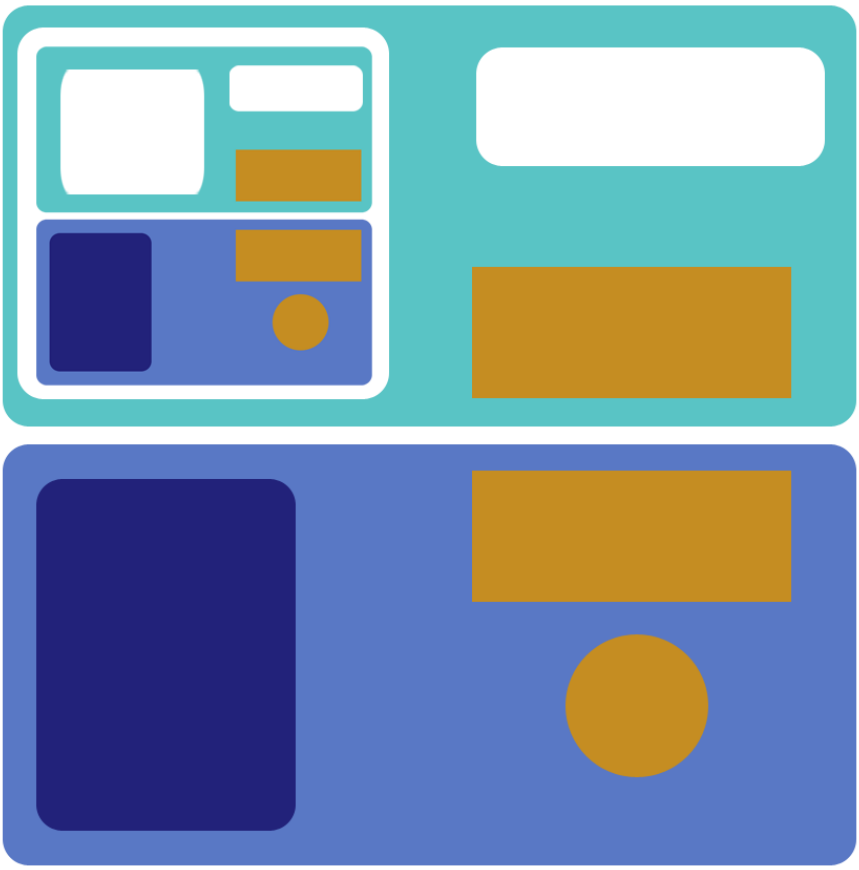
Evaluation of Real-Time Performance in Virtualized Environment



Nils Forsberg

Mälardalen University, Department of Innovation, Technology and Design

Supervisor: Mikael Åsberg

Examiner: Thomas Nolte

2011-05-29

Abstract

In this report is documented the research, tests and conclusions of a thesis work with the aim of investigating the possibilities of running real-time tasks in a virtualization environment. First we introduce the reader to the concepts and technology we will be touching on, and then we investigate the available solutions. We find that most of these are merely in a theoretical or development stage, and so we evaluate them theoretically. We also attempt to test one of the solutions that are fully developed and available, but fail because of issues related to the design of the solution. Based on our experiences and evaluations we come to the conclusion that the solutions available are lacking, and we give a suggestion of our own that we think should address the issues we have found.

Preface

The author wishes to thank the following contributors:

My supervisor Mikael Åsberg, PhD Student at Mälardalen University for continuous help and advice, whenever I needed it.

Sisu Xi, PhD at Washington University, St Louis, co-creator of RT-Xen for help and advice during installation of RT-Xen.

Table of Contents

1. Introduction

This is the first section of this document. It will introduce the different concepts needed to understand the terms in the following problem formulation and discussion. First we will take an in-depth look at subjects relevant for this thesis and explain what kinds of problems are present and why we would want to solve them. We will also look at existing solutions for these problems.

1.1 Background

Here we introduce the reader to the concepts examined in this thesis. The subjects of scheduling, virtualization and real-time computing is explained, as well as why real-time and virtualization is interesting to combine and what the problems associated with this are.

1.1.1 Scheduling

To achieve acceptable performance most computers today uses a technique called multiprogramming, i.e. they switch between tasks or processes, processing each for a short time in rapid succession creating the illusion that everything is done simultaneously. Multi-core processors exists, increasing the possibilities for this, but the number of running applications on any system often outnumber even the most capable system's cores. This switching between tasks needs to be done in a particular way; some processes may require higher priority than the rest and so should get CPU time sooner than these. Still others may be blocked while waiting for an I/O (input/output) event and it would be beneficial for the entire system to let some other task run instead of leaving the CPU blocked. To decide how CPU resources should be distributed, operating systems has a component called a *scheduler*. Different system environments require different scheduling strategies. In order to give the reader sufficient knowledge to be able to understand the terminology and problem formulation used in later sections some of these environments will be presented here. Environments can be categorized as follows:

- **Batch System** – This classification is common in business-type computers where the focus is on calculating account balance or interest and stock exchange rates where there is no user waiting impatiently behind the keyboard. This means that processes are allowed to take their time without having to be interrupted by other tasks needing to execute, i.e. non-preemptive scheduling is used.
- **Interactive System** – "Ordinary" computers, desktop stations used at work or in homes. These kinds of systems need to be responsive. Because of this it is preferred to avoid having one process block all the others and achieve good performance on all tasks, meaning preemption is used frequently in interactive systems.
- **Real time System** – Common in embedded systems such as cars, airplanes and mobile phones, real-time classified systems need to finish their tasks before their given deadline to avoid a system failure, and a potential catastrophe.

Now that different criteria have been presented we may look at different scheduling strategies. In the examples shown a capital letter corresponds to a task and the number following it is the time length it needs to run. If it is necessary priority is included as well.

1. **First-Come First-Served (batch)** – This might be considered the simplest scheduling strategy; tasks are executed in precisely the order they arrived without being interrupted. The next process in the run queue is not allowed to run until the first either finishes or is blocked

because it is waiting for an event and the concept of priority is not present here. If tasks A(15), B(10), C(20), D(5), E(10) arrive in that order then that is how they will execute: A -> B -> C -> D -> E

2. **Shortest Job First (batch)** – This algorithm, as its name implies, selects the task with the shortest execution time available and lets it run first. It is required that the run times of each task are known beforehand. SJF will finish more tasks in a shorter time and it is optimal if all tasks are available from the start. If we have tasks A(15), B(10), C(20), D(5), E(10) and they are all available from the start the execution order will be: D -> B -> E -> A -> C
   If they become available one by one at different times, a newly arrived task will interrupt a running task if it is shorter than the running one and if *preemption* (interruption of a running process) is used.
3. **Round Robin (interactive)** – With the Round Robin strategy, every process gets an equal share of the CPU; each task takes turns executing for an equally long time each. While being preemptive is an obvious requirement, there is still no priority. So depending on how long each time period or quantum is and how many tasks there are, tasks may have to wait for a long time before they can start running again. Tasks A(15), B(10), C(20), D(5), E(10) will run for say, 5 time units each until they have finished like so: A -> B -> C -> D -> E -> A -> B -> C -> E -> A -> C -> C
4. **Priority Scheduling (interactive)** – Every task is assigned a priority rating, depending on how urgent it is that they finish, and the highest priority tasks may run first. A problem that may arise is that low priority tasks never get any execution time. A solution might be to decrease the priority rating of tasks that have been running a lot. Other designs exist where tasks are grouped into priority hierarchies and when they have consumed their quantum in a priority level their rating is decreased and they are moved to the lower level. For tasks A(15, 4), B(10,3), C(20,1), D(5,2), E(10,5) we will get: C -> D -> B -> A -> E
   Note that task E will have to wait 20+5+10+15=50 time units before it is allowed to run.
5. **Lottery Scheduling (interactive)** – The Lottery algorithm gives out "tickets" to processes based on their priority or similar. With these tickets processes have a chance to "win" execution time. This lottery is held at regular intervals. Since each process holds different amount of tickets of a total set of tickets, their CPU time corresponds to the amount of tickets they have compared with that total amount; holding 30% of the tickets means 30% CPU time, so all processes should be able to execute some time or another. Once again with tasks A(15, 4), B(10,3), C(20,1), D(5,2), E(10,5). It cannot be predicted in what exact sequence the tasks will run, but let's say there are a hundred tickets. C with the highest priority will receive a majority of these, maybe 50. This means it will get an average of 50% CPU time for this time period. D will receive perhaps 20 tickets and is awarded with 20% CPU time**,** B will get 15 tickets and 15% CPU time and so on.

Real time scheduling is a slightly different story. Since it is partly the main subject of this work it is presented in a section of its own below [14].

1.1.1.1 Hierarchical Scheduling

Hierarchical scheduling is a concept that needs to be explained. It is used to describe a kind of scheduling used in *partitioned systems* (a system containing several sub systems). Here two levels of schedulers are used. These are usually not aware of each other. Hierarchical scheduling can be illustrated like in figure 1. Here we can see that the global scheduler schedules the partitions much

like individual tasks; they each get a fair share of the CPU. The local schedulers schedules their tasks in this allotted time; a local task will only ever get as much CPU time as the partition will be given [5] [13].

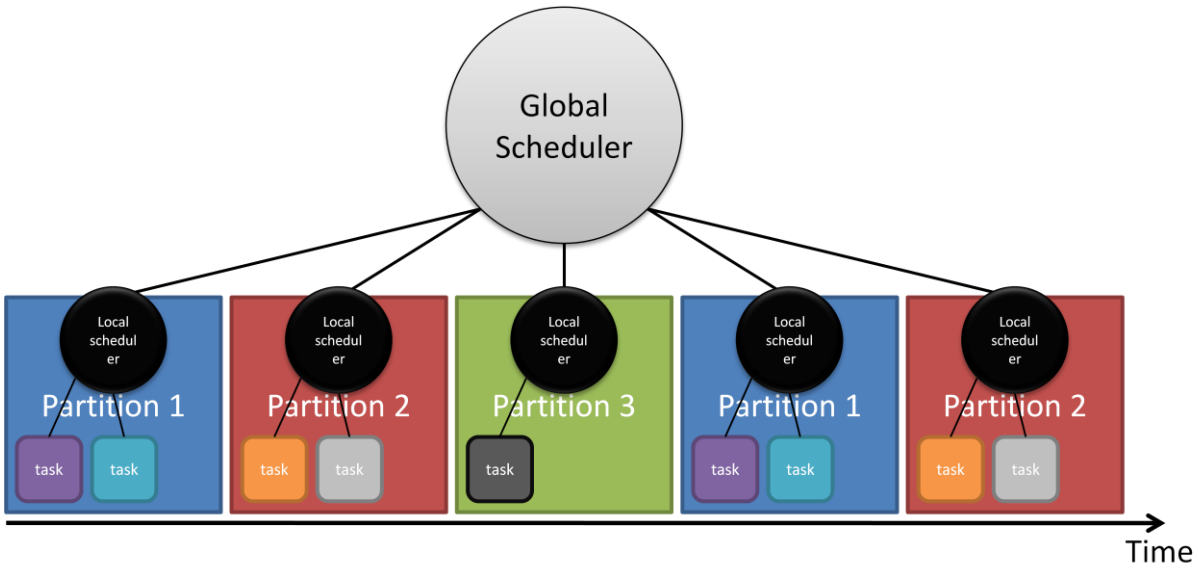


Figure 1: A visual representation of a hierarchical scheuling model

1.1.2 Real-Time Computing

For those unfamiliar with the term, real-time computing is a branch of computer science whose main focus is on deadlines. In real-time systems tasks and programs must execute before their deadline or a system failure occurs. A popular example is the embedded computer system of a car: the program handling the airbag in case of collision must execute and respond to fill the airbag with air before the driver's head hits the windshield instead. This is also an excellent example where system failure would mean a fatal outcome for human beings. It should not be assumed that real-time computer systems require very fast and advanced computational power. While this might be the case if the task at hand is a complex and demanding one, the only real concern is that the tasks finishes in time, before their deadlines. Another example that illustrates this case is that of a chess player program. If this program is to be as efficient as possible it might be beneficial to remove any time constraints, allowing the program to take as much time as needed to find the best move to make. However, if the program is to be used in a competition with a timer where moves must be made within certain time limits, the conditions are different. Here a real-time factor is introduced, since if the next move cannot be decided within the given time span the program will lose. It can also be noted that high performance should be considered desirable in this program, since moves are improved by more advanced computations.

There are different classifications of real-time. Depending on what kind of system it is the need for real-time performance may be different. If it misses a deadline and the consequences are severe the system is said to be hard real-time; one single failure means the failure of the entire system and a possibly fatal outcome, like in the example with the airbag above. If missed deadlines only results in degraded performance of the system it is of soft type real-time. A practical example might be a media player playing a movie: each frame must be processed and displayed at a steady rate. If one or even several frames are not displayed on time the movie can still be watched but the quality will be reduced. A common factor for all kinds of real-time environments is that they require a stable

platform, which means that the hardware and software base that is used must be tested for reliability and stability [14].

1.1.2.1 Real-Time Scheduling

The scheduling of real-time systems requires some attention compared to that of normal systems. Here the highest concern is to make sure that each task and process gets enough execution time and resources to complete its goals and, most importantly, that they can complete before their deadline. In order to accomplish this each task's requirements must be known before they can be scheduled. There are two ways to do this: Dynamic and static scheduling. Dynamic scheduling decides how to schedule tasks at runtime, making decisions based on current requirements and the state of tasks. This strategy is flexible, making sure that the system is always optimized since a task requiring lots of resources can take the place of another task that finishes its work quickly. The downside of it all is the increased amount of overhead calculations; the system must continuously decide how to schedule each task. Static scheduling works the opposite way: Scheduling decisions are made before execution, during compilation of the scheduler. The prerequisite is that data about the tasks that will run on the system must be collected before the scheduler can be built. This is optimal for systems whose behavior remains the same, since no changes to the scheduling can be made once the system is operational. There is also the choice of preemptive and non-preemptive schedulers. A system is preemptive if a running task can be aborted before it has finished its current operations in favor of a task with higher priority, while a non-preemptive system never interrupts its tasks.

Different kinds of strategies exist to realize real time scheduling. While there are too many different variants and offshoots too study in detail, we should look at one possible way to verify the scheduling of a set of tasks. Let us consider a theoretical example: if tasks are periodic, i.e. they arrive at regular intervals, and that these intervals are known prior to scheduling and designated as *P*. Let it also be known that tasks must run for *C* time units (e.g. seconds) during each such period. Task *i* runs for *C* seconds every *P* seconds. If there are a total of *n* tasks and we know each of these properties for every task, we can find out if this set of tasks is schedulable, if all of them will complete in time every period. The formula to be used is this:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

This means that the task set is clearly schedulable and that every task should be able to run their required amount of time without missing any deadlines. Too examine this closer we can look at a visual representation of the tasks running times. Tasks are color coded as such: **A**, **B**, **C**, and **D**. Each square is equal to one second, which means that an entire row below is equal to twenty seconds. Every fifth second is marked.

1				5					10					15					20
21				25					30					35					40
41				45					50					55					60

As can be seen each task finishes on time; A runs for two seconds every ten seconds in a regular pattern, B and C has similar patterns. D's running times are slightly irregular, but it nevertheless finishes right on time, using the freed up slots created where other tasks had finished. We can also note that it is possible to insert one more task into the system without causing deadline misses; it

would have to run for six seconds every sixty seconds, or we could insert two tasks each running three seconds every sixty seconds. Inserting these values into the formula proves we are correct:

$$\frac{6}{60} = 0.1$$

$$0.9 + 0.1 = 1$$

The criterion for successful scheduling is still satisfied [14].

1.1.3 Virtualization

This section attempts to explain the concept of virtualization in computer science, the different types of virtualization and what can be accomplished with it.

Virtualization is the creation of a virtual system capable of the same things as an actual version of the same system. Examples include virtual operating systems, storage or network devices and hardware [18]. It should not be confused with emulation where a program also attempts to imitate another program or system; virtualization also simulates hardware but they are used in differently. Emulation is mainly aimed towards the recreation of older or unavailable systems or running applications on systems for which it was not designed [19]. There are different ways to implement virtualization. The technique interesting for this thesis is hardware virtualization, so other techniques will not be discussed.

1.1.3.1 Why Use Virtualization?

Virtualization is a clever solution to a problem where the interests of reliability and cost savings conflict. Let us for example consider an internet server company who offers many different services to their customers. They may need an e-mail server, a website server and an FTP server to allow files to be downloaded. With today's hardware a single computer should be able to run all the applications handling these services without any problem. The reason you would not want to do this is because of stability; if one of these applications crashes and brings the system down with it, everything will stop working. This is the reason why each software application is installed in its own expensive computer. Virtualization offers a solution to this: By creating several virtual machines on one computer and installing the server applications in each of these the programs are isolated and if they crash the others will remain unaffected. Most failures are the cause of faulty software, while hardware, as stated, is relatively stable. This diminishes the still present risks of having all programs on a single computer [14].

1.1.3.2 Hardware Virtualization

Hardware virtualization is the specific name for virtualization of computers and operating systems. It provides an abstraction of computer hardware and hides its actual properties from whatever application that uses it, usually an operating system [20]. This is often handled by software specifically dedicated to the task, often called a *hypervisor*. The name is derived from the fact that it has even higher control than a supervisor program. This hypervisor creates a simulated computer environment, a virtual operating platform known as a *virtual machine* on which operating systems can run. There are different ways to do this. The two following methods stands in contrast to each other, while the third deals with virtualization of lesser magnitude.

1.1.3.2.1 Full Virtualization

Full virtualization is a complete simulation of hardware in a virtual machine. It is more or less a software imitation of an entire computer system, and any operating system running on it is incapable of telling the difference between it and an actual hardware system. This enables various operating systems to run unmodified and as-is with these kinds of hypervisors. These kinds of systems are not very common however, mostly due to the fact that they are a very demanding implementation; every small detail and aspect of hardware must be duplicated in the hypervisor software [21]. The processor of the computer that is running the virtualization also needs to be able to support hardware virtualization, which is done with certain built-in techniques from the manufacturers [22] [23]. The obvious advantage is of course that just about any operating system can be run alongside others, such as Windows and even MAC OS may be active on virtual machines on the same system. This is rarely possible with Paravirtualization where modified operating systems are needed, as explained in the next section.

1.1.3.2.2 Paravirtualization

Paravirtualization is a virtualization technique like full virtualization but unlike it, paravirtualization does not offer a complete simulation of hardware. Instead it only simulates certain features of a physical system, making guests run partly on hardware. The aim of this design is to improve performance of the execution of guests by removing the difficult parts of virtualization from the hypervisor. This method does require that guests be modified as well, which leads to an increase in maintenance cost [24].

1.1.3.2.3 Partial Virtualization

Partial virtualization simulates only a part of a piece of hardware, often a systems address space. This makes it unsuitable for running entire operating systems on a partial virtualization's virtual machine. Instead, partial virtualization is aimed at applications which only need to utilize a specific address space on different systems. While being easy to implement, this kind of virtualization may require extensive backwards compatibility or portability depending on the application it is supposed to run, something that can be difficult to anticipate. These kinds of virtual machines are very stable however, and can be used successfully for sharing resources between user instances [25] [14].

1.1.3.3 Virtualization Implementations

After examining the available techniques, this section takes a closer look at specific implementations of these and explains their setup and requirements. There are different types of hypervisors: Type 1 is a hypervisor whose software runs directly on the systems hardware without the need of an underlying operating system. It can also be said to be the operating system. Type 2 is the kind of hypervisor that is installed on top of another system [14].

1.1.3.3.1 Xen

Xen is a type 1 hypervisor running primarily paravirtualization, although it is capable of full virtualization as well if the hardware allows it. One of the most interesting things about this software is, as stated, that it runs directly on top of the system hardware as can be seen in figure 2, between it and the guest operating systems which should lead to improved performance.

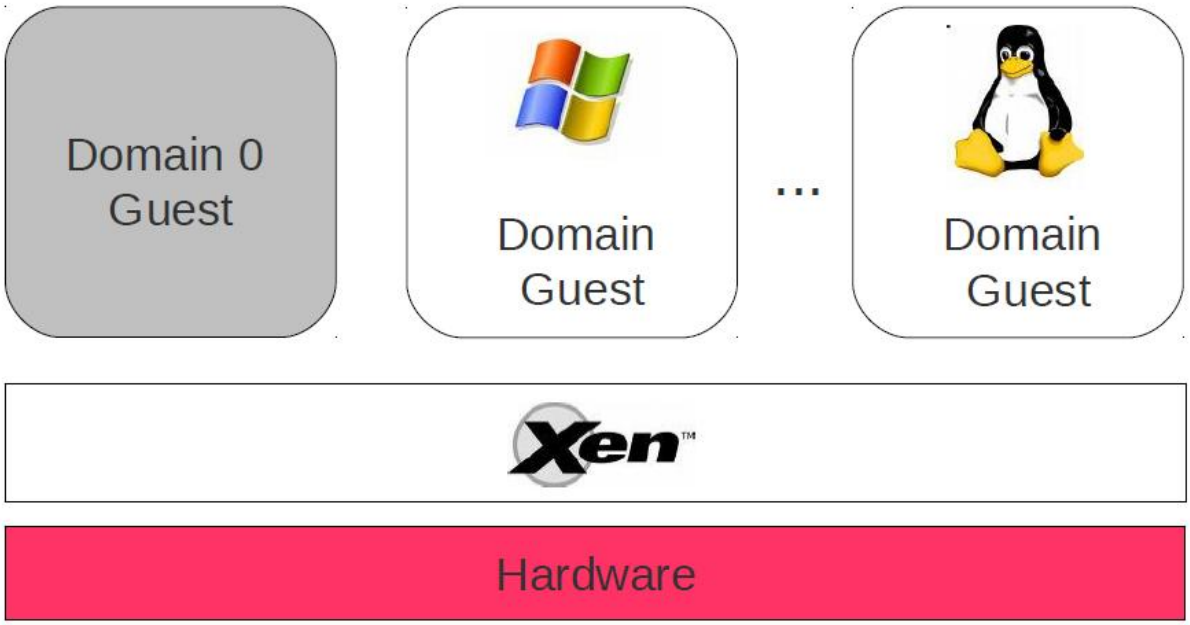


Figure 2: The structure of a Xen system

More specifically, the hypervisor is integrated into a Linux kernel. This means that, while Xen offers full virtualization where no modification of guest systems are needed, the privileged domain always require some changes to run, since it is the administration platform for the entire virtual system, responsible for starting and stopping the virtualized guests. These changes come in the form of modifications to the Linux kernel and although there are kernels available for download, the case is more than often that they must be configured and built on site [26]. Xen is open source software and its architecture is designed so that it is easy to add or modify it, e.g. you can comparatively easily change the scheduling with some advanced knowledge of coding [8].

1.1.3.3.2 Kernel-based Virtual Machine - KVM

KVM is a virtualization solution for Linux capable of full virtualization. It accomplishes this using techniques built into the hardware, specifically the processor. The implementation uses the Linux kernel to achieve good performance with most systems. Using KVM mostly involves a simple install procedure, and using the included tools an unmodified operating system can be installed as guest using an installation CD image. According to virtualization senior director Navin Thadani KVM is a type 1 hypervisor, despite rumors that might say otherwise [27]. KVM also has great support in most Linux distributions. Because it is open sourced software it should be possible to modify it to suit any specific needs and make it capable of real-time tasks, as we shall see in the Related Works section. Unmodified KVM is not suited for real-time environment since it incurs certain overhead [17], among other things.

1.1.3.3.3 VirtualBox

VirtualBox is an open source application with full virtualization support and it runs as an application on an operating system so only an installation is required to run it, meaning also that it is a type 2 hypervisor. It also does not require any specific hardware; its virtualization is made possible through heavily capable software. Because of this there is no need to use modified operating systems or kernels. VirtualBox has great support because it is being professionally developed by Oracle, and it is often the virtualization software of choice for private users as well as companies when it comes to

running multiple operating systems simultaneously. Since VirtualBox is purely a software installation there is no guarantee of real-time performance, but the application is open source and so it should be possible to modify it, although because of the architecture this might be difficult. No such solution has yet been implemented [28].

1.1.3.3.4 Oracle VM

Oracle VM is a hypervisor from the Oracle Corporation aimed at server maintenance. The software is actually based on the Xen project, and several of the Oracle crew helps develop Xen. Oracle VM is a bare metal hypervisor, i.e. a type 1 hypervisor that is installed directly on a hardware system. It does not feature a direct interface, only a web-based manager application, and as such is not suitable for desktop virtualization. Due to its nature there is also no intended support for a real-time environment, and it should not be the platform of choice for such modifications despite being open source [29].

1.1.3.3.5 VMware

VMware is a company offering basic desktop and server hypervisors, both for free. The desktop hypervisor is a type 2 hypervisor; it is installed as an application and is capable of running most Linux and Windows operating systems as host and guest respectively. The VMware server virtualization technology is a bare metal hypervisor, a type 1 hypervisor that is only used for server management. The company offers more advanced solutions at a cost [30]. There is little support for real-time performance in VMware's line of products; steps have however been taken to ensure proper performance with IP-telephony [31], as well as other soft real-time tasks [12].

1.1.3.3.6 QEMU

QEMU is not a hypervisor in the strict sense, but it is often used as such and should be mentioned here. Rather it could be said to be an emulator, capable of imitating a full hardware system. It is a type 1 hypervisor and many other hypervisors such as Xen, VirtualBox and KVM features technology based on it. QEMU is open source technology and so it is possible to make modifications to it [32]. There is however no proper real-time solution at this date.

1.1.3.3.7 Windows Virtual PC

Virtual PC is a Windows virtualization product. The latest version only supports the most recent editions of the Windows operating system, while the previous version was capable of running a Mac OS as guest as well. It is an application type 2 hypervisor, and the limitations of only being able to run one kind of operating system may be outweighed by the fact that it is available for free and with full support [33][34]. There are no possibilities of real-time performance on this platform; it is only supposed to give increased compatibility for Windows systems and these do not support real-time from stock. The code base is closed so the abilities to modify this hypervisor are small.

1.1.4 Virtualization in Real-Time Systems

Real-time performance is almost always needed in the embedded systems market, due to the critical and important tasks performed by them, e.g. break control in cars, flight systems in airplanes, etc. For a long time embedded systems were mostly small and isolated instances in various craft and machinery, where performing a single task was often the only thing needed. This is changing however, as embedded systems are taking on a larger, more varied and common role. They were first installed as several nodes in the same system, with each node performing a different task and sometimes cooperating with each other to reach common goals. This distribution is becoming

cumbersome because the increasing amount of functionality adds more nodes to the system, increasing hardware costs greatly. An obvious solution that is gaining ground is to relocate all functionality to a single hardware system, allowing many tasks to run and cooperate on this single computer, greatly reducing the need for hardware. This comes at a cost of software however, since whatever operating system that manages and schedules the programs controlling every different function needs to be a lot more complex. There is another issue associated with the quickly advancing development of real-time embedded systems, namely that of hybrid functionality devices such as smart phones. These kinds of systems must be able to handle mission critical real-time tasks such as signal broadcasting and networking, while at the same time execute general purpose applications such as movie and music players and games. Here too is a situation where several applications have to coexist on the same hardware, bringing with them some severe problems that could very possibly prevent undisturbed execution of the software on the device in question:

- **Resource Sharing** – While not a big problem in an advanced enough system, there is still the issue of two or more software applications having to share the computational and memory resources of the hardware, increasing the workload as more programs are added. For a real-time critical piece of software the consequences of not getting enough resources to complete within their allotted time may possibly be catastrophic depending on their task. This will also be a problem for general purpose programs since stuttering during the playback of a movie or a non-responsive game will be considered inferior quality of the device that is being used.
- **System Failure** – A lone application running into an error and causing the entire system to crash is a major problem for any system and especially in an embedded one. Here several critical real-time programs may be running and failing because the system that manages them crashes because one of these, or worse, a general purpose program, is making a forbidden execution. This kind of issue was not present in the distributed systems, where each application was isolated in the different nodes. The incorporation of several programs into one single computer introduces this kind of danger, and avoiding it is greatly sought after in the construction of these systems.

Server applications and hardware faced very similar problems some time ago; multiple software instances with different requirements and the capability of taking down other programs with them by causing a system crash was becoming a major problem as the technology and functionality of server systems advanced. The issue was solved with the use of virtualization software. Server applications would instead run on hypervisors mimicking actual hardware, creating isolated instances in which each program could operate freely and not disturb other functionalities in case of failure. The issue of resource sharing might still be present, or even increase, despite this solution, but increasingly advanced hardware such as multi-core processors and larger memory (and let us not forget clever implementations) would effectively solve this.

Due to the similarity of the problems it is with interest that real-time industry and researchers have recently begun looking at the alternatives of implementing embedded systems running on virtualized hardware. There are a number of issues and problems associated with this however.

1.1.4.1 Virtualization & Real-time Problems

While a combination of virtualization technology and real-time systems is an attractive solution to the problems that are appearing as the complexity and requirements of embedded systems are increasing, there are inherent issues and complications in compatibility that prevents a smooth transition into this kind of setup. Heiser [7] thinks that the limitations of virtualized technology are too great to act as a host for real-time demanding tasks, mentioning things such as broken integration between tasks and programs, scheduling problems and unaddressed issues of energy management and software complexity. Heiser instead suggests the addition of micro-kernels, lightweight and fully capable of supplementing hypervisors so that they may provide a virtualized environment that meets the demands. Kaiser [8], as well as Aguiar and Hessel [1], has similar opinions: virtualizations is a promising technology for solving the issues appearing in embedded systems, but it needs some modification and fine tuning before it can be of any proper use. The problems discussed are described in detail here:

- **Isolation** – While being one of the arguments for virtualization of embedded systems in the first place, the isolation incurred by tasks running in separate virtual machines can have a negative effect as well. The subsystems of an embedded system often needs to share their data, with each other or perhaps with any general purpose application, in order to function optimal or even properly. Hypervisors traditionally does not come with any kind of message system that would facilitate systems requiring real-time performance, relying instead on network connections, which works like that of actually separate computer systems due to the simulated hardware nature. This means that some extra copying would have to be done which would be an unwanted waste of processor power.
- **Scheduling** – Almost the defining aspect of real-time systems is that of scheduling. Scheduling algorithms for these kinds of systems differ from regular schedulers in that they must be able to guarantee that the running tasks will meet their deadline, taking into account the length of the tasks, their quotas and possibly making calculations to rearrange the execution order so that deadlines are not missed. Real-time systems are demanding, and their demands are not quite met by hypervisors today. Instead, the scheduler in virtual environments treats virtual machines as a normal system would treat applications; it gives them a fair quota to execute in the CPU with no regard to what is running within the virtual machine. The guests themselves are responsible for scheduling their processes and the hypervisor knows nothing about them.
- **Architecture** – Most hypervisors today are built to run on desktop or server systems, where the dominating architecture is x86. Embedded systems on the other hand use many different architectures and x86 oriented software may not be optimized for this market [8].
- **Code Base** – The size of the code base is an important aspect to consider. Measured in LoC (Lines of Code) it describes how large the source code of a program is. Most hypervisors have very large codebases, and since they often integrate themselves with a system kernel, their size can grow to several million LoC. Embedded systems have many safety requirements, and codebases of such sizes cannot guarantee that they will fulfill these requirements [8].

These issues may or may not be present depending on what kind of system we are talking about. Still, just about all of them are more or less present in most commercial and industrial embedded systems today.

1.2 Problem Formulation

The work in this thesis is aimed at the combination of virtualization and real-time performance. Virtualization is a technique where software imitates hardware, allowing other software applications such as operating systems to run on this virtual platform, possibly side by side with other so called virtual machines. There are a number of different ways and methods to do this, with advantages and disadvantages. Meanwhile, the field of real-time performance is advancing, and due to issues that arise with this advancement a lot of research has recently been directed towards the problem of how to adapt virtualization to the requirements of real-time systems. This work is an evaluation of the existing solutions to this problem, in order to get an overview of the state of the research in this field. The solutions should be analyzed with an emphasis on their real-time performance, how easy they are to use and how they can be changed suit any specific needs. If the investigation shows that existing solutions are inefficient, these shortcomings should be pointed out and addressed in the form of a suggestion of its own. The questions that should be addressed in this thesis work are:

- What different techniques are available for virtualization?
- What are their real-time performance capabilities?
- Do they need modification for real-time adaption, and how good is that performance?
- Are these techniques insufficient? If yes, what should be used instead? Why?

1.3 Related Work

Combining real-time scheduling with virtualization is a relatively new research area. Despite this there are several researchers and groups testing and implementing different setups of this kind. This section will examine some of these. Please note that some of them are further examined and evaluated in later sections.

There has been one interesting attempt at enhancing the Xen hypervisor and enable it to handle the challenges imposed by real-time scheduling. Carried out as a joint project between different schools at Shanghai Jiao Tong University, it improves the default scheduler in Xen, tuning it for real-time execution. The result is proven to be a 20% improvement of handling real-time tasks [16].

A similar attempt has been carried out at Beijing University. Here it is also argued that the Xen hypervisor is not adapted to real-time tasks and changes or additions to it must be made in order to accommodate real-time guests. This method is also proved to be successful through experimental measurements [15].

Also from Shanghai comes a suggestion of an embedded real-time virtualization architecture based on the Kernel-based Virtual Machine hypervisor. Here the execution of the real-time tasks is handled by VxWorks, a real-time operating system, while the general operating system Linux is used to run general purpose tasks. Both these operating systems are installed as virtual machines on a KVM and Linux combination hypervisor. Tests were run to see how the real-time tasks were affected by running on a virtual machine together with other guests, and what kind of latencies KVM introduced to the real-time operating system. An architecture based on the results were then proposed, where modifications to the host operating system was the solution [17].

Li et al. proposes a real-time scheduling mechanism for virtual machines based on list scheduling. Two algorithms have been designed: The Virtual Machine Management algorithm and the Processor Selection algorithm. According to the authors these designs have the potential to be successful [11].

Investigations of the Xen hypervisor's capabilities as a media server have been made by Lee et al. at the Georgia Institute of Technology and Avaya Labs respectively. Classifying music and video playback as soft real-time tasks, they found that Xen is unable to support these kinds of task without notable performance loss. The problem was discovered to be because the virtual machines running the soft real-time tasks were not given enough CPU time. The Xen hypervisor uses a mechanism where tasks that are waiting for input or an event have their priorities boosted when one such arrives. The interesting thing was that the Xen host (known as dom0) spent almost all its time in this boosted state because it is almost always waiting for some input, effectively stealing the guest virtual machines CPU time. A guest can only be boosted from a blocked state, and when it is running a media application it is considered to be in a running state, making it impossible for them to ever compete with dom0 for execution time. Modifications were made and tests made with an IP telephony program showed that this new setup performed well and that non-real-time tasks were not affected by the changes [10].

Bruns et al. notes the lack of proper tests for microkernel-based hypervisors; several comparisons have been made of a microkernel with different real-time operating systems, but there are few benchmark tests where realistic use-cases are also included. According to the authors the present solution in mobile phones, physically divided subsystems each dedicated to either real-time or

general purpose functionality, is ineffective and a hypervisor solution would be a lot more cost effective. The article presents test results made with the L4/Fiasco microkernel as a hypervisor for a merged system compared with the FreeRTOS real-time kernel [4].

1.2.1 VSched

In their article, Bin Lin and Peter A Dinda describe their scheduler VSched for the program Virtuoso, also designed by the authors. VSched is a user level tool that is part of the Virtuoso project, and Virtuoso is an application that is supposed to be middleware for virtual machine computing, presenting the user with tools to manage virtual machines and customize them in many ways. Lin and Dinda concludes that VSched is successful in that it has very low deadline miss rates, and that several different tasks such as long computations and interactive applications can be run on the virtual machines simultaneously without losing usability [12].

1.2.2 The RT-Xen Project

RT-Xen is a project primarily carried out by PhD students Sisu Xi and Justin Wilson at Washington University in St. Louis, USA. The goal is to extend the open source hypervisor Xen with hierarchical scheduling to achieve hierarchical real-time scheduling and performance in this virtualized environment. RT-Xen is a framework currently under development, but it is already available as a patch to the original Xen. RT-Xen is based on virtual CPUs that are pinned on one physical CPU. The VCPUs are assigned as being of a certain category, which gives them properties that are used for deciding how they should be scheduled. Four different types of VCPU, or server, are included in the framework: Deferrable Server, Periodic Server, Polling Server and Sporadic Server. They have property parameters such as *budget*, *period* and *priority* that are used to decide how they should be moved between different queues in the physical CPU core. Depending on what strategy (i.e. Deferrable, Periodic, Poloing or Sporadic) is used, the queues are handled slightly different, but what follows is a basic overview: The first queue, *RunQ* lists VCPUs with tasks that should run, in order of priority. A function called *do_schedule* is called at each quantum to see if there is any VCPU with higher priority available; if there is the VCPU running is removed and stored in either *RunQ* or *RdyQ*, depending on priority and budget. This *RdyQ* is used to keep track of VCPUs without any tasks that need to run. These VCPUs are compared with the ones in the *RdyQ* to decide if it should be scheduled anyway. If it has some budget left despite being idle and has the highest priority in the *RdyQ*, it is inserted at the back of the running queue. Finally we have the *RepQ*. “Rep” stands for “replenishment” and the purpose of this queue is to keep track of when a VCPU should have its budget restored, and how much. The *RepQ* is also checked each quantum to see if the priority of a replenished VCPU is higher than a running one, in which case it is inserted instead of the running one. VCPUs that are allowed to run are scheduled with a fixed priority, preemptive scheduling scheme. The project are amongst the first of its kind, and are reporting successful test results, claiming that real-time scheduling can be achieved in a virtualized environment in an effective way [44][35].

1.2.3 Company Solutions

There are also a few companies offering products capable of combining real-time systems and virtualized hardware as well.

1.2.3.1 LynuxWorks

LynuxWorks is a company offering highly secure virtualization/real-time combination systems. Their solution is to use a separation kernel with an embedded hypervisor that is installed directly on the

device that it is supposed to use, making it a type 1 hypervisor. It may not be the best choice for desktop virtualization however, as it is mostly used in systems that require high security, safety and reliability such as military and aerospace computers or embedded systems, as well as in telecommunications and automated systems. The LynuxWorks hypervisor is bare metal and capable of running Linux, Windows and LynuxWorks' own operating systems [36] as depicted in figure 3.

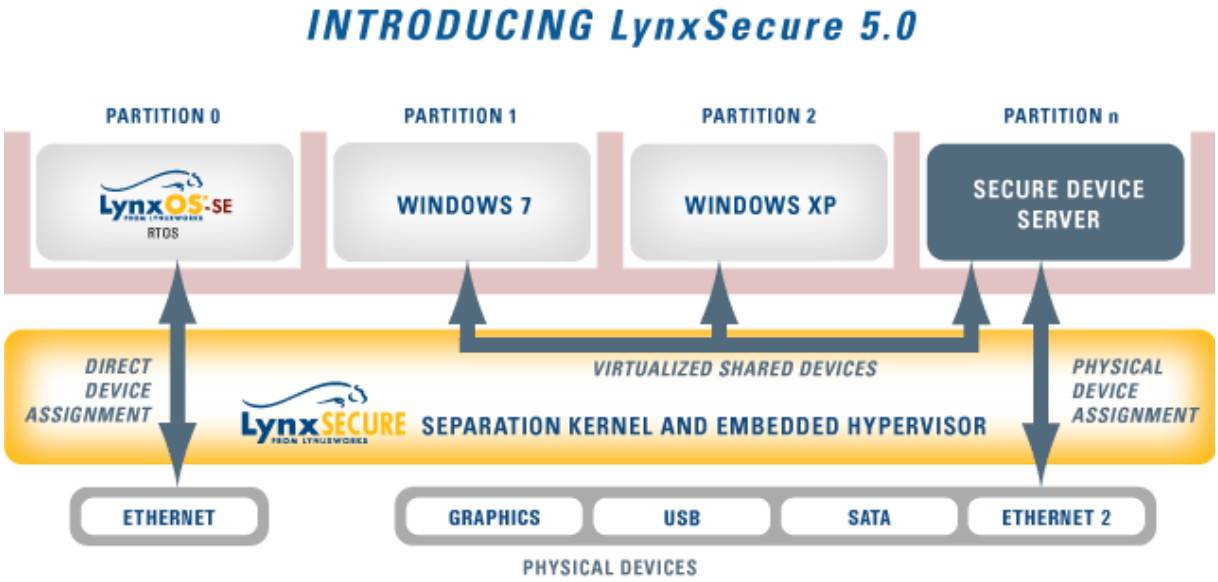


Figure 3: LynuxWorks' hypervisor system: LynxSecure.

1.2.3.2 Open Kernel Labs

Open Kernel Labs is a half commercial and half research community company with the main focus of developing commercial systems, such as smart phones. For this particular area OK Labs have designed and built a so-called microvisor, an embedded hypervisor based on a microkernel[1]. The purpose of this microvisor, called OKL4 Microvisor, was to reduce the costs of the hardware, allowing smart phones to be made with lower production costs while still capable of running the increasing amount of software that is demanded on these platforms [37].

[1] A microkernel, according to OK Labs, is a kernel but with as few abstraction sets as possible. These can be used by someone wishing to add operation system services [37].

2. Treatise

This section will present the work and describe the procedure of the research that was carried out. Any problems encountered will also be listed as well as any measures taken to solve them. A theoretic analysis of the capabilities of each solution is also available, accompanying their subsections.

2.1 Installing Xen

Installing the hypervisor Xen on a computer system and making it running was the first task that needed to be solved. The first step was taken on the Internet in an attempt to find a guide on how to proceed. Ubuntu was the desired platform because the scheduling framework Resch, which we wanted to use, had only been tested here, and to make a fair comparison, different tests should be made in environments that are as similar as possible. Since Ubuntu was the preferred target system, the Ubuntu Community web page documenting Xen [38] was where the attention was initially directed. This section will also focus largely on how to install Xen on Ubuntu, while other system setups will be described later. On the web site several things could be learned:

- Xen is only available for Linux systems.
- Ubuntu no longer supports Xen directly after version 8.04, i.e. there are Xen packages available for this and earlier versions in the repository for an easy install, but for later versions the procedure is more complicated.
- Xen is not only installed in a system; it is integrated into a compatible Linux kernel. Stock kernels usually do not have this compatibility, which means that a compatible kernel either have to be downloaded and installed or actually compiled from scratch and then installed in order for the system to be able to accommodate a Xen setup.
- If your system is not supported, Xen has to be installed manually in a number of steps: A binary package has to be downloaded and unpacked, and the source code within must be compiled using a Makefile. A compatible kernel must be acquired somehow. Details on how to do this will follow below.
- If your system is supported it should be no more difficult than running a few simple bash commands and the rest should sort itself out, with a Xen hypervisor system as a final result.
- Most Linux versions, and even Windows in the case of full virtualization, have the possibility of running under Xen as guests.
- Guests can be one of two types: fully virtualized or paravirtualized.
- Fully virtualized guests requires special hardware, Virtualization Technology must be built into the processor such as Intel's VT or AMD-V.
- Paravirtualized guests also need to be modified.

The way to a working Xen/Ubuntu system was long and many errors and mistakes slowed the progression. For clarity's sake any errors will be described below in a section of its own and the general steps taken to install Xen, as well as decisions taken and discoveries made along the way, will be documented first.

A computer system, preferably with capable hardware, was needed. Two machines were actually available; one was borrowed from the university and the other belonged to the author (specifications are available in the appendix). Several attempts were made with both of them, but the computer belonging to the university had inferior hardware and would later be used to test a version of Ubuntu

with proper support of Xen, i.e. 8.04. On the other system however, Ubuntu version 10.10 was already installed and Xen version 4.0.1 could be installed using the Ubuntu Community guide. Several tools such as updated compilers and code libraries are needed to install Xen, but these are listed on said web site and will not be described here. If these have been acquired the Xen package should be downloaded; the file is a tar ball which should be unpacked in a directory of choice, preferably one with proper access rights to avoid errors. The step that will install Xen is fairly simple: using a command console a number of commands from the included *Makefile* should be run within the newly created Xen folder (e.g. *xen-4.0.1*), preferably with administrator privileges. And that is all that is needed to install Xen. What follows next is perhaps a bit more complicated. A Xen-compatible Linux kernel was needed and to get it, one was compiled from scratch. It is not as simple as just copying the existing kernel, make modifications to it and then recompile the changes and use it. In this work a *git tree* was used. With it, a stable Linux kernel with architecture capable of running paravirtualized on a hypervisor can be found and downloaded. Before it can be compiled certain parameters must be set in the kernel configuration file to make it fully compliant with Xen. Details about these parameters are available in the appendix as well as on the official Xen homepage. This newly configured kernel was then built and installed. The next thing that needed to be done was creating a boot option with this new kernel and Xen in the Grub menu. Grub is the boot loader from the GNU project and is used in most Linux systems to specify which operating system to choose, if more than one is present. Users can change the Grub configuration file in order to create new entries or change the existing ones, i.e. adding different parameters for the operating system, to suit their needs. The newly configured kernel will most likely be added to this file automatically, as was the case during this work, but in order to run Xen along with it a new entry had to be created. This new entry would have to load both the new kernel as well as the installed Xen module. When this was done only a reboot was needed and after choosing the right Grub entry it could be verified that Xen was running properly.

2.1.1 Creating and Running a Guest Operating System

The easiest way to install a guest (or domU in Xen terminology) on a Xen system proved to be using xen-tools. This is a stand-alone release of scripts which was created in order to make Xen management easier. With it all that was needed was to specify a number of installation methods, such as what distribution to install, the online address to get the files from and the desired IP-address. What was then created was actually an *operating system image*. This could be started as a guest by Xen. A graphical utility manager was not installed, and so the only way to use these guests was to log in via a terminal window.

2.1.2 Errors Encountered

Numerous errors were encountered while attempting to install Xen. Some were due to incompatible hardware and/or software, but a majority of them were due to the author's inexperience with Linux systems, its software and various strategies as well as configurations and terminology. Here follows a list of errors (and sometimes mistakes) encountered, with an extensive description of major topics.

- Trivial as it may sound, it took some time to get used to using Linux from a terminal. Figuring out different commands such as how to manipulate access rights or searching for files as well as getting to know the file structure of the system was an initial time consuming process. As this was getting easier however, the work was starting to speed up and it was getting a lot

easier. Many tutorials followed assumed a great deal of knowledge in many areas, and blindly following any instructions without proper research would result in major failures.

- It was not certain at first if the borrowed computer could run Xen properly. When it was discovered that its processor did not have built in hardware virtualization support it was abandoned and the installation was started anew on the other machine.
- There was also some confusion over whether or not the version of Ubuntu at hand could run Xen at all, and installing an earlier version seemed like an attractive option. The attempt was carried out on the author's computer however and the one that had been borrowed was later used to try out said earlier version. Using a later version of both Ubuntu and Xen was motivated further by the need to install RT-Xen, a process discussed further ahead.
- Editing the Grub configuration file was also associated with some difficulty. First of all the syntax of Grub entries, and the parameters used, was almost entirely unfamiliar, so these had to be researched properly and sorted out so as not to make any mistakes, which might have caused a kernel to become unbootable. Linux and Grub uses different syntax when referring to partitions, e.g. Linux is letter-number based, starting the partition index at 1. Grub uses digits only which start at 0. Also, in this work there are appeared another special variant which could be solved only by copying other entries syntax. Later versions of Grub, such as the one used, even configures the entries by itself without allowing user changes to be made. This problem had to be solved as well before the new kernel could be booted without freezing on error messages at startup.
- Acquiring a Xen-compatible kernel was not an easy task. The *git tree* method applied was a great help, but the progress was hindered because the git-client was running slow and in the end aborted. What happens is that a huge amount of data is downloaded that is used in order to decide what kernel version should be used. Despite a fast connection speed this would sometimes take hours before it would abruptly stop uncompleted. The error is believed to be that the git information was stored in a folder with special access rights, which would not allow all the information to be written into it. It is believed that this is the error because when the data was stored in a shared folder the client would always succeed. Another issue associated with getting a kernel was to get all the parameters right. There is a list on the Xen web page with options that should be present in the kernel configuration file in order for it to work with Xen. These could not be found in the interactive configuration menu at first, and so had to be inserted by hand. After repeating this task a few times, which had to be done in order to try out different setups, the proper procedure was discovered. An option would first have to be inserted manually into the file which would cause the other necessary options to appear in the interactive menu. Verifying that these options are present in the configuration file was a wise choice nevertheless, since they can be hard to find in the interactive configuration or sometimes are simply not there at all anyway. As stated, Linux kernels were compiled and recompiled several times which was incredibly time consuming. Once or twice the compilation would be aborted with an error which was often due to a configuration mistake or some other unknown error, which was solved by simply creating a new configuration file.
- An issue that caused a great deal of problems was networking. First of all it should be mentioned that initial work was carried out with a system connected to a wireless USB network card. The later versions of Ubuntu shipped with compatible drivers so the Internet could be reached with little to no problems. This was not the case for earlier versions

however, which failed to recognize the card. Trying out the Xen Live CD requires a connection on device eth0 (using a cable) so this was not an option at all. This would mean that any guest operating system would not be able to connect to the Internet either. After much hassle a broadband subscription was purchased, allowing a substantially more stable internet connection. Second in the line of networking troubles was the way it is handled in Linux. The author's knowledge of the system again proved insufficient when the networking capabilities had to be configured manually for each of the guests. This issue remains unsolved, as networking on either of the guests still has not been functioning properly.

- When Xen was finally installed successfully there were some issues with installing the guests. The networking problem is mentioned above but difficulties arose even before the guests were installed correctly. One prominent error was that a nonexistent file system file was specified. Guests store their directory hierarchy in a large file located on the hard disk drive of the host along with other files, and at first the guest would fail when starting up because a file like this had been specified which did not exist. The solution was to find the directory where such files would be stored and simply choose another existing file for storage.

2.1.3 Analysis of Xen

Xen is a very popular hypervisor, mostly because it is open source and has a good code structure and architectural design which makes it comparatively easy to modify it or add custom made code. It is because of this that most experiments of combining virtualization with real-time is carried out with this software, and that the platform is promising for any such an attempt. It is also the cause of most of these projects being an actual modification of the source code, which in just about every case means that Xen has to be reinstalled in order to use these changes. Xen performs well, but this performance is because it is integrated into a compatible Linux kernel which may not be a desired solution since there is always a risk of various errors or compatibility issues. To complicate matters further, not all Linux distributions are actively supporting Xen, making an installation in such a system even more difficult. For the sake of honesty we should mention once again the author's inexperience with the Linux system and its management, but while this caused some errors on its own, many of the troubles we encountered were beyond our control entirely.

2.1.4 Xen Live CD

In addition to the possibility of installing XEN on a computer you can use the live CD. This CD has been developed by the community [39] and offers a chance to test out XEN in a pre-configured environment. What you get is Debian Lenny 5.0 as host OS and two Ubuntu 8.10 systems as guests. This can be run on most desktop computers and offers a flexible choice of simply trying out the different options available for virtualization in XEN.

2.2 Installing RT-Xen

Once a working Xen-Ubuntu system had been established the possibility of running RT-Xen was explored. This would prove to be a difficult task indeed and, as of this writing, has still not been completed. Why this came to be, along with details about the problems and the solutions attempted, is described below.

The creators of RT-Xen recommend using the same Linux distribution they did: Fedora 64 bit version. Since Ubuntu was the preferred test platform the fist attempts to install RT-Xen were made with that setup. RT-Xen is applied as a patch which only makes changes to the Xen installation and because of

this we reasoned that a different Linux distribution should make little difference. This thinking was challenged when the first attempt at running RT-Xen did not work. An unknown error was causing the booting of the Ubuntu-Xen configuration to fail, which in hindsight was most likely due to a mistake in configuration of either the Grub menu or Xen. This led to making an attempt with Fedora.

2.2.1 RT-Xen on Fedora

Installing Xen on a Fedora system was no easier than on Ubuntu, despite the experiences gained which to be fair made the process faster than otherwise. Installing Fedora was trivial; version 14 was used to get the latest updates. Again an online guide to installing Xen was sought out. The one recommended on the RT-Xen page was tried first [40], but when it proved to be complicated and inefficient (Xen had trouble starting at all) a tutorial available on the Xen page was used instead [41]. The procedure was much the same for Fedora as for Ubuntu, and the number of errors and mistakes was fewer this time around since the approach had been repeated a few times already. In the end it was decided to switch back to Ubuntu since the Xen configuration in Fedora seemed to have some trouble with the hardware; A Xen compatible kernel would not boot correctly, displaying error codes in hexadecimal form and messages about the graphics card. There was no hope to be able to understand these errors and so a tried and tested method was used once again: Ubuntu 10.10 with Xen.

2.2.2 RT-Xen on Ubuntu

When it was decided that another attempt should be made to install RT-Xen on an Ubuntu system such a setup once again had to be configured. Keep in mind that all attempts were carried out on the same machine, with a hard disk drive wipe required every time. Once Xen was running again there came the issue of the RT-Xen installation. Instructions about how to do this were not very detailed which caused some confusion The information available stated however that all files downloaded from the web site should replace the ones corresponding to these within the Xen folder, and all that needed to be done then was to install Xen again, i.e. recompile all the parts using the Makefile mentioned above. An RT-Xen system was very close to work before the project was abandoned which was due to errors that kept occurring despite much effort and even outside help. This help came in the form of e-mail exchanges with one of the RT-Xen creators, Sisu Xi, who deserves some credit in contributing to this work. Every error listed below was solved with his help and by his suggestions, save for the last one which marks the point where the project was stopped. The following problems are listed in chronological order.

1. RT-Xen could be added and Xen could then be compiled without error but the configuration would not boot. The startup sequence would stop and refuse to proceed. It was obvious that RT-Xen was causing the error since an unmodified version of Xen was able to boot. The suggested and working solution was to alter and add some parameters to the Grub entry. After these changes RT-Xen would boot successfully.
2. The next error that appeared was somewhat peculiar: While it was obvious that Xen was running, none of the built in RT-Xen commands would give any response, e.g. for changing scheduling method or parameters. It was stranger still that a Xen command which would display what scheduling was used showed that it was indeed using Sporadic Server scheduling, a feature not present in the original Xen but introduced by RT-Xen. This issue was never solved because at that precise time an update of RT-Xen was released, and it was recommended to try this one instead.

3. The first attempt to compile the updated RT-Xen, version 0.3, did not work. The advised solution was to give full access rights to one of the folders in the xen-4.0.1 directory. This did remove the previous error but revealed another one.
4. According to the compiler a file could not be processed because it contained illegal characters. While the advice given still pointed towards access rights the problem remained. In the end the file was found and when examined proved to contain nothing but binary code, and not at all any python code which was what the compiler assumed. Coming to the conclusion that this was a waste dump file, it was removed and the next compile attempt was one step closer to success.
5. It should be mentioned that this new version of RT-Xen came in the form of a large package containing almost the entire directory structure of the original Xen folder. The next compile error that appeared did so when one such specific directory was the compile target. When it was learned that this directory was not affected by RT-Xen in any way, it was decided that it simply should not be replaced. This finally resulted in a successful compile of RT-Xen.
6. After this latest successful compile, a boot with the new configuration was attempted; the grub entries had been modified as recommended as well. The error that appeared next was to be the final one which caused the project to be stopped. What happened was that the system froze and shut itself down right after booting into Ubuntu and displaying the desktop. It would seem that everything had started up just fine, but some unknown error caused the entire system to crash. A suggested solution is available but has not been tried because of reasons mentioned above, and this was to connect another machine to the computer in question in order to get proper debug output.

2.2.1 Analysis of RT-Xen

The RT-Xen project presents a promising and proper solution to the problem of combining real-time systems with virtualization technology. The framework is performing well and some interesting test results have been produced, indicating that this might be a solution worthy of further development. It should also be noted that it is one of very few solutions that uses hierarchical scheduling in a virtualized environment.

However, as we have shown with this work there are some problems associated with the way RT-Xen is implemented. We attempted to install this framework and encountered numerous issues throughout the process, as can be seen in our documentation in previous sections. Just about all the problems were encountered when the RT-Xen patch was to be applied to the original Xen installation. Even though several different setups and configurations were tested a working system was never achieved, even though the creators of RT-Xen reported successful compiling and running instances. This can be blamed on incompatible software and hardware issues, but also on the framework itself having narrow implementation target without fully tested support for all kinds of systems. From these experiences we assume that the modifications of stock as well as other software bring with them an increased risk of errors and issues preventing effective testing and performance and the cause of many obstacles in a research environment. If we also consider the fact that the Xen software itself needs a modified kernel to run we claim that the design as a whole has the potential of being unstable to run and cumbersome to set up. From the aspect of real-time performance, RT-Xen should most certainly be one of the favored choices while simultaneously being a difficult system to handle. Once again the issue of any skills with Linux comes to mind, but this time as well as during previous attempts the majority of the errors were the blame of the software or hardware.

2.3 Other Solutions

There are very few solutions available that lets virtualization be combined with real-time functionality in an effective way. Fewer still does this successfully and out of these only two or three are available as complete projects. The following sections will examine some of these solutions; some are fully fledged systems with impressing performance, while others are shown to not even present a proper solution. Most however are merely promising suggestions that are not fully developed. An analysis of each solution is given after its presentation.

2.3.1 Performance Analysis towards a KVM-Based Embedded Real-Time Virtualization Architecture

While Xen is more than often the platform of choice to develop new strategies, there exist a few suggestions based on its rival KVM. This work [17] focuses on the problem of letting two kinds of different systems share the same hardware: A partition running general purpose tasks and one with real-time requirements, a problem that is common in mobile phones. Actually, it is at this kind of problem that the research is aimed; to find out whether the following setup is suitable for mobile phone devices, making the real-time classification soft, since failure will only result in degraded performance. The guests are not modified but are instead chosen specifically for their task, with a Linux system for the general purpose tasks and VxWorks as the second operating system. They run as-is because they are already adapted to their purposes. The changes are however made to the host operating system. KVM is available in most Linux distributions from the start, and is really only used to supply the virtualization technique while Linux itself is left to do the heavy work such as scheduling. With this in mind the project made changes to the CentOS 5.4 host to better make it suitable for real-time scheduling. The first change was to make sure the general purpose operating system did not cause interrupts that would disturb the real-time operating system. This was done by applying a real-time patch for Linux systems [45]. *CPU-shielding* was also introduced, allowing a CPU core to be dedicated to a single process, in this case the real-time operating system, in order to reduce interrupt latencies caused by task switching. Both tasks and interrupts was bound to specific CPUs if needed, using the system call *sched_setaffinity* or a shell tool *taskset/cpuset* for tasks while interrupt binding was set with the help of a user interface via files (*/proc/irq/>irq_number>/smp_affinity*). A power management service (ACPI) for the processor was also disabled in the kernel configuration so that its response time would not be affected. As stated, the host operating system was a modified CentOS 5.4 (RT-Patch) and the real-time guest was VxWorks 5.5. An unmodified CentOS 5.4 was used as the general purpose operating system guest as well.

2.3.1.1 Analysis

While being a simple and lightweight solution due to KVM, this solution only uses relatively simple scheduling methods with no concept of hierarchy. The real-time adaption of the operating system are few but substantial, especially considering the kernel patch. Modifying the kernel may be considered insecure however. While it is being supported by a large community this is no guarantee that it will work flawlessly and, as can be learned from our experiences, the obligatory recompiling and subsequent testing for instability is very possibly still present. There is some relief in the fact that guests do not have to be modified though. It is an interesting experiment because it is directed towards the emerging problems of the growing smart phones market. This is an attractive solution because it considers the growing problem of different types of systems on one platform and introduces a simple way of letting them reside in the same system without modifying them. No

particular scheduling scheme is added though; the real-time patch for Linux only introduces preemption, no new scheduling algorithms. This means that real-time is achieved using only the standard scheduling in Linux, along with a few preemptive tweaks. While an attractive solution, the question remains whether or not it is advanced enough for real-time demands.

2.3.2 VSched

VSched is in fact a user-level tool that interacts with a Linux kernel in order to act as a scheduler. It has only really been tested with the type 2 hypervisor GSX Server from VMware, but the authors of the VSched article claims that Virtuoso should be able to run with any type 1 or type 2 hypervisor, provided the virtual machines are run as Linux processes. In order to install VSched its source code must be compiled by hand. It was developed with several different types of workloads in mind: interactive and batch workloads have already been described above, and a third type, batch parallel workload, is much the same as batch except that it can be worked on by an arbitrary number of additional virtual machines.

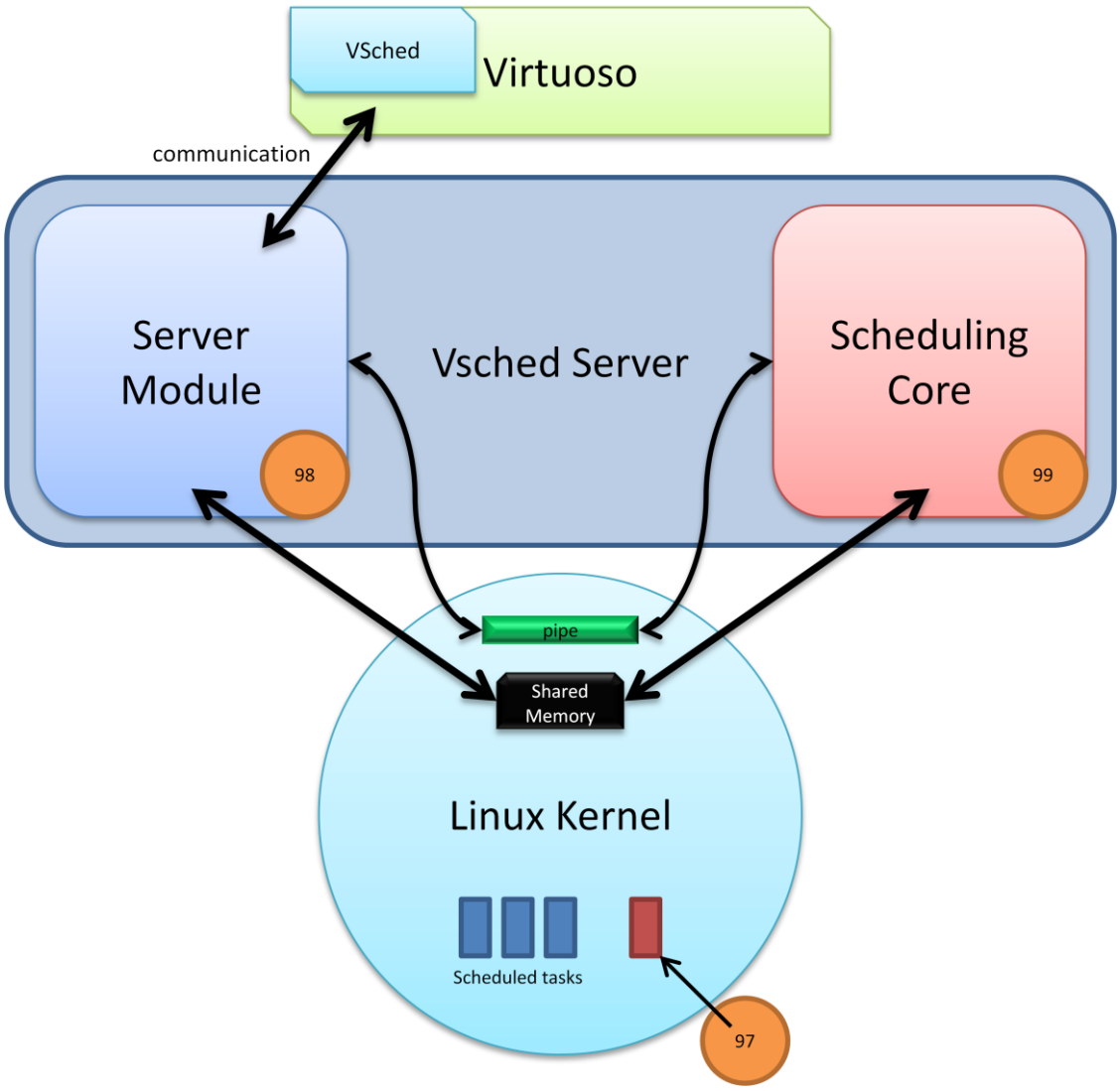


Figure 4: The different parts of VSched and how they interact. The numbers in the orange circles signifies the priority for the appointed task. The "pipe" and "Shared Memory" blocks lets the server and the scheduling core communicate

VSched is running as an application of its own, which means no other modifications are needed, with two separate parts: a server module that creates the scheduling core process and communicates with the VSched client and Virtuoso, while the scheduling core monitors the active and inactive tasks and decides how they should be scheduled (See Figure 4). The scheduling algorithm that is used is called Earliest Deadline First (EDF), meaning that the one with the shortest deadline will have the highest priority. In the system, each task is given properties that are used to decide which one should run first. VSched does this at regular intervals, which makes it a dynamic scheduling approach and quite unusual because of it. Running tasks can also be interrupted by higher priority ones, so the concept of priority preemption is also present. VSched is a part of the virtual machine tool Virtuoso, which has been developed at Northwestern University in Chicago, USA, with the purpose of being able to handle both trivial tasks such as word processors and computer games, as well as computationally heavy assignments such as simulations and calculations. It is supposed to be able to do these things simultaneously without loss of performance. VSched was developed for this application in order to be able to provide reliable scheduling for real-time demanding tasks. In order to guarantee that VSched always has complete control over and knowledge of the available tasks it is given the highest priority in the system. This would be priority 99 under the scheduling policy SCHED_FIFO in Linux, which is assigned to the scheduling core. The server module is given priority 98 and the task that has been set to run is marked with priority 97. This is how VSched uses the scheduling of Linux, it does not implement any new scheduling algorithms per se. VSched is given the highest priority which could be assumed to be a problem because it might steal the CPU from real-time tasks, but this is not really a problem due to the fast and effective EDF algorithm that is used to decide the scheduling. VSched uses this algorithm to decide in which order the tasks should be placed in the run queue by using the parameters *slice* and *period* assigned to each task. The *period* parameter is how often the task must run, it will run every *period* seconds, and *slice* is for how long at a time it will run, for example it will run for *slice* seconds. These values are inserted into the formula

$$U(n) = \sum_{k=1}^{n} \frac{slice_k}{period_k} \leq 1$$

The resulting *U* will show the total utilization of the tested task, i.e. how effective this set of schedulable tasks *n* will be. This test is done at period boundaries, which is the only time a task can miss its deadline making VSched run only as often as the task with the shortest period. Since the tests are done at regular intervals to decide what task that should be allowed to run, VSched uses a dynamic scheduling approach and it is quite unusual because of it; most solutions are entirely based on static strategies. Running tasks can also be interrupted by higher priority ones, so the concept of priority preemption is also present. This process is somewhat simplified by the fact that the deadlines are assumed to be the ends of the periods, and these have already been defined before the system is started. There is also no priority inheritance mechanism or bounded interrupt services. Tests were performed with a Mandrake Linux 10.1 as host, along with the virtual machine manager VMware GSX Server 3.1 with two virtual machines: one running Windows XP (as the interactive virtual machine) and the other Red Hat Linux 7.3 (the batch virtual machine). Most VMware virtualization products support full virtualization so the guests should not have to be modified for paravirtualization. The interactive VM was tested by playing MP3 audio and MPEG video files as well as computer games, and browsing the Internet. The batch VM was only started every ten minutes, and during its one

minute run time it was set to consume CPU as fast as possible while reporting on its progress over a network connection. The results show that the best configuration was to use 5ms for *period* and 1ms for *slice*, where the interference was minimal. VSched guarantees soft real-time functionality and runs best on a Linux kernel version 2.4 or 2.6. The lack of any hard real-time guarantee is said to be because of limitations in the Linux kernel; there is no priority inheritance mechanism or bounded interrupt services, but according to tests VSched performs well despite this with very few tasks actually missing their deadlines [12].

2.3.2.1 Analysis

VSched is an attractive solution to the problem at hand, mostly due to its lightweight implementation and relatively easy usage and management, which it achieves by using the scheduling mechanisms of the Linux kernel. This approach could be considered as something of a drawback since it is actually Linux that limits the scheduler and prevents it from achieving actual hard real-time performance. Currently, only soft real-time performance can be guaranteed. This is by no means a bad thing however; technology is advancing fast and soft real-time media devices are becoming increasingly more common, and because of this the demand for new and clever solutions is great. And this *is* a clever solution indeed. Getting two-level EDF scheduling to work on a system such as a virtualized environment without any modifications to either the host operating system or the hypervisor software is quite the feat. And the fact that it only operates on the user level of the system does not make it any less impressive. As has already been stated the application has only been tested and even developed on a single software platform, and as such the problem of complete compatibility with all software and hardware might still be present. Despite this, the hassle of modifying the kernel has been avoided which is a huge plus. Summarized, VSched is an interesting solution that performs well and a design like this may be the answer to the problem of combining real-time scheduling with virtualization.

2.3.3 *Laxity* in Xen

In this work the default scheduler in Xen is modified to support soft real-time tasks in the virtualized environment. Things such as media based applications are mentioned, to give an idea of the goal. It is designed as a user-level application, making it lightweight and supposedly easy to manage. When it was found out that the host of the virtual machines was effectively stealing CPU from the guests, changes were made to enable Xen to better accommodate soft real-time tasks. These modifications were made to the standard credit based scheduler in Xen; a *laxity* parameter was defined for every task to allow the scheduler to recognize which tasks that needs real-time priority, and insert them into the run queue to make sure that their deadline is met. To clarify, it is the credit based scheduling strategy that is used which distributes CPU time to the tasks; no other kind of algorithm was included, although some were considered. Starvation caused by the credit scheduler should have been avoided because the strategy does not affect its original distribution mechanism, yet the problem would arise in the system after the modifications had been added because the real-time tasks run periodically and hence would continually steal the CPU away from the lower prioritized tasks. Yet another feature had to be added to increase the performance, namely that of a fine-tuned load balance algorithm. This was solved by allowing CPUs to steal high priority tasks from each other, effectively balancing the workload. Another issue was resolved as well, namely that of cache thrashing that appeared when tasks were repeatedly being switched between CPUs and the reduced real-time performance that this caused. The solution introduced was not to allow a task to be stolen if it was a real-time task, effectively binding them to a specific CPU. While testing shows that this

design is successful, the modifications are intended as a proposition only, and are therefore not available for a wider user base [10].

2.3.3.1 Analysis

While it is only aimed at soft real-time performance, and most capable of guaranteeing such performance, the concept of introducing a *laxity* parameter for virtual machines in Xen is an interesting one, as its design is intricate enough to allow real-time tasks to run properly alongside ordinary tasks under the same hypervisor. Once again it is highly relevant because of the booming handheld devices industry. The choice of using Xen for this platform could be questioned however: Would an advanced piece of software like Xen with its many features and large codebase have a negative impact on the relatively lightweight hardware of such systems? This is likely the case, but nothing is really lost by exploring new options and ideas. This particular idea however, is not using hierarchical scheduling, which would be desirable; only the virtual machines themselves are given the special real-time treatment and although this treatment may be a clever solution indeed it is uncertain whether or not it will be able to reach the full potential needed for real-time scheduling. The solution is not an extensive project either, existing only as a suggestion and as such it is not available for testing, which is not very useful. Since the modifications are made once again to existing software, requiring a patch and very possibly a reinstallation of the program, the danger of incompatible software and untested code is once again present.

2.3.4 Real-Time Enhancement for Xen Hypervisor

For this work [16] it was argued that the default credit scheduler in Xen was unable to handle guests running in real-time configuration, and it was proved by running tests with this kind of setup. The goal was to make Xen able to handle real-time operating systems, such as RTLinux, and make it achieve performance that could accommodate embedded devices-like software. To reach this goal a number of modifications was made to the hypervisor which would mark the real-time guests so that they would be placed at the front of the run queue and make the scheduler aware that it had guests that needed special attention. Once again it is Xen's stock scheduler that is modified, the credit scheduler, and while it still distributes CPU time it is made aware that there are real-time tasks present with certain scheduling requirements. The concept of preemption had to be introduced, since virtual machines waiting for input are blocked until one such arrives and then placed amongst other running virtual machines, without privileged places. Another issue that had to be remedied was that of conflicting real-time virtual guests. This was solved by making the scheduler able to balance multiple real-time guests; if a run queue contains more than one real-time guest one is chosen and moved to a parallel run queue with no guests of this type. This is somewhat reminiscent of the *laxity*-project where virtual machines are switched between different physical CPUs.

2.3.4.1 Analysis

Also a modification to the Xen source code, this project is interesting as well due to its thoroughness with real-time priorities and the ability to balance different quests to even the workload between run queues. It is once again the stock scheduler that has been changed, the credit scheduler has received some modifications and additions, but regrettably no hierarchical scheduling is used which would indicate that hard real-time performance cannot be guaranteed. Due to Xen's nature, these changes most likely require that source code files be replaced or changed, and to use these modifications the entire Xen installation will have to be done again; the code files would have to be compiled and installed.

2.3.5 Research of Real-time Task in the Xen Virtualization Environment

The need for a two-level scheduler is not quite satisfied with this solution. Hierarchical scheduling is not used, but instead a message system is implemented which should help performance since it will increase awareness of different needs in the system. This is done by introducing a solution based on message passing where the Xen scheduler that handles the VCPUs is synched with the real-time scheduler, making sure that the real-time tasks get the attention they need. While the system is stable, nothing is said about any potential delays caused by the design itself [15].

2.3.5.1 Analysis

This solution, where the base scheduler actually has a way of checking the contents of its guests, is most certainly a step in the right direction of being able to assure real-time performance in virtualization. The changes made would have to be quite extensive however, affecting many parts of the system besides the hypervisor itself. There is also the potential issue of overhead; if messages are continuously sent to and fro is it not possible that some loss of performance should occur?

2.3.6 A Real-time Scheduling Mechanism of Resources for Multiple Virtual Machine System

In this work two scheduling algorithms for real-time performance in a virtualization environment is suggested. They were both tested using Xen, but it is proposed as a solution for any general hypervisor. The design is similar to the previously mentioned implementation in that it receives information about the tasks at hand from the virtual machines. The Virtual Machine Management (VMM) method collects information about the different tasks available, specifically earliest start time and earliest finish time parameters are examined. This is sorted into a list based on the information, and a *single image management system* examines these and decides how they should be scheduled. The Processor Selection algorithm works in a similar way, but makes a more refined prediction of the execution time by using parameters such as the current workload and different computing weights as well as results from earlier trial and test runs [11].

2.3.6.1 Analysis

While not a hierarchical scheduler in any way, this kind of approach should be considered a step in the right direction; letting the base scheduler know more specifically what is needed for the virtual machines should lead to more precise scheduling. Getting such a scheme to work would nevertheless be an issue; once again the stock software must be modified which may or may not be an easy task. The complexities of the proposed algorithms are also an obstacle; there is a great deal of computing having to be done which is time consuming and perhaps not the cornerstone of an optimal approach.

2.3.7 Using Microkernel-based Virtualization in Embedded Systems

Heiser [7], Kaiser[8] and Aguiar and Hessel [1] have all made suggestions of a apecialized architecture with a microkernel as base for the much apparent need of virtualization in embedded systems. They argue that microkernels have an advantage over traditional virtualization due to their lightweight construction and the lack of obstructing functionality. A related invention is the microvisor from OK Labs, which in fact is very much like the technology that has been suggested. Such functionality have however only been properly evaluated by Bruns et al. [4]. They reached the conclusion that only tasks with short deadlines or work requiring large amounts of thread switching had any real effect on the performance; the overhead caused by the microkernel did little to disrupt the real-time tasks. Cache resources were also a great problem, and so the suggested solution was faster processors with bigger caches and improved cache predictability techniques.

2.3.7.1 Analysis

The suggestion is something that needs special manufacturing and customization, but it is also only aimed at a specific market; the new generation of combined real-time and general purpose systems that consists mostly of smart phones but also of similar emerging devices. It is an attractive solution that would be great to use in other hypervisors, but since the whole concept is that these are much too sluggish and bloated with software features to be able to manage a practical implementation of said concept, it is better to use simpler but sturdier and more effective software and hardware as a foundation. The design is narrow and only usable for the particular field of modern hybrid embedded devices, but if that is where interests lie then these suggestions seem to contain much promise indeed.

2.3.8 Real-Time Systems

Real-Time Systems is a company that deals mostly with different real-time product solutions, such as virtualization. They have developed a hypervisor guaranteed to handle real-time operating systems without fault. This is done, not by a particular scheduling method, but by implementing two different modes for the operating systems to run in. The first of these are an unmodified operating system running on an isolated and secure partition, from where it is not supposed to interfere with any other system while at the same time having degraded performance. The other mode is supposed to guarantee hard real-time performance by allowing a paravirtualized guest to have full hardware access, without being hindered by the hypervisor [42].

2.3.8.1 Analysis

While guaranteed real-time performance is nice, the methods of achieving them are certainly a topic of debate. Releasing the hypervisor's control of one of its virtual machines is a questionable measure, since in effect this means that the guest is no longer running on top of a hypervisor and therefore it could be argued that it is no longer a virtualized guest system at all. It is reliable, but it breaks the concept of isolation, and allows a guest privileged access to the hardware which goes against the idea of virtualization.

2.3.9 Xtratum

Xtratum is an open source bare metal virtualization technology, i.e. a level 1 hypervisor, developed specifically with real-time performance in mind. Developed at a university in Spain, Xtratum is mainly aimed at safety critical systems such as satellites and uses paravirtualization to achieve its performance. It is capable of running modified ordinary operating systems as well as simple applications directly on the hypervisor at the cost of some changes to the software source code. Applications can of course run in original condition on an operating system if there is one such installed on a virtual machine. Xtratum is based on static pre-defined knowledge of the tasks that are supposed to run, the system is based on the concept of predictability, and so no changes can be made to the system once it is running. This means however that it can implement a simple fixed cyclic scheduling mechanism without preemption at hypervisor bottom level, that lets the virtual machines run in the order and magnitude that it knows that they need. The guests in turn use a fixed-priority preemptive scheduler that schedules their local tasks in a regular and predefined pattern. Xtratum does not have a host operating system per se, but because some way of monitoring and controlling the hypervisor is required an interesting alternative is introduced: guest partitions can have different status, where a supervisor status allows the guest to modify and monitor the virtualization environment. It should be noted that the much sought after isolation is preserved

despite this elevated status, and also that there is no difference in scheduling between partitions with different status. Concerning the isolation it has previously been stated that embedded systems are highly integrated and need to communicate. Xtratum has considered this and provided all partitions with a message system based on specific access ports, allowing them to communicate with each other or the hypervisor if need be. Xtratum is available for download as an installation as well as a Live CD. It is however only available for the LEON2 processor architecture, with a recent port to Sparc V8 [6] [43].

2.3.9.1 Analysis

The Xtratum project seems to be a well thought out solution to a very important problem. No performance results are available, but since the project is collaborating with space satellite programs it could be assumed that the hypervisor is performing well. Its strength is definitely the tailor made architecture, which makes sure that performance is fine tuned, along with some clever solutions to the problems with combining real-time systems with virtualization technology such as communication and monitoring. This predefined design could also be considered a weakness however; no software with a purpose different than the one which the creators had in mind could run successfully on the hypervisor, limiting its usefulness somewhat. Combined with the fact that Xtratum is a static system, with no possibility of changing the system once it is running, or running tasks that require dynamic scheduling makes it a somewhat limited platform. Much like the microkernel and hypervisor solution it should perform well on its targeted device, but anything outside its narrow spectrum should consider it an unsuitable solution. It should be mentioned that the latest news on the project is from 2009.

3. Conclusion

This is the final part of the report and here we will summarize the work that has been done, evaluate the result, and based on this evaluation we will present a solution of our own.

3.1 Evaluation

Now that we have seen the alternatives of merging real-time systems with virtualization techniques it is clear that there are certain properties of these alternatives that are either advantages or defects. Together with a study of both the virtualization concept and real-time scheduling we find that combining these technologies have a number of requirements and preferred features associated with them that may complicate the implementation of a virtualized embedded system. We would now like to present the results of our research and tests by listing the requirements we believe this kind of system would have, as well as give our opinion on whether these requirements are satisfied in today's available suggestions and solutions.

- **Existing Technology** – Many theoretic solutions are available, while there are few actual implementations. This would indicate that having a great idea is easy compared to the immense task that it is to put the idea into reality. A solution based on existing technology would be very successful in that it would be easy and fast to implement.
- **Compatibility** – Many available solutions are only aimed at a very specific area of operations, e.g. satellite systems, mobile phones or other very specific hardware architectures. It should be obvious that a solution with capabilities of running on any common computer system or in the context of any kind of task would be preferred, as it would be open to any type of user.
- **Combination** – Slightly related to the above is the need to be able to run not only safety critical real-time tasks, but ordinary general purpose tasks such as media players, games or word processors as well. It has been learned that the need for such functionality is growing fast these days, and an architecture where this is possible would be greatly desired. This is of course possible in most systems, but the issues of not letting these general purpose tasks disturb and possibly ruin the running real-time tasks has been found to be the real problem.
- **Stability & Reliability** – We have learned that real-time performance is heavily associated with the needs for a reliable system to avoid possible catastrophe. This would indicate that a platform as well as other software is required that have been properly tested and tuned for stable performance across a wide variety of different environments. As such, this requirement should be associated with the following ones, since a system that is easy to use is reliable in that the risk of error is greatly diminished, while a system that has not been modified is both stable and reliable because, as we have seen, any changing of the hardware can have unforeseen and undesirable consequences.
- **Usability** – A system that is easy to use would prosper in the worlds of research and industry both. This concept of easy usability should not necessarily be limited to the usage of a system that has been installed and are now running, but also to the installation process itself. With the experience of our tests, and the fact that we never actually got a working setup, we cannot stress the importance of simplifying this process enough. Usability could also be applied to a system that offers a wide variety of choices as to what specific applications and operating systems that should be used, letting the user choose his or her own preferred environment. This kind of system would be able to accommodate a wide user base.

- **Unmodified System** – We claim that the decision to make changes to the software base, be it the operating system or the hypervisor, is a fundamental one, that may very possibly cause instability. This is not the only time such a statement is made; Åsberg et al. [2] [3] emphasize the importance of stable distributions of Linux when embedded systems performance is sought. During our attempts to try the different methods available we found that the majority of the problems and errors we encountered were connected to the process of changing the existing software; it was all too often the thing to blame entirely. Through our research we have also learned that this is how just about all suggested solutions approach the problem: an existing kernel or hypervisor's basic functionality is changed, an operating system's scheduling mechanism is modified or mechanisms that are supposed to alter or interfere with the existing scheduling are added to either operating systems or hypervisors or even both. In addition to being difficult and costly to modify many of these designs, while released for public use and testing, have often been developed on a single platform, and different software or even hardware combinations may cause the entire system to become inoperative or crash completely. We think that any optimal solution to the real-time-hypervisor problem should contain as few modifications as possible. Preferably no changes should be made at all to the operating system, be it hypervisor host or guest, or to the hypervisor itself.
- **Hierarchical Scheduling** – Having studied different scheduling implementations that aim to enable real-time performance in a scheduling environment, we think that the most successful ones have the common factor of using hierarchical scheduling. Enabling real-time conscious scheduling in only one level of the partitioned system that is virtualization is clearly not enough to ensure the performance that real-time tasks demand. Existing hierarchical scheduling implementations, or similar solutions, are very advanced or complicated however, often bringing with them the need to alter or modify the existing scheduling components. While it is a substantial challenge, a design that uses hierarchical scheduling, is simple and does not affect other components would be something we consider a superior and optimal solution.

Before we move on we would like to motivate the use of hierarchical scheduling further: Realizing real-time scheduling in virtualized systems is associated with several problems. A major one is that of scheduling. Most virtual implementations use hierarchical scheduling, with a global and a local scheduler; the global scheduler often manages the virtual machines, or partitions, as nothing more than ordinary tasks, while the local schedulers manage their tasks as best they can in the time space they are given. This is far from optimal in a real-time perspective. If we have a situation like in figure 5 we can clearly see that with fair-share scheduling, where the global scheduler takes no consideration of partition-local tasks, there is a great risk of deadlines misses.

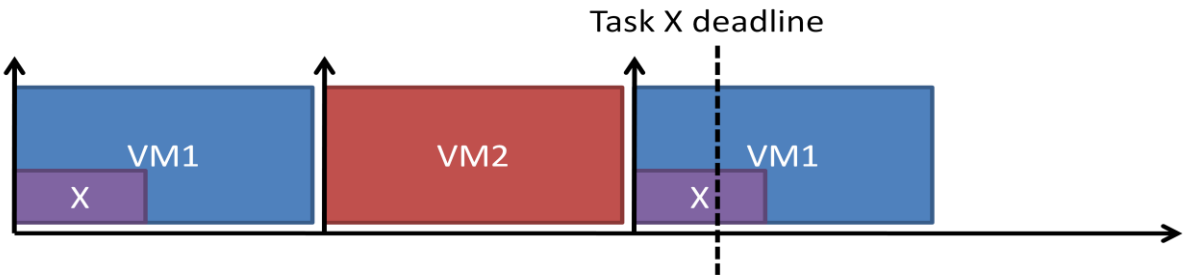


Figure 5: Task X misses its deadline because partitions are given equal share of CPU time, making VM1 unable to accommodate task X's requirements.

If however we were to introduce a hierarchical scheduling model that is aware of different partitions' needs and capable of preempting a running partitions like in figure 6, deadline misses could very possibly be avoided.

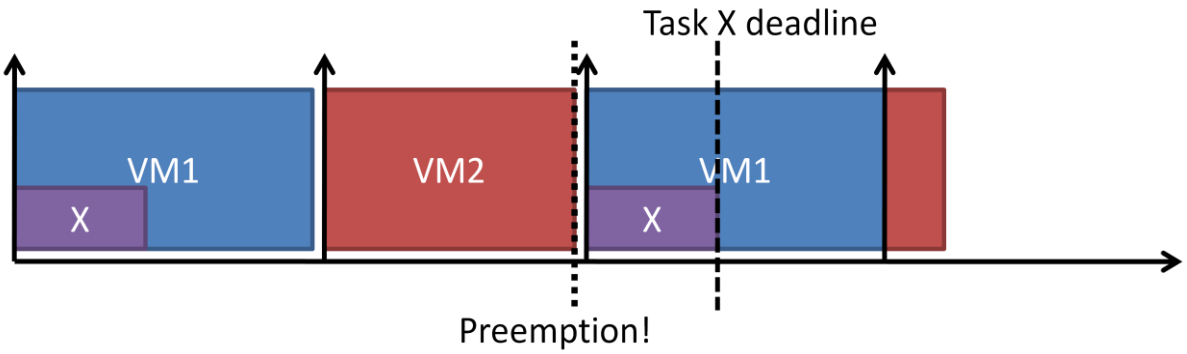


Figure 6: With a global scheduler conscious of the partitions' needs, partitions can be preempted to allow real-time critical partitions to finish.

Another thing to strive for would be usability and simplicity in installation of this kind of system. Setups like this, preceded by a trivial initiation process without too many and severe modifications having to be made to original system would definitely be preferred to increase efficiency. Also taken into consideration should be the fact that, as has previously been stated, real-time and embedded systems require stable and reliable hardware and software platforms.

3.2 Solution

We have seen what is being done to create a successful virtual system that is capable of real-time performance. We have also investigated what the available solutions contribute and we have used our research and experience to identify the elements that they lack. We will now present a solution of our own that takes these shortcomings in consideration and that is based on this research and our observations.

3.2.1 Suggestion of a Virtualized Real-Time System Model with Hierarchical Scheduling

What we are proposing is a design that deals with all of the problems we have encountered as well as fulfilling the real-time hypervisor requirements we have determined. This design is based on using the established, highly compatible and easy-to-use hypervisor VirtualBox running on a Linux Ubuntu system together with scheduling software that supports the loading of advanced scheduling methods that the user chooses. This software, named RESCH [9] allows a Linux user to change the scheduling

to whatever is desired without making any changes to the kernel or any other part of the operating system.

RESCH (REal-time SCHeduling) is a loadable real-time scheduler suite developed by Kato et al. [9] in order to achieve a solution of changing the scheduling mechanism without the hassle of patching the kernel and get away from the limitations of compatibility that this brings with it. RESCH's scheduling performance starts in user space in the application that is to be scheduled where an API call to RESCH's core in kernel space is made, which is made accessible by including the provided library. RESCH then communicates with the systems kernel and depending on what plug-in is used, what scheduling algorithm the user has inserted, the scheduling of that specific application is handled according to it, see figure 7 for a graphical overview.

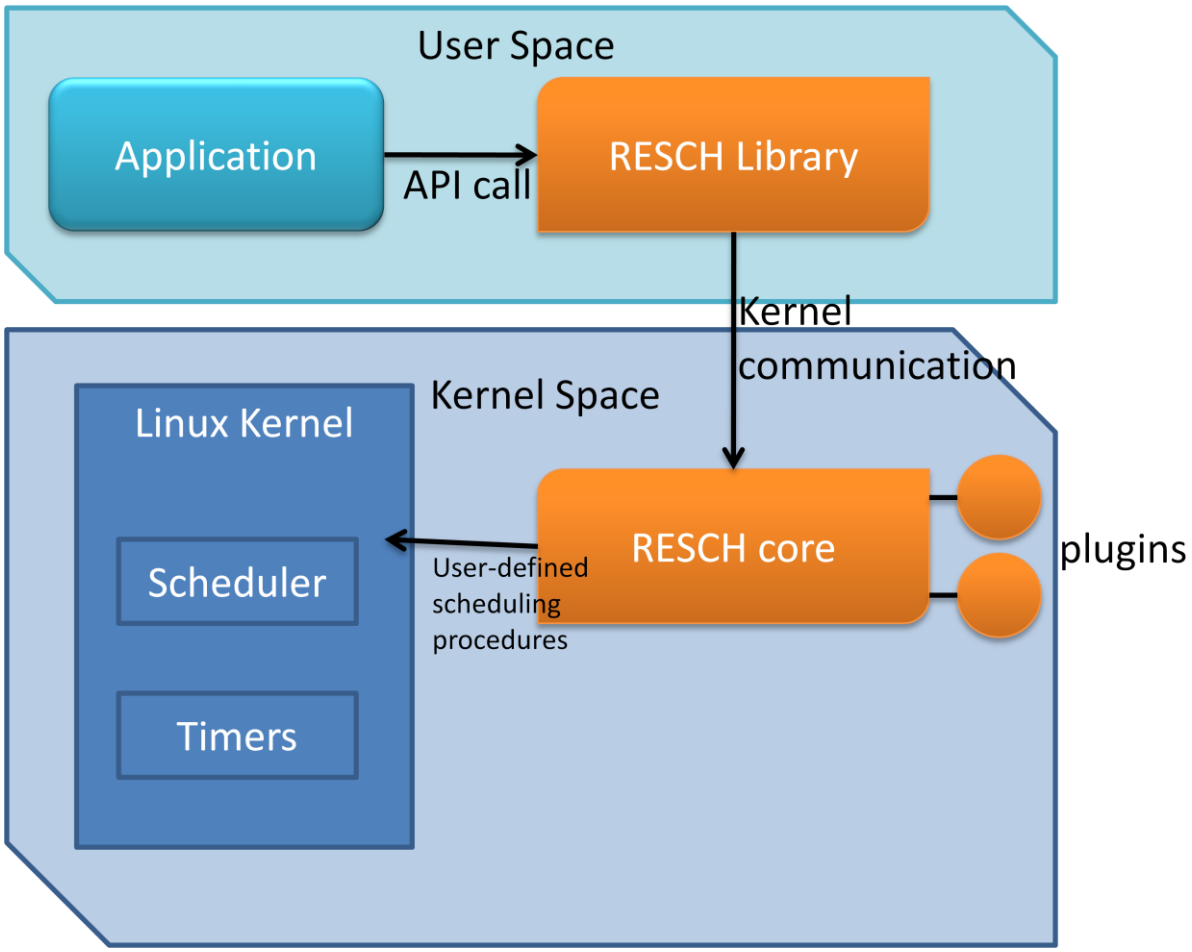


Figure 7: A graphical overview of the RESCH architecture

We suggest that the RESCH scheduling suite should be used on both levels; on the host operating system running VirtualBox as well as within the virtual machines running Ubuntu as well, resulting in hierarchical scheduling with full Fixed Priority Preemptive Scheduling (FPPS) functionality. It should be mentioned that [3] is related to this work and it is the framework described in that article that we suggest should be used along with RESCH. The framework in question is HSF (Hierarchical Scheduler Framework) which has been designed to enable precisely hierarchical FPPS. This is a static scheduler that assigns the properties *period, budget* and *priority* to the different tasks by using their process ID, allowing them to run for *budget* time units every *period* time units ordered by *priority*. Figure 8

shows an overview of what we have in mind. Notice the fact that the RESCH functionality is present in the same way within the virtual machines as it is in the base system.

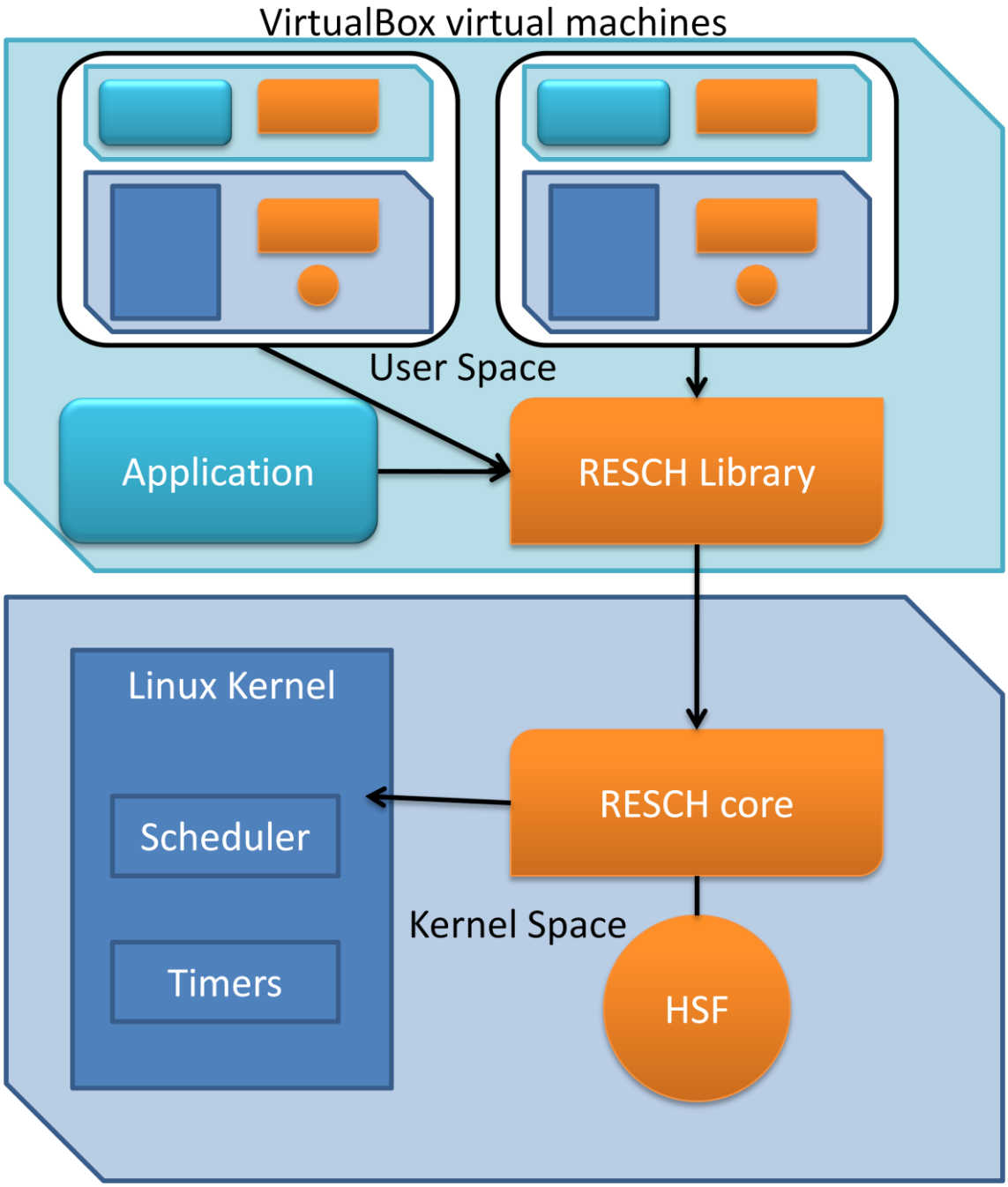


Figure 8: An overview of our suggested real-time hypervisor concept. The hierarchy is evident in the way that RESCH is used identically at the base as well as in the partitions.

To summarize, we suggest a static, hierarchical and fixed priority preemptive scheduling design that uses VirtualBox together with the RESCH suite, allowing soft real-time performance to be achieved without any kind of modification to the Linux operating system.

3.2.2 Motivation

We think that our suggested design would be a preferred alternative to other solutions when it comes to achieving soft real-time performance in a virtualized environment. This is because it has almost no drawback, and all the problems we identified previously are addressed:

- The issue of **existing technology** is solved by the fact that VirtualBox and RESCH are both available for use and they should be considered finished products, especially VirtualBox which has been developed by a company for years. This is also true for Ubuntu which has strong professional company support.
- While we have only tested RESCH properly on an Ubuntu system, the **compatibility** problem is addressed by the capabilities of VirtualBox, as it can run on most operating systems today, be it Linux, Windows or even Macintosh OS, which means that any kind of application can run as well.
- HSF is unaware of what is running within the partitions so **combination** is achieved due to the freedom of running whatever application you wish on them, such as soft real-time media players or ordinary networking experiments. These applications can be either real-time type tasks or merely ordinary programs, since it is up to the user and RESCH how to schedule these but other than that, there is no problem letting them run side by side.
- Linux and VirtualBox are long running products that have been around for a while, making them **stable and reliable**. Because of their large user base they have been tested and tried properly and are presently very stable releases.
- Also because of this fact they have been fine-tuned over the years, with ever increasing simplicity in their usage and great **usability** as a result. As open source software, both Linux and VirtualBox used to require advanced experience, but this has indeed changed.
- The really great thing about this setup however, is that using RESCH allows it to be used without having to change any existing software, making it an **unmodified system**; RESCH merely communicates with the operating system's kernel, it does not change it or integrate any new software into it. This does not only guarantee easy usage, it can also be certain that there are no issues due to compatibility or malfunctioning modified software. Not even RESCH itself is modified; it will only run a different plug-in module which is exactly what it was designed to do.
- Along with the plug-in framework HSF we suggested, RESCH allows us to achieve the **hierarchical scheduling** that we have seen is vital for any partitioned system to work properly. It is not hard to achieve this; as has already been stated, we only need to add a new plug-in module.

3.3 Summary

In this work we have examined many existing solutions for using virtualization in real-time systems. We have performed research and studied these solutions as well as analyzed their suitability for this task, and we find that most of them fall short in one way or another. Most prominently is the issue of software like operating systems that needs to be modified in order for most solutions to work, as this was causing us much trouble during the course of this work. Coming to the conclusion that this kind of design is not feasible in the long run, we suggest a solution of our own that is supposed to address the discovered shortcomings. Based on a combination of Ubuntu, VirtualBox and the scheduling suite RESCH, we think that this solution is optimal for soft real-time scheduling in a virtualized environment.

4. References

1. Aguiar A, Hessel F, (2010), Embedded Systems' Virtualization: The Next Challenge?, Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on
2. Åsberg M, Kraft J, Nolte T, Kato S, (2010), A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux
3. Åsberg M, Nolte T, Kato S, (2010), Towards hierarchical scheduling in Linux/multi-core platform, Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on
4. Bruns F, Traboulsi S, Szczesny D, Gonzalez E, Xu Y, Bilgic A, (2010), An Evaluation of Microkernel-Based Virtualization for Embedded Real-Time Systems, Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on
5. Chandra A, Shenoy P, (2008), Hierarchical Scheduling for Symmetric Multiprocessors, Parallel and Distributed Systems, IEEE Transactions on
6. Crespo A, Ripoll I, Masmano M, (2010), Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach, Dependable Computing Conference (EDCC), 2010 European
7. Heiser G, (2008), The Role of Virtualization in Embedded Systems, IIES '08 Proceedings of the 1st workshop on Isolation and integration in embedded systems
8. Kaser R, (2009), Complex Embedded Systems – A Case for Virtualization, Intelligent solutions in Embedded Systems, 2009 Seventh Workshop on
9. Kato S, Rajkumar R, Ishikawa Y, (2010), A Loadable RealTime Scheduler for Multicore Platforms
10. Lee M, Krishnakumar A S, Krishnan P, Singh N, Yajnik S, (2010), Supporting Soft Real-Time Tasks in the Xen Hypervisor, VEE '10 Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments
11. Li Y, Xu X, Wan J, Li W, Yuan Y, (2010), A Real-Time Scheduling Mechanism of Resource for Multiple Virtual Machine System, ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual
12. Lin B, Dinda P A, (2005), VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling, Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference
13. Pulido J A, Urueña S, Zamorano J, Vardanega T, de la Puente J, (2005), Hierarchical Scheduling with Ada, presentation
14. Tanenbaum, A S, (2009), Operating Systems 3rd Edition, Pearson Education Inc. New Jersey
15. Wang Y, Zhang J, Shang L, Long X, Jin H, (2010), Research of real-time task in Xen virtualization environment, Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on
16. Yu P, Xia M, Lin Q, Zhu M, Gao S, Qi Z, Chen K, Guan H, (2010), Real-Time Enhancement for Xen Hypervisor, Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on
17. Zhang J, Chen K, Zuo B, Ma R, Dong Y, Guan H, (2010), Performance analysis towards a KVM-Based embedded real-time virtualization architecture, Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on
18. http://searchservervirtualization.techtarget.com/definition/virtualization
19. http://www.computerworld.com/s/article/338993/Emulation_or_Virtualization_
20. http://www.wisegeek.com/what-is-hardware-virtualization.htm
21. http://www.wisegeek.com/what-is-full-virtualization.htm

22. http://www.hardwaresecrets.com/article/263
23. https://help.ubuntu.com/community/Xen
24. http://www.wisegeek.com/what-is-paravirtualization.htm
25. http://dictionary.sensagent.com/partial+virtualization/en-en/
26. http://www.xen.org/
27. http://virtualizationreview.com/blogs/mental-ward/2009/02/kvm-baremetal-hypervisor.aspx
28. http://www.virtualbox.org/
29. http://www.oracle.com/us/technologies/virtualization/oraclevm/index.html?origref=http://www.oracle.com/us/technologies/virtualization/oraclevm/024974.htm
30. http://www.vmware.com/
31. http://www.vmware.com/solutions/business-critical-apps/mitel/index.html
32. http://wiki.qemu.org/Main_Page
33. http://www.microsoft.com/windows/virtual-pc/default.aspx
34. http://blogs.technet.com/b/chenley/archive/2011/02/14/windows-virtual-pc.aspx
35. http://sites.google.com/site/realtimexen/
36. http://www.lynuxworks.com/virtualization/hypervisor.php
37. http://www.ok-labs.com/products/okl4-microvisor
38. https://help.ubuntu.com/community/Xen
39. http://wiki.xensource.com/xenwiki/LiveCD
40. http://fclose.com/b/linux/2252/setting-up-stable-xen-dom0-with-fedora-xen-4-0-0-with-xenified-linux-kernel-2-6-32-13-in-fedora-12/
41. http://wiki.xensource.com/xenwiki/Fedora13Xen4Tutorial
42. http://www.real-time-systems.com/
43. http://www.xtratum.org/index.html
44. No author listed, believed to be Xi S et al., RT-Xen: Towards Real-Time Hierarchical Scheduling in Xen
45. https://rt.wiki.kernel.org/index.php/Main_Page

5. Appendix

This section contains various related information.

5.1 Computer specifications

The computer used had these technical specifications:

- Intel® Core™2 Quad Processor Q6600, 2.4 GHz, 4 cores, VT technology
- 2GB RAM
- ATI Radeon X600 graphics card
- 250GB HDD

5.2 How to Install Xen – A Compiled Guide

This is a description of how to install Xen 4.0.1 on an Ubuntu 10.10 system, based on the author's experiences. Other combinations of versions may work as well, but this is by no means guaranteed. The guide at https://help.ubuntu.com/community/Xen will be followed in large steps, but with workarounds where problems were encountered, along with proper explanation where needed. Every step is executed using the command tool unless otherwise specified. Please not that this guide is written for someone with the same skills as the author initially had.

1. The site mentions that some tools need to be installed, so do this:
   ```
   sudo apt-get install bcc bin86 gawk bridge-utils iproute
   libcurl3 libcurl4-openssl-dev bzip2 module-init-tools transfig
   tgif texinfo texlive-latex-base texlive-latex-recommended
   texlive-fonts-extra texlive-fonts-recommended pciutils-dev
   mercurial build-essential make gcc libc6-dev zlib1g-dev python
   python-dev python-twisted libncurses5-dev patch libvncserver-
   dev libsdl1.2-dev libjpeg62-dev iasl libbz2-dev e2fslibs-dev
   git-core uuid-dev ocaml libx11-dev bison flex
   ```
   And
   ```
   sudo apt-get install gcc-multilib xz-utils
   ```
   for 64-bit systems.
2. Move to a directory where you want to place the Xen installation files. A directory with access rights such as *Documents* is preferred instead of */usr/src* as suggested at the site. There, use this
   ```
   wget http://bits.xensource.com/oss-xen/release/4.0.1/xen-
   4.0.1.tar.gz
   ```
   to get the Xen installation files.
3. Unpack the tar ball in your directory:
   ```
   sudo tar –xzvf xen-4.0.1.tar.gz
   ```
4. There should now be a directory *xen-4.0.1* visible. Change to this directory and run these commands to build and install Xen:
   ```
   sudo make xen
   sudo make tools
   sudo make stubdom
   sudo make install-xen
   sudo make install-tools PYTHON_PREFIX_ARG=
   sudo make install-stubdom
   ```
   I must confess that I don't know why the `PYTHON_PREFIX_ARG=` line is added, but the

author of that guide obviously knows what he is doing, and I can't argue with this decision. Verify that Xen is installed by checking the */boot* directory; if everything went alright the files *xen-4.0.gz, xen-4.gz* and *xen.gz* should be there.

5. Xen is now installed. You can however not use it, since you probably don't have a compatible kernel. To do this we need to *check out* a stable *kernel branch*.
6. You must now download a stable kernel by using a *git clone*. Move to a directory with no access rights restrictions such as *Documents* once again. Here, you should issue this command:
   ```
   git clone
   git://git.kernel.org/pub/scm/linux/kernel/git/jeremy/xen.git
   linux-2.6-xen
   ```
   The last part of the command is a directory where the information should be downloaded. If it does not exist it will be created. A lot of data will be downloaded, and this is where problems may be encountered: in my experience the *git clone* may fail if a directory with strong access protection is used. It will then be very slow and abort after the first step.
7. Move into the directory where the information was placed and run the following
   ```
   git reset –hard
   git checkout –b xen/stable-2.6.32.x origin/xen/xen/stable-
   2.6.32.x
   ```
   This is in order to change to a stable kernel that supports Xen. Note that your current kernel most likely cannot be used because of its version, so this is why we cannot make changes to that kernel and recompile only those.
8. It is almost time to start configuring the kernel. First we need to make sure to set a number of properties. Run
   ```
   make menuconfig
   ```
   And move to
   Processor type and features -> Processor Family
   and choose your architecture. If you are using a 32-bit system you need to enable *PAE,* also under
   Processor type and features -> High Memory Support (64GB) -> PAE
   Also make sure that ACPI support is enabled under
   Power Management and ACPI Options
   Exit and save the configuration.
9. Unless you found it (I never did) we need to add support for Xen manually. Open the hidden file *.config* that was created in the directory and add CONFIG_XEN_DOM0=y if you can't find it.
10. Run
    ```
    make menuconfig
    ```
    again and set
    Processor type and features -> Paravirtualized guest support (PARAVIRT_GUEST [=y])
    Processor type and features -> Xen guest support (Xen [=y])
11. The following properties can be set in the interactive kernel configuration, but they may be hard to find. If you don't find them you can simply add them annually in the *.config* file.
    ```
    CONFIG_ACPI_PROCFS=y
    CONFIG_XEN=y
    CONFIG_XEN_MAX_DOMAIN_MEMORY=32
    CONFIG_XEN_SAVE_RESTORE=y
    ```

```
CONFIG_XEN_DOM0=y
CONFIG_XEN_PRIVILEGED_GUEST=y
CONFIG_XEN_PCI=y
CONFIG_PCI_XEN=y
CONFIG_XEN_BLKDEV_FRONTEND=m
CONFIG_NETXEN_NIC=m
CONFIG_XEN_NETDEV_FRONTEND=m
CONFIG_XEN_KBDDEV_FRONTEND=m
CONFIG_HVC_XEN=y
CONFIG_XEN_FBDEV_FRONTEND=m
CONFIG_XEN_BALLOON=y
CONFIG_XEN_SCRUB_PAGES=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_NETDEV_BACKEND=y
CONFIG_XENFS=y
CONFIG_XEN_COMPAT_XENFS=y
CONFIG_XEN_XENBUS_FRONTEND=m
CONFIG_XEN_PCIDEV_FRONTEND=y
```

These are properties taken from the Xen wiki and are guaranteed to be needed for a Xen kernel. In addition, just enable almost everything with “Xen” in the name.

12. Now that the kernel has been configured, it is time to build it. Run
    ```
    sudo make
    ```
    and go away for a break, because this might take a while.
13. If the kernel was built successfully you should have a file *vmlinuz-2.6.32.xx* in the */boot* directory where xx is your version, mine was 39.
14. Update the boot options:
    ```
    sudo update-initramfs –c –k 2.6.32.xx
    ```
15. Now it is time to update your grub entries. Depending on your grub version, it will add this new kernel automatically, and sometimes even a Xen option will be added. Try out your new kernel without Xen to see if it works. If the eventual Xen entry exists as well, and it works, you can skip a few steps and move on to creating a guest. Otherwise you need to edit your grub file manually.
16. Before you do this however, you need to tell xen and xendomains to run:
    ```
    sudo update-rc.d xend defaults 20 21
    sudo update-rc.d xendomains defaults 21 20
    ```
17. The *grub.cfg* (name may vary depending on versions) is located in */boot/grub* and contains entries for your bootable systems and kernels. The site suggests adding this
    menuentry "Xen 4.0.1 / Ubuntu 10.04 kernel 2.6.32.32" {
    insmod ext2
    set root='(hd0,9)'
    multiboot [(hd0,9)]/boot/xen.gz dummy=dummy
    module [(hd0,9)]/boot/vmlinuz-2.6.32.19-xen dummy=dummy root=<sd* | GUID> ro
    [console=tty0 quiet splash]
    module [(hd1,5)]/boot/initrd.img-2.6.32.32
    }
    and it should work overall except for a few changes:
    Make sure that the *set root* parameter points to your partition, i.e. switch the “(hd0,9)” part for what you have. It is often as simple as just copying the path from other entries.
    The *root* parameter may also require special attention. Again, you can just copy other

entries. Some systems uses *root UUID* instead of paths.
Here you may also want to set the menu to show on boot, as well as increasing the timer to allow you some time to choose.

18. That should be it for Xen! Reboot, choose your new Xen entry in the grub boot menu, and if you can boot into Ubuntu as usual, run this
    ```
    sudo xm info
    sudo xm list
    ```
    to verify that Xen is running. If it is, you should see the running domains and a long list of information about the system.
19. Installing a guest system can be difficult or easy. Let us choose the easy way, which is using *Xen-tools*. This is not included in Ubuntu after the Lucid Lynx version, so if you have anything later you have to get it from here https://launchpad.net/~xtaran/+archive/xen-tools. Instructions on how to install are available on the site ("Read about installing").
20. Now, to create a guest image that can be used, I ran this complicated command:
    ```
    sudo xen-create-image --hostname=xen1.example.com --size=4Gb --swap=512Mb --ide --ip=192.168.0.101 --netmask=255.255.255.0 --gateway=192.168.0.1 --force --dir=/home/xen --memory=256Mb --arch=i386 --kernel=/boot/vmlinuz-2.6.32.39 --initrd=/boot/initrd.img-2.6.32.39 --install-method=debootstrap --dist=maverick --mirror=http://archive.ubuntu.com/ubuntu/ --passwd
    ```
    You should not copy it directly, but examine each parameter and adjust it according to your needs. Issuing this command will take some time, but if all parameters are correct, it should finish within a few minutes.
21. Before you start the guest, you should have a look at its configuration file:
    ```
    /etc/xen/xen1.example.cfg
    ```
    Take not of the *kernel* parameter, as well as the *disk* specification. I ran into an error where the *swap* and *disk* files were not present, so I had to change these manually.
22. Almost there now. Run
    ```
    sudo xm create /etc/xen/xen1.example.com.cfg
    ```
    to start your guest virtual machine. Use
    ```
    sudo xm list
    ```
    to see if your new guest is running. If it is, use
    ```
    sudo xm console xen1.example.com
    ```
    to log in to the system via the console. If a proper console is not showing, it might be the problem I had, that a proper *disk* file has not been specified in the configuration file as stated above.
23. If you want network access from your virtual machines, then I am sorry, I don't know how to fix this. I believe it has got something to do with the configuration file of the guest however, or possibly the xen-tools configuration file.
24. To stop a virtual machine use:
    ```
    sudo xm shutdown xen1.example.com
    ```
    or
    ```
    sudo xm destroy xen1.example.com
    ```
    The last command is equal to pulling out the power cord of the system.