="Partition2" processor="0" flags="boot"
           loadPhysAddr="0x900000" headerOffset="0x0" imageId="0x1"
            console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x900000" size="1MB" flags="mapped␣write" />
                 <Area start="0x1000000" size="1MB" flags="mapped␣write␣shared"/
                    >
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>

    <Devices>
        <PcUart name="PcUart" />
    </Devices>

</SystemDescription>
```

B.5 2 Partitions - Mixed Bare C and Partikle - Shared Memory

```
<?xml version="1.0"?>
```

```
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    channels">
    <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
        <PhysicalMemoryAreas>
            <Area start="0x100000" size="3MB" />
        </PhysicalMemoryAreas>
        <HwDescription>
            <Processor id="0">
                <Sched>
                    <CyclicPlanTable>
                        <Plan id="0" majorFrame="1000ms">
                        <Slot id="0" start="0ms" duration="500ms" partitionId="0
                            " />
                        <Slot id="1" start="500ms" duration="500ms" partitionId=
                            "1" />
                    </Plan>
                    </CyclicPlanTable>
                </Sched>
            </Processor>
            <MemoryLayout>
                <Region type="ram" start="0x0" size="128MB" />
            </MemoryLayout>
        </HwDescription>
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="Partition1" processor="0" flags="boot"
            loadPhysAddr="0x2000000" headerOffset="0x0" imageId="0x0"
             console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2000000" size="1MB" flags="mapped write" />
                 <Area start="0x2200000" size="1MB" flags="mapped write shared"/
                    >
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="Partition2" processor="0" flags="boot"
            loadPhysAddr="0x2100000" headerOffset="0x0" imageId="0x1"
             console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2100000" size="1MB" flags="mapped write" />
                <Area start="0x2200000" size="1MB" flags="mapped write shared"/>
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>

    <Devices>
        <PcUart name="PcUart" />
    </Devices>

</SystemDescription>
```

B.6 2 Partitions - Bare C - Sampling and Queuing Messaging

```
<?xml version="1.0"?>
<SystemDescription xmlns="http://www.xtratum.org/xm-2.3" version="1.0.0" name="
    channels">
    <XMHypervisor console="PcUart" loadPhysAddr="0x100000">
        <PhysicalMemoryAreas>
            <Area start="0x100000" size="3MB" />
        </PhysicalMemoryAreas>
        <HwDescription>
```

```
            <Processor id="0">
                <Sched>
                    <CyclicPlanTable>
                        <Plan id="0" majorFrame="1000ms">
                        <Slot id="0" start="0ms" duration="500ms" partitionId="0
                            " />
                        <Slot id="1" start="500ms" duration="500ms" partitionId=
                            "1" />
                    </Plan>
                    </CyclicPlanTable>
                </Sched>
            </Processor>
            <MemoryLayout>
                <Region type="ram" start="0x0" size="128MB" />
            </MemoryLayout>
        </HwDescription>
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="Partition1" processor="0" flags="boot"
            loadPhysAddr="0x2000000" headerOffset="0x0" imageId="0x0"
             console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2000000" size="1MB" flags="mapped write" />
            </PhysicalMemoryAreas>
            <PortTable>
                <Port type="queuing" direction="source" name="portQ"/>
                <Port type="sampling" direction="source" name="portS"/>
            </PortTable>
        </Partition>
        <Partition id="1" name="Partition2" processor="0" flags="boot"
            loadPhysAddr="0x2100000" headerOffset="0x0" imageId="0x1"
             console="PcUart">
            <PhysicalMemoryAreas>
                <Area start="0x2100000" size="1MB" flags="mapped write" />
            </PhysicalMemoryAreas>
            <PortTable>
                <Port type="sampling" direction="destination" name="portS"/>
                <Port type="queuing" direction="destination" name="portQ"/>
            </PortTable>
        </Partition>
    </PartitionTable>

    <Devices>
        <PcUart name="PcUart" />
    </Devices>

    <Channels>
        <QueuingChannel maxNoMessages="16" maxMessageLength="128B">
            <Source partitionId="0" portName="portQ"/>
            <Destination partitionId="1" portName="portQ"/>
        </QueuingChannel>
        <SamplingChannel maxMessageLength="128B">
            <Source partitionId="0" portName="portS"/>
            <Destination partitionId="1" portName="portS"/>
        </SamplingChannel>
    </Channels>
</SystemDescription>
```

C Makefiles

In the following section we list each test case's *makefile*. Some of the following *makefiles* are the original *makefiles* provided by the SDK.

C.1 1 Partition - Bare C

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=/opt/xm-sdk-x86//xal

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.ia32.xml

# PARTITIONS: partition files (xef format) composing the example
PARTITIONS=partition.xef

all: container.bin resident_sw
include $(XAL_PATH)/common/rules.mk

partition.xef: partition.o
	$(TARGET_LD) -o partition $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
	    ,1,$(XMLCF))
	@$(XEF) partition -o $@ -i 0

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
	-b partition.xef \

container.bin: $(PARTITIONS) xm_cf.bin.xmc
	$(XMPACK) build $(PACK_ARGS) $@
```

C.2 1 Partition - Partikle

```
ifndef COMMON_DIR
COMMON_DIR=$(CURDIR)/../common
endif

# PRTK: path to the Partikle distribution
PRTK_PATH=/opt/prtk-sdk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SOURCES=$(wildcard *.c)
OBJECTS=$(SOURCES:.c=.o)
PARTITIONS=$(SOURCES:.c=.xef)

POC_COMMON_FLAGS = -I$(COMMON_DIR)/lib -I$(COMMON_DIR)/configurations
CFLAGS += $(CFLAGS_ARCH) -O2 $(POC_COMMON_FLAGS)

all: resident_sw
include $(PRTK_PATH)/prtkconfig
include $(PRTK_PATH)/lib/rules.mk

${PARTITIONS:.xef=}: $(OBJECTS)
	$(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 0 $(XMLCF))

$(PARTITIONS): ${PARTITIONS:.xef=}
	@$(XEF) $^ -o $@ -i 0

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
	-b $(PARTITIONS)
```

```
container.bin: $(PARTITIONS) xm_cf.xef.xmc
	$(XMPACK) build $(PACK_ARGS) $@
```

C.3 2 Partitions - Bare C - Shared Memory

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=/opt/xm-sdk-x86//xal

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.ia32.xml

# PARTITIONS: partition files (xef format) composing the example
PARTITIONS=partition0.xef partition1.xef

all: container.bin resident_sw
include $(XAL_PATH)/common/rules.mk

partition0.xef: partition0.o
	$(TARGET_LD) -o partition0 $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
	    ,1,$(XMLCF))
	@$(XEF) partition0 -o $@ -i 0

partition1.xef: partition1.o
	$(TARGET_LD) -o partition1 $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
	    ,2,$(XMLCF))
	@$(XEF) partition1 -o $@ -i 1

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
	-b partition0.xef \
	-b partition1.xef \

container.bin: $(PARTITIONS) xm_cf.bin.xmc
	$(XMPACK) build $(PACK_ARGS) $@
```

C.4 2 Partitions - Partikle - Shared Memory

Note that in order to build this system we need to have each partition in its own directory, with its own *makefile*. The we call an external *makefile* to combine the partition files.

Partition 0 :

```
ifndef COMMON_DIR
COMMON_DIR=$(CURDIR)/../common
endif

# PRTK: path to the Partikle distribution
PRTK_PATH=/opt/prtk-sdk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SOURCES=$(wildcard *.c)
OBJECTS=$(SOURCES:.c=.o)
PARTITIONS=$(SOURCES:.c=.xef)

POC_COMMON_FLAGS = -I$(COMMON_DIR)/lib -I$(COMMON_DIR)/configurations
CFLAGS += $(CFLAGS_ARCH) -O2 $(POC_COMMON_FLAGS)

all: resident_sw
include $(PRTK_PATH)/prtkconfig
include $(PRTK_PATH)/lib/rules.mk
```

```
${PARTITIONS:.xef=}: $(OBJECTS)
	$(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 0 $(XMLCF))

$(PARTITIONS): ${PARTITIONS:.xef=}
	@$(XEF) $^ -o $@ -i 0

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
	-b $(PARTITIONS)

container.bin: $(PARTITIONS) xm_cf.xef.xmc
	$(XMPACK) build $(PACK_ARGS) $@
```

Partition 1 :

```
ifndef COMMON_DIR
COMMON_DIR=$(CURDIR)/../common
endif

# PRTK: path to the Partikle distribution
PRTK_PATH=/opt/prtk-sdk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SOURCES=$(wildcard *.c)
OBJECTS=$(SOURCES:.c=.o)
PARTITIONS=$(SOURCES:.c=.xef)

POC_COMMON_FLAGS = -I$(COMMON_DIR)/lib -I$(COMMON_DIR)/configurations
CFLAGS += $(CFLAGS_ARCH) -O2 $(POC_COMMON_FLAGS)

all: resident_sw
include $(PRTK_PATH)/prtkconfig
include $(PRTK_PATH)/lib/rules.mk

${PARTITIONS:.xef=}: $(OBJECTS)
	$(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 0 $(XMLCF))

$(PARTITIONS): ${PARTITIONS:.xef=}
	@$(XEF) $^ -o $@ -i 0

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
	-b $(PARTITIONS)

container.bin: $(PARTITIONS) xm_cf.xef.xmc
	$(XMPACK) build $(PACK_ARGS) $@
```

Final System :

```
ifndef COMMON_DIR
COMMON_DIR=$(CURDIR)/../common
endif

# PRTK: path to the Partikle distribution
PRTK_PATH=/opt/prtk-sdk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml
```

```
POC_COMMON_FLAGS = -I$(COMMON_DIR)/lib -I$(COMMON_DIR)/configurations -I$(
    PRTK_PATH)/include
CFLAGS += $(CFLAGS_ARCH) -O2 $(POC_COMMON_FLAGS)

all: resident_sw
include $(PRTK_PATH)/prtkconfig
include $(PRTK_PATH)/lib/rules.mk

p0/pthread_0: p0/pthread_0.o
        $(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 0 $(XMLCF))
p1/pthread_1: p1/pthread_1.o
        $(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 1 $(XMLCF))

p0/pthread_0.xef: p0/pthread_0
        @$(XEF) $^ -o $@ -i 0
p1/pthread_1.xef: p1/pthread_1
        @$(XEF) $^ -o $@ -i 1

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
        -b p0/pthread_0.xef\
        -b p1/pthread_1.xef\

container.bin: p0/pthread_0.xef p1/pthread_1.xef xm_cf.xef.xmc
        $(XMPACK) build $(PACK_ARGS) $@
```

C.5 2 Partitions - Mixed Bare C and Partikle - Shared Memory

Again, two extra *makefiles* for each partition are needed.

Partition 0 :

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=/opt/xm-sdk-x86//xal

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.ia32.xml

# PARTITIONS: partition files (xef format) composing the example
PARTITIONS=partition.xef

all: partition.xef
include $(XAL_PATH)/common/rules.mk

partition.xef: partition.o
        $(TARGET_LD) -o partition $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
            ,1,$(XMLCF))
        @$(XEF) partition -o $@ -i 0
```

Partition 1 :

```
ifndef COMMON_DIR
COMMON_DIR=$(CURDIR)/../common
endif

# PRTK: path to the Partikle distribution
PRTK_PATH=/opt/prtk-sdk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml
```

```
POC_COMMON_FLAGS = -I$(COMMON_DIR)/lib -I$(COMMON_DIR)/configurations -I$(
    PRTK_PATH)/include
CFLAGS += $(CFLAGS_ARCH) -O2 $(POC_COMMON_FLAGS)

all: pthread_1.xef
include $(PRTK_PATH)/prtkconfig
include $(PRTK_PATH)/lib/rules.mk

pthread_1: pthread_1.o
        $(LD) -o $@ $^ -Dstart=$(shell $(XPATHSTART) 1 $(XMLCF))

pthread_1.xef: pthread_1
        @$(XEF) $^ -o $@ -i 1
```

Final System :

```
export COMMON_PATH=$(CURDIR)/common
export POC_BUILD=RUN_OVER_XTRATUM

COMMON_DIR=$(COMMON_PATH)

#all: par_xefs resident_sw
all: container.bin resident_sw
include $(COMMON_PATH)/xef_rules.mk

DIRS=\
        p0\
        p1

CLEANDIRS=\
        p0\
        p1\
        common/lib

# partitions in xef format that will be packed to form the final executable
PARTITIONS=partition.xef pthread_1.xef


partition.xef:
        cd p0; make $(MAKECMDGOALS)

pthread_1.xef:
        cd p1; make $(MAKECMDGOALS)

PACK_ARGS=-h $(XMCORE):xm_cf.bin.xmc \
        -b p0/partition.xef\
        -b p1/pthread_1.xef

#par_xefs:
#       @for dir in $(DIRS) ; do \
#               (cd $$dir; make $(MAKECMDGOALS)); \
#       done

container.bin: $(PARTITIONS) xm_cf.bin.xmc
        $(XMPACK) build $(PACK_ARGS) $@

resident_sw: container.bin
        $(RSWBUILD) $^ $@

#**WARNING** manually run, make needs bugfix
resident_sw.iso: resident_sw
```

```
        /opt/xm-sdk-x86/xal/bin/grub_iso $@ $^

clean:
        @find -name "*.o" -exec rm '{}' \;
        @find -name "*.xef" -exec rm '{}' \;
        @find -name "*~" -exec rm '{}' \;
        @find -name "*.xmc" -exec rm '{}' \;
        @find -name "*.bin" -exec rm '{}' \;
        @find -name "resident_sw" -exec rm '{}' \;

distclean: clean
        @for dir in $(CLEANDIRS) ; do \
                (cd $$dir; make $(MAKECMDGOALS)); \
        done
```

C.6 2 Partitions - Bare C - Sampling and Queuing Messaging

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=/opt/xm-sdk-x86//xal

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.ia32.xml

# PARTITIONS: partition files (xef format) composing the example
PARTITIONS=partition0.xef partition1.xef

all: container.bin resident_sw
include $(XAL_PATH)/common/rules.mk

partition0.xef: partition0.o
        $(TARGET_LD) -o partition0 $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
            ,1,$(XMLCF))
        @$(XEF) partition0 -o $@ -i 0

partition1.xef: partition1.o
        $(TARGET_LD) -o partition1 $^ $(TARGET_LDFLAGS) -Ttext=$(call xpathstart
            ,2,$(XMLCF))
        @$(XEF) partition1 -o $@ -i 1

PACK_ARGS=-h $(XMCORE_BIN):xm_cf.bin.xmc \
        -b partition0.xef \
        -b partition1.xef \

container.bin: $(PARTITIONS) xm_cf.bin.xmc
        $(XMPACK) build $(PACK_ARGS) $@
```

D Source files

The performance tests were written in the C language. All code for each test case can be found below.

D.1 1 Partition - Bare C

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <xm.h>

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
```

```
} while (0)

char charArrayTx[128];
char charArrayRx[128];

void PartitionMain(void) {

  xmTime_t start, end, duration;

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  memcpy (charArrayRx, charArrayTx, sizeof(charArrayRx));
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("Duration:  %lluus\n", duration);

  XM_halt_partition(XM_PARTITION_SELF);
}
```

D.2 1 Partition - Partikle

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>

#include <xm.h>

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

pthread_t t1;
char charArrayTx[128];
char charArrayRx[128];

void *f(void *args) {
  struct timespec t = {2, 0};
  xmTime_t start, end, duration;
  xm_u32_t index;

  PRINT ("I'm t1: %p\n", pthread_self ());
  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  memcpy (charArrayRx, charArrayTx, sizeof(charArrayRx));
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("Duration:  %lluus\n", duration);
  nanosleep (&t, 0);
  return (void *)32;
}

int main (int argc, char **argv) {
  void *ex = 0;
  pthread_create (&t1, 0, f, 0);
```

```
  pthread_join (t1, &ex);
  return 0;
}
```

D.3 2 Partitions - Bare C - Shared Memory

Shared memory header file :

```
#define VECTOR_LEN        128U

typedef struct vecWithChecksumStruct
{
  unsigned char vector[VECTOR_LEN];
  unsigned char checksum;
  unsigned char updated;
} vecWChck_t;

typedef struct memShareStruct
{
  vecWChck_t shareA;
  vecWChck_t shareB;
} memShare_t;
```

D.4 2 Partitions - Bare C - Shared Memory

Partition 0 :

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <xm.h>
#include <irqs.h>

#include "shared.h"

#define SHARED_ADDRESS  0x2200000

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorA;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;

  vec->checksum = 0;
  while (index < len)
  {
    XM_get_time(XM_HW_CLOCK, &val);
    memcpy (&(vec->vector[index]), &val, 1U);
    vec->checksum += vec->vector[index++];
  }
  vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
```

```
    unsigned char checksum = 0;

    while (index < len)
    {
        checksum += vec->vector[index++];
    }
    return (checksum);
}

void PartitionMain(void)
{
    xmTime_t start, end, duration;
    unsigned char sharedChecksum;

    // Initialize shared mem
    memset (sharedMem, 0x00, sizeof (memShare_t));
    PRINT("Initialized shared memory\n");
    XM_idle_self();

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    VectorFill (&vectorA, VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("Vector fill duration:  %lluus\n", duration);
    PRINT("Vector Checksum:  %u\n", vectorA.checksum);

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    memcpy (&(sharedMem->shareA), &vectorA, sizeof(vectorA));
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("memcpy vector duration:  %lluus\n", duration);

    // Wait for the other partition to update its vector
    while (!sharedMem->shareB.updated)
    {
        XM_idle_self();
    }
    PRINT("Checksum updated\n");

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    sharedChecksum = VectorCalcChecksum (&(sharedMem->shareB), VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("checksum calculation duration:  %lluus\n", duration);
    PRINT("calculated checksum:  %u\n", sharedChecksum);
    PRINT("shared checksum:  %u\n", sharedMem->shareB.checksum);
    if (sharedChecksum == sharedMem->shareB.checksum)
    {
        PRINT("Checksum match\n");
```

```
  }
  else
  {
      PRINT("Checksum mismatch\n");
  }

  XM_halt_partition(XM_PARTITION_SELF);
}
```

Partition 1 :

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <xm.h>
#include <irqs.h>

#include "shared.h"

#define SHARED_ADDRESS  0x2200000

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorB;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;

  vec->checksum = 0;
  while (index < len)
  {
    XM_get_time(XM_HW_CLOCK, &val);
    memcpy (&(vec->vector[index]), &val, 1U);
    vec->checksum += vec->vector[index++];
  }
  vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  unsigned char checksum = 0;

  while (index < len)
  {
    checksum += vec->vector[index++];
  }
  return (checksum);
}

void PartitionMain(void)
{
  xmTime_t start, end, duration;
  unsigned char sharedChecksum;
```

```
    // Initialize shared mem
    memset (&vectorB, 0x00, sizeof (vectorB));
    PRINT("Initialized shared memory\n");
    XM_idle_self();

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    VectorFill (&vectorB, VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("Vector fill duration:  %lluus\n", duration);
    PRINT("Vector Checksum:  %u\n", vectorB.checksum);

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    memcpy (&(sharedMem->shareB), &vectorB, sizeof(vectorB));
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("memcpy vector duration:  %lluus\n", duration);

    // Wait for the other partition to update its vector
    while (!sharedMem->shareA.updated)
    {
      XM_idle_self();
    }
    PRINT("Checksum updated\n");

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    sharedChecksum = VectorCalcChecksum (&(sharedMem->shareA), VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("checksum calculation duration:  %lluus\n", duration);
    PRINT("calculated checksum:  %u\n", sharedChecksum);
    PRINT("shared checksum:  %u\n", sharedMem->shareA.checksum);
    if (sharedChecksum == sharedMem->shareA.checksum)
    {
        PRINT("Checksum match\n");
    }
    else
    {
        PRINT("Checksum mismatch\n");
    }

    XM_halt_partition(XM_PARTITION_SELF);
}
```

D.5 2 Partitions - Partikle - Shared Memory

Partition 0 :

```
#include <stdio.h>
```

```
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <xm.h>

#include "shared.h"

#define SHARED_ADDRESS  0x1000000

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

pthread_t t1;
memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorA;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;

  vec->checksum = 0;
  while (index < len)
  {
    XM_get_time(XM_HW_CLOCK, &val);
    memcpy (&(vec->vector[index]), &val, 1U);
    vec->checksum += vec->vector[index++];
  }
  vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  unsigned char checksum = 0;

  while (index < len)
  {
    checksum += vec->vector[index++];
  }
  return (checksum);
}

void *f(void *args) {
  xmTime_t start, end, duration;
  unsigned char sharedChecksum;

  // Initialize shared mem
  memset (sharedMem, 0x00, sizeof (memShare_t));
  PRINT("Initialized shared memory\n");
  XM_idle_self();

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  VectorFill (&vectorA, VECTOR_LEN);
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
```

```
    // Calculate duration
    duration = end - start;
    PRINT("Vector fill duration:  %lluus\n", duration);
    PRINT("Vector Checksum:  %u\n", vectorA.checksum);

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    memcpy (&(sharedMem->shareA), &vectorA, sizeof(vectorA));
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("memcpy vector duration:  %lluus\n", duration);

    // Wait for the other partition to update its vector
    while (!sharedMem->shareB.updated)
    {
      XM_idle_self();
    }
    PRINT("Checksum updated\n");

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    sharedChecksum = VectorCalcChecksum (&(sharedMem->shareB), VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("checksum calculation duration:  %lluus\n", duration);
    PRINT("calculated checksum:  %u\n", sharedChecksum);
    PRINT("shared checksum:  %u\n", sharedMem->shareB.checksum);
    if (sharedChecksum == sharedMem->shareB.checksum)
    {
        PRINT("Checksum match\n");
    }
    else
    {
        PRINT("Checksum mismatch\n");
    }

    XM_halt_partition(XM_PARTITION_SELF);
    return (void *)32;
}

int main (int argc, char **argv) {
    void *ex = 0;
    pthread_create (&t1, 0, f, 0);
    pthread_join (t1, &ex);
    return 0;
}
```

Partition 1 :

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <xm.h>
```

```
#include "shared.h"

#define SHARED_ADDRESS  0x1000000

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

pthread_t t1;
memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorB;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;

  vec->checksum = 0;
  while (index < len)
  {
    XM_get_time(XM_HW_CLOCK, &val);
    memcpy (&(vec->vector[index]), &val, 1U);
    vec->checksum += vec->vector[index++];
  }
  vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  unsigned char checksum = 0;

  while (index < len)
  {
    checksum += vec->vector[index++];
  }
  return (checksum);
}

void *f(void *args) {
  xmTime_t start, end, duration;
  unsigned char sharedChecksum;

  // Initialize shared mem
  memset (&vectorB, 0x00, sizeof (vectorB));
  PRINT("Initialized shared memory\n");
  XM_idle_self();

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  VectorFill (&vectorB, VECTOR_LEN);
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("Vector fill duration:  %lluus\n", duration);
  PRINT("Vector Checksum:  %u\n", vectorB.checksum);

  // Get start time
```

```
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  memcpy (&(sharedMem->shareB), &vectorB, sizeof(vectorB));
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("memcpy vector duration:  %lluus\n", duration);

  // Wait for the other partition to update its vector
  while (!sharedMem->shareA.updated)
  {
    XM_idle_self();
  }
  PRINT("Checksum updated\n");

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  sharedChecksum = VectorCalcChecksum (&(sharedMem->shareA), VECTOR_LEN);
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("checksum calculation duration:  %lluus\n", duration);
  PRINT("calculated checksum:  %u\n", sharedChecksum);
  PRINT("shared checksum:  %u\n", sharedMem->shareA.checksum);
  if (sharedChecksum == sharedMem->shareA.checksum)
  {
      PRINT("Checksum match\n");
  }
  else
  {
      PRINT("Checksum mismatch\n");
  }

  XM_halt_partition(XM_PARTITION_SELF);
  return (void *)32;
}

int main (int argc, char **argv) {
  void *ex = 0;
  pthread_create (&t1, 0, f, 0);
  pthread_join (t1, &ex);
  return 0;
}
```

D.6 2 Partitions - Mixed Bare C and Partikle - Shared Memory

Partition 0 :

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <xm.h>
#include <irqs.h>

#include "shared.h"

#define SHARED_ADDRESS  0x2200000
```

```
#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorA;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;

  vec->checksum = 0;
  while (index < len)
  {
    XM_get_time(XM_HW_CLOCK, &val);
    memcpy (&(vec->vector[index]), &val, 1U);
    vec->checksum += vec->vector[index++];
  }
  vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  unsigned char checksum = 0;

  while (index < len)
  {
    checksum += vec->vector[index++];
  }
  return (checksum);
}

void PartitionMain(void)
{
  xmTime_t start, end, duration;
  unsigned char sharedChecksum;

  // Initialize shared mem
  memset (sharedMem, 0x00, sizeof (memShare_t));
  PRINT("Initialized shared memory\n");
  XM_idle_self();

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  VectorFill (&vectorA, VECTOR_LEN);
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("Vector fill duration:  %lluus\n", duration);
  PRINT("Vector Checksum:  %u\n", vectorA.checksum);

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  memcpy (&(sharedMem->shareA), &vectorA, sizeof(vectorA));
  // Get stop time
```

```
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("memcpy vector duration:  %lluus\n", duration);

  // Wait for the other partition to update its vector
  while (!sharedMem->shareB.updated)
  {
    XM_idle_self();
  }
  PRINT("Checksum updated\n");

  // Get start time
  XM_get_time(XM_HW_CLOCK, &start);
  // Run task
  sharedChecksum = VectorCalcChecksum (&(sharedMem->shareB), VECTOR_LEN);
  // Get stop time
  XM_get_time(XM_HW_CLOCK, &end);
  // Calculate duration
  duration = end - start;
  PRINT("checksum calculation duration:  %lluus\n", duration);
  PRINT("calculated checksum:  %u\n", sharedChecksum);
  PRINT("shared checksum:  %u\n", sharedMem->shareB.checksum);
  if (sharedChecksum == sharedMem->shareB.checksum)
  {
      PRINT("Checksum match\n");
  }
  else
  {
      PRINT("Checksum mismatch\n");
  }

  XM_halt_partition(XM_PARTITION_SELF);
}
```

Partition 1 :

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <xm.h>
#include "shared.h"

#define SHARED_ADDRESS  0x2200000

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

pthread_t t1;
memShare_t* const sharedMem = (memShare_t* const)SHARED_ADDRESS;
vecWChck_t  vectorB;

static void VectorFill(vecWChck_t* vec, unsigned int len)
{
  unsigned int index = 0;
  xmTime_t     val;
```

```
    vec->checksum = 0;
    while (index < len)
    {
      XM_get_time(XM_HW_CLOCK, &val);
      memcpy (&(vec->vector[index]), &val, 1U);
      vec->checksum += vec->vector[index++];
    }
    vec->updated = 1;
}

static unsigned char VectorCalcChecksum(vecWChck_t* vec, unsigned int len)
{
    unsigned int index = 0;
    unsigned char checksum = 0;

    while (index < len)
    {
      checksum += vec->vector[index++];
    }
    return (checksum);
}

void *f(void *args) {
    xmTime_t start, end, duration;
    unsigned char sharedChecksum;

    // Initialize shared mem
    memset (&vectorB, 0x00, sizeof (vectorB));
    PRINT("Initialized shared memory\n");
    XM_idle_self();

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    VectorFill (&vectorB, VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("Vector fill duration:  %lluus\n", duration);
    PRINT("Vector Checksum:  %u\n", vectorB.checksum);

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    // Run task
    memcpy (&(sharedMem->shareB), &vectorB, sizeof(vectorB));
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("memcpy vector duration:  %lluus\n", duration);

    // Wait for the other partition to update its vector
    while (!sharedMem->shareA.updated)
    {
      XM_idle_self();
    }
    PRINT("Checksum updated\n");

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
```

```
    // Run task
    sharedChecksum = VectorCalcChecksum (&(sharedMem->shareA), VECTOR_LEN);
    // Get stop time
    XM_get_time(XM_HW_CLOCK, &end);
    // Calculate duration
    duration = end - start;
    PRINT("checksum calculation duration:  %lluus\n", duration);
    PRINT("calculated checksum:  %u\n", sharedChecksum);
    PRINT("shared checksum:  %u\n", sharedMem->shareA.checksum);
    if (sharedChecksum == sharedMem->shareA.checksum)
    {
        PRINT("Checksum match\n");
    }
    else
    {
        PRINT("Checksum mismatch\n");
    }

    XM_halt_partition(XM_PARTITION_SELF);
    return (void *)32;
}

int main (int argc, char **argv) {
    void *ex = 0;
    pthread_create (&t1, 0, f, 0);
    pthread_join (t1, &ex);
    return 0;
}
```

D.7 2 Partitions - Bare C - Sampling and Queuing Messaging

Partition 0 :

```
#include <string.h>
#include <stdio.h>
#include <xm.h>
#include <irqs.h>

#define QPORT_NAME "portQ"
#define SPORT_NAME "portS"

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

char qMessage[128];
char sMessage[128];

void PartitionMain(void)
{
    xm_s32_t qDesc, sDesc, e;
    xm_u32_t flags, sSeq, qSeq;
    xmTime_t start, stop, duration;

    PRINT("Opening ports...\n");                                    /*
        Create ports */
    qDesc = XM_create_queuing_port(QPORT_NAME, 16, 128, XM_SOURCE_PORT);    /*
        Parameters of creation              */
    if (qDesc < 0) {                                                /*
        calls must match XML configuration */
```

```
        PRINT("error %d\n", qDesc);
        goto end;
    }
    sDesc = XM_create_sampling_port(SPORT_NAME, 128, XM_SOURCE_PORT);
    if (sDesc < 0) {
        PRINT("error %d\n", sDesc);
        goto end;
    }
    PRINT("done\n");

    PRINT("Generating messages...\n");
    sSeq = qSeq = 0;
    for (e=0; e<1; ++e) {
        sprintf(sMessage, "<<sampling message %d>>", sSeq++);
        PRINT("SEND %s\n", sMessage);
        // Get start time
        XM_get_time(XM_HW_CLOCK, &start);
        XM_write_sampling_message(sDesc, sMessage, sizeof(sMessage));
        // Get stop time
        XM_get_time(XM_HW_CLOCK, &stop);
        // Calculate duration
        duration = stop - start;
        PRINT("Duration to send sampling message:  %lluus\n", duration);
        XM_idle_self();

        sprintf(qMessage, "<<queuing message %d>>", qSeq++);
        PRINT("SEND %s\n", qMessage);
        // Get start time
        XM_get_time(XM_HW_CLOCK, &start);
        XM_send_queuing_message(qDesc, qMessage, sizeof(qMessage));
        // Get stop time
        XM_get_time(XM_HW_CLOCK, &stop);
        // Calculate duration
        duration = stop - start;
        PRINT("Duration to send queuing message:  %lluus\n", duration);
        XM_idle_self();
    }
    PRINT("Done\n");

end:
    XM_halt_partition(XM_PARTITION_SELF);
}
```

Partition 1 :

```
#include <string.h>
#include <stdio.h>
#include <xm.h>
#include <irqs.h>

#define QPORT_NAME      "portQ"
#define SPORT_NAME      "portS"

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

char sMessage[128];
char qMessage[128];
xm_s32_t qDesc, sDesc, seq;
```

```
void ChannelExtHandler(trapCtxt_t *ctxt)
{
    xm_u32_t flags;
    xmTime_t start, stop, duration;

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    if (XM_receive_queuing_message(qDesc, qMessage, sizeof(qMessage), &flags) >
        0) {
        // Get stop time
        XM_get_time(XM_HW_CLOCK, &stop);
        // Calculate duration
        duration = stop - start;
        PRINT("Duration to receive queuing message:  %lluus\n", duration);
        PRINT("RECEIVE %s\n", qMessage);
    }

    // Get start time
    XM_get_time(XM_HW_CLOCK, &start);
    if (XM_read_sampling_message(sDesc, sMessage, sizeof(sMessage), &flags) > 0)
        {
        // Get stop time
        XM_get_time(XM_HW_CLOCK, &stop);
        // Calculate duration
        duration = stop - start;
        PRINT("Duration to receive sampling message:  %lluus\n", duration);
        PRINT("RECEIVE %s\n", sMessage);
    }
    XM_unmask_irq(XM_VT_EXT_OBJDESC);
}

void PartitionMain(void)
{
    PRINT("Opening ports...\n");
    qDesc = XM_create_queuing_port(QPORT_NAME, 16, 128, XM_DESTINATION_PORT);
    if (qDesc < 0) {
        PRINT("error %d\n", qDesc);
        goto end;
    }
    sDesc = XM_create_sampling_port(SPORT_NAME, 128, XM_DESTINATION_PORT);
    if (sDesc < 0) {
        PRINT("error %d\n", sDesc);
        goto end;
    }
    PRINT("done\n");

    InstallTrapHandler(XAL_XMEXT_TRAP(XM_VT_EXT_OBJDESC), ChannelExtHandler);
    HwSti();
    XM_unmask_irq(XM_VT_EXT_OBJDESC);

    PRINT("Waiting for messages\n");
    while (1);

end:
    XM_halt_partition(XM_PARTITION_SELF);
}
```