Synchronization of Operating Systems in Heterogeneous Testing Environments

RWTH Aachen University
LuFG Informatik 4 Distributed Systems

Diploma Thesis

Florian Schmidt

Advisors:

Dipl-Inf.	Elias Weingärtner
Prof. Dr.-Ing.	Klaus Wehrle

Registration date:	12 December	2007
Submission Date:	12 June	2008

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 12. Juni 2008

Kurzfassung

Die Entwicklung und Analyse von Software durchläuft im Normalfall mehrere Phasen. Am Anfang steht ein Konzept, das im Fall von Netzwerkprotokolldesign in einem Netzwerksimulator getestet werden kann. Schließlich erfolgt die Entwicklung eines Prototyps zur Analyse einer konkreten Implementierung. Simulation und Prototypen haben beide ihre spezifischen Vor- und Nachteile. Netzwerkemulation erlaubt die Verknüpfung beider Ansätze, allerdings muss für eine realistische Analyse im Hinblick auf das Zeitverhalten die Simulation in Echtzeit ablaufen können. Dies ist in komplexeren Simulationsszenarien nicht gewährleistet. Diese Diplomarbeit entwirft und setzt eine Testumgebung um, die durch Kapselung eines x86-Systems in Xen eine Synchronisation der Implementierung mit beliebig komplexen und langsamen Simulationen erlaubt und gleichzeitig eine Analyse des Zeitverhaltens ermöglicht.

Abstract

Software design generally employs different techniques in different phases for analysis. In the beginning, a new network protocol may be conceptualized and tested in a network simulator. Later on, a prototype of the actual implementation is thoroughly analyzed. Both approaches have their specific up- and downsides. Network emulation allows to combine both to gather additional data and facilitate easier prototype testbed layouts. For meaningful analysis however, especially in regard to timing behavior, the simulation has to be real-time capable. This is often not the case for complex simulated scenarios. This diploma thesis designs and implements a synchronization that allows to run any x86 operating system with arbitrarily complex simulations. By encapsulating the OS in the Xen hypervisor, it relieves the simulation from the real-time capability constraint while still maintaining realistic timing behavior.

Acknowledgments

A work such as this is never the result of one person working all alone. I want to use this opportunity to thank all the people who made it possible to write this thesis, and helped in improving it:

Prof. Dr. Wehrle, for giving me the opportunity to write this thesis at the distributed systems group, and for being my supervisor.

Prof. Dr. Kowaleswki, for kindly agreeing to be my second supervisor for this thesis.

My advisor, Elias Weingärtner, for sparking in me the interest in the topics of synchronization and virtualization, for always taking time for discussions of problems and solutions, and for being a great motivator.

Ismet Aktas, for always being supportive, and for having strong belief in my skills, sometimes more than myself—and he was right.

All members of the distributed systems group, especially Stefan Götz and Olaf Landsiedel, for many interesting and fruitful discussions, and for contributing to the excellent work atmosphere.

My brother Gereon, Corinna Habets, and David Piegdon, for helping me proofread this text.

Finally, my parents, for always supporting me during the not always easy time of my studies.

Contents

1 Introduction

Whenever a new piece of software is developed, it has to be tested for proper functionality. Furthermore, if it contains a new concept, such as (in the field of network communications, which this thesis concentrates on) a new networking protocol, it has to be analyzed to make sure there are no conceptual errors. Two important ways of analysis are simulation and prototyping. In simulation, the concept in its abstract form is implemented to use it in a simulation environment, and then tests are run. The main advantage is that it is generally comparably easy to do the step from concept to a simulation model in contrast to a full prototype that runs on an end-user system, and it allows for a great deal of flexibility. Once the functionality of the concept has been implemented in the simulation, networks of arbitrary size and complexity can be constructed for stress-testing, and tests can be rerun to closely analyze special cases.

However, simulations abstract from many factors that the final product has to deal with, such as side-effects from hardware or operating system. At some point, it is therefore necessary to create a prototype and analyze it in a natural testbed environment. Testbeds are limited in size to a couple hundred of prototypes however, and the latter already introduces a massive cost and effort. Hardware for all the nodes inside the testbed has to be procured, and every time some part of the prototype implementation is changed, all the prototypes have to be updated. Practicality therefore dictates the maximum number of prototype nodes in the testbed. This is unfortunate because it would be very desirable to analyze the performance of the prototype in more complex network topologies.

Consequently, it is desirable to find a way to combine prototyping with simulation. One way to reach this goal is network emulation, in which a prototype is connected to the simulation by means of an emulator, that can translate between real network packets from the prototype, and packet messages from the simulator, so that communication between the two is possible. The problem is that this only works as long as the simulation is able to run at real-time speed. When it does not, results measured are not correct in respect to timing information any more, which can lead to incorrect behavior. For example, a simulation might take so long to route a packet to the receiver, and afterwards the ACK packet back to the prototype, that the prototype assumes it has been lost and retransmits it. In the worst case, this can lead to an ever-increasing load on the simulation, to the point where it becomes *live-locked* (events are created faster than they can be processed, and the simulation stalls). Unfortunately, simulations tend to run slower and lose their real-time capabilities the more complex the simulation is. But this is exactly the most interesting use case for network emulation, since small networks can be feasibly analyzed with a testbed of prototypes only.

The solution would be to synchronize simulation and prototype to each other, so that the faster can wait for the slower to catch up again. In this case, this means to stop the prototype. This, however, is easier said than done for end-user computer systems, such as the x86, since they keep track of time. Even if the operating system is modified to sleep for certain periods until the simulation has caught up again, a number of hardware clocks and timers inside the system will make it immediately noticeable that time has passed. To come back to the prior example, a retransmission timer would still expire at the same time as before, regardless whether the system has been put to sleep or not, which means nothing has been won.

Obviously, it is a very tantalizing thought to solve the synchronization problem in network emulation, because it would allow to analyze how prototype implementations behave in large network, without the immense cost of creating a testbed of the same size. This thesis proposes a solution for x86 based systems. It uses the Xen Hypervisor [9] to disconnect an operating system from directly accessing the hardware, and modifies it so that two goals are reached:

1. The Operating System must be stoppable and startable at any point in time, and run for precisely the amount of time assigned.

2. The Operating System must not notice that, during the time it did not run, any time passed. In other words, although it will run only intermittently, it must seem to it as if time passed continuously without any gaps.

Under these two constraints, a system can be stopped whenever the simulation falls behind, and restarted after the latter has caught up. Throughout this work, they will be referenced as “requirement 1” and “requirement 2”. The prototype can therefore be synchronized to the simulation. The presented Xen implementation can mask the passing of time from an OS during descheduled times and accurately control the execution down to time slices of 10μs, which is therefore the maximum amount of time the clocks inside simulation and prototype can ever differ.

To complete synchronization set-up, the author developed or reused a few other components to make the synchronization work as a whole:

1. A central synchronization server* that makes sure no prototype or simulation deviates from the common time by more than a predefined amount.

*Developed by Elias Weingärtner

2. A synchronization client[†] that runs on the same machine as the hypervisor, and interfaces with Xen to control the execution of the synchronized operating system.
3. An OMNeT++ scheduler that interfaces with the synchronizer.[*]
4. An emulator that translates between network packets and simulator messages, for the OMNeT++ simulator.[‡]

Note that on the one hand, while the work described in this thesis uses OMNeT++ [69] as simulator, the changes are generic enough to be easily applicable to most other discrete event network simulators. On the other hand, while the concept of using virtualization to encapsulate a prototype can be applied to other virtualizers, or even full-system simulators such as Simics [50], or processor emulators such as QEMU [12], the work done for this thesis is too specific as to allow for fast porting to those.

The rest of this diploma thesis is structured as follows: Chapter 2 will give an overview over and background information on different components that are important to the understanding of the work done. The actual implementation, with an explanation of how Xen has been changed to reach the two goals for synchronization outlined above, how the synchronizer works, etc. is presented in Chapter 3. Chapter 4 will analyze results of tests that have been conducted. Chapter 5 will discuss previous work done in related fields, while potential future fields of work will be proposed in Chapter 6. Finally, Chapter 7 will give a summary.

[†]Own work
[*]Developed by Elias Weingärtner
[‡]Own work, parts of which have been adapted from an earlier work by Joachim Riedl [59].

2

Background

In any field of work that involves creating large numbers of a developed invention, testing and analysis are important steps from the initial idea to the final product. No car in the world is built right away from the initial concept art. Especially in engineering, the testing can take several years and will pass through different phases.

While the testing standards might not be quite as high in the field of computer science, testing and analysis still play a vital role in the development of new communication systems, such as network applications or communication protocols. Testing will also typically pass through different phases. Early on, and for a long time during the development, *simulation* is a prime choice; later on, *prototype implementations* will give additional insight into potential problems.

2.1 Network Simulation

The most widely used type of network simulation is discrete event simulation (DES). In this type of simulation, every action by any node that changes the simulation's overall state is represented as an event. Every event, in turn, is assigned a certain point in time at which it will happen. The fact that an event is assigned a time in the future at its creation time means that the simulation can maintain an ordered event list at all times. This allows the simulation to advance directly to the next event in the queue after it has finished processing the current one. Time is therefore not modeled as a constant flow, but rather in discrete steps.

In network simulation, DES is generally used in the form of packet-level simulation. OMNeT++ [69], the simulator used in this thesis, as well as the also widely-used ns-2 [28], are both packet-level DES systems. This allows for a very detailed simulation, since communication between nodes closely resembles the communication behavior in the real world.[1]

The advantage of simulation is the high degree of flexibility. It is possible to easily and quickly set up networks consisting of hundreds, thousands, or hundreds of thousands of nodes. For example, if the goal is to evaluate a new network protocol, its behavior is modeled in the simulator—for example, as a C++ class in the case of OMNeT++—and then can be defined to be part of the network stack of each of the nodes in the network. Many simulation systems give the user a high degree of interactivity, such as stopping the simulation or stepping through it event by event, and provide some sort of visualization that will allow to monitor certain parts of the simulated network. Also, some simulators allow to take snapshots of the simulation state. This, coupled with the fact that DES is deterministic (except for randomness deliberately inserted into the system by the user), facilitates repeatable analysis of critical situations. However, the results apply only to the concept of the network protocol, as it has been modeled inside the simulator, and can therefore expose faults only in the concept, such as placing too much load on some nodes in the network, or low performance for others. Simulation generally abstracts from the internals of nodes: There is no operating system running on them with a complete network stack and tasks to schedule, and whatever happens inside the node is generally modeled to run without any time consumption—and if there are models to account for it, they are necessarily simplified. While this elimination of side-effects facilitates repeatability and determinism, these very side effects can have a noticeable influence on the performance in the real world. In other words, the simulation models only a concept, not any actual implementation.

2.1.1 The OMNeT++ Discrete Event Simulator

The OMNeT++ simulator is highly modular. It allows to define behavior and interfaces in the form of C++ classes, termed “simple modules”. Interface are defined in a two-fold way. From the simulator core’s point of view, each module is derived from a base class that already defines certain methods that are called by OMNeT++ whenever an event of a specific type occurs. For example, the method `handleMessage()` is called whenever a message is received by the module and has to be processed. From the module’s point of view, the interface consists of a number of defined input and output gates, through which messages are sent and received. These gates are connected to gates of other modules via connections that can also be assigned specific properties, such as propagation delay.

Furthermore, it is possible to combine several simple modules into a compound module. Recursively, compound modules can be combined with other compound modules into new compound modules without any limit to the created hierarchy. These compound modules are defined by a simple description language. Such a module consists of gates and of its contained modules. A contained module’s gate can either be connected to the corresponding gate of another contained module, or to a gate of the compound module. An example of this structure is depicted in

[1]There are other ways, such as fluid simulation [48], which can reduce the computational complexity compared to packet-level simulation, and is therefore suited for very large and complex simulations, if details down to every single package are not needed.

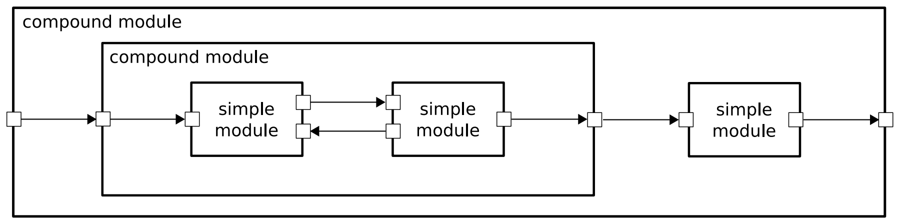


Figure 2.1 A schematic structure of OMNeT++ modules.

Figure 2.1. Arrows represent connections between gates (small boxes). A compound module is created from the combination of two simple modules, and in turn is used, together with another simple module, in the definition of another compound module. Compound modules form some sort of black box: From the outside, they cannot be discriminated from simple modules, and their behavior is defined by the co-action of their contained modules, without them being visible from the outside. In the end, the simulated network that is created by the user is nothing more than a complex compound module that contains every node that was defined.

The extendability of OMNeT++ is demonstrated by the number of published simulation models. While most of the models extend the functionality for network simulation, such as the INET framework [38], which will be described in the next section, OverSim [11], which was designed to simulate overlay networks, as used in many peer-to-peer applications, or the Mobility Framework [25], which supports the simulation of wireless networks, where nodes oftentimes are not stationary. The simulator is not limited to network simulation (although this is the field where it is most widely used); for example, there also exists a model that allows the simulation of a SCSI bus and devices.[2] Moreover, modules are not limited to defining new behavior inside the simulation model. For example, it is also possible to write a new scheduling model to influence how OMNeT++ schedules its event queues and processes events. This is exactly what has been done to synchronize OMNeT++ against other components and is described in Section 3.2.2.

This combination of simple modules that can be defined in an exact way via a standard programming language, and compound modules that allow the combination and reusage of modules, makes OMNeT++ both powerful and flexible. The possibility to define module parameters that can modify some parts of the module's behavior eases the creation of large networks of similar, but not necessarily identical nodes. However, the general limitations of simulations obviously still apply.

2.1.2 The INET Framework

The INET framework [38] is a collection of modules for the OMNeT++ simulator. It contains modules that model packets of different types, such as TCP or ICMP, and the encapsulation and decapsulation of one packet type into another, modules that simulate the behavior of a network layer, queues, interfaces, etc., and a few

[2]Or rather, existed. The module is not maintained any more. Nevertheless, it serves as an example to the wider range of applications that can be served by the OMNeT++ simulator, it was mentioned by Varga when he presented his work on OMNeT++ [69], and the sources are still available.

definitions of full-fledged hosts with a complete networking stack, as well as hubs, switches and routers, compounded from the other modules.

Simulation events are to a large extent associated with payloads and their packaging, sending, unpackaging and processing. Every packet that would be sent by a real hardware system is simulated as an independent (chain of) event(s). Typically, an event such as "package this payload as UDP" would also send out the package via an output gate to the network layer module, where it would be processed and fit with an IP header, from which it would be sent to the data link layer, and so on. Self-addressed messages that wake up a component at a certain point in the future can be used to simulate package retransmission timers.

All in all, the INET framework allows detailed simulation and monitoring of the innards of network stacks as they are typical today in virtually all networked hosts that employ the TCP/IP model of communication. For this work, the INET framework was used together with OMNeT++ to create hosts inside the simulation that could communicate with real systems that ran directly on hardware.

2.2 Prototyping

For performance evaluation under realistic conditions, simulations are not viable. To acquire such data, the concept is generally implemented as a prototype and run in a testbed. This allows for the highest accuracy in analysis, since the prototype is an exact replication of the final product. Prototype testing will involve ensuring that no side effects introduced by the hardware, operating system, and other elements that were not simulated, will have a decidedly negative influence on the performance. The trade-off is that the testing is more complicated. Not only does introspection become much more tedious than in simulation: There is no kernel programmer that has not used the printk/recompile cycle. Also, some side effects are hard to trace down: The reason that the sensor node's operation breaks down may be due to a programming error—or maybe the power supply is not powerful enough and leads to brownouts.

There is another fundamental problem with prototype testing. It works reasonably well for small-scale investigations, i.e. if the network comprises at most a few dozen nodes. If the numbers are greater than this, testbeds become problematic: The cost increases with each node that has to be bought in hardware, and the implementation has to be distributed among all of them; this has to be repeated whenever the implementation changes, for example when a bug has been found and fixed. Furthermore, if the analysis involves performance investigation on nodes that are scattered around the world, maintaining the testbed becomes a logistical nightmare. And while open platforms such as PlanetLab [19] exist for testbed investigation, they do not grant exclusive access to their resources. This is necessary, however, if the prototype involves changes in the operating system itself, and is not confined to a user-space application. Moreover, without exclusive rights, it can not be ensured that other concurrently running programs will skew the results of the performance evaluation. It therefore remains a necessity to construct your own testbed environment, which is generally impossible for very large scale tests.

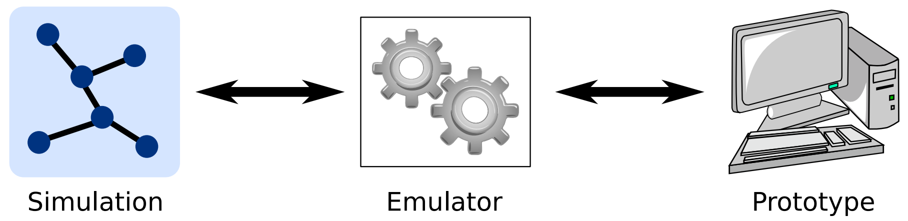


Figure 2.2 Network Emulation

2.3 Emulation

This leaves the evaluator with two approaches, each with its own advantages and disadvantages, and disjoint fields they are feasible in. It also means that for the important area of meaningful performance evaluation of prototypes in large scale networks, neither of the two ways works in a satisfactory way. *Emulation* combines the specific strengths of both approaches. A special component (the *emulator*) is charged with the task to translate between the two worlds of simulation and prototype implementation; in other words, to create a *functional coupling* between the two entities.

One way to combine the two, called *environment emulation*, is to insert a framework into the simulation that allows to run the implementation designed for the prototype from within the simulator. Generally, work in this field has focused on integrating an operating system's network stack into the simulator for use by the nodes, typically the FreeBSD stack [14, 41], which facilitates communication between the nodes via standard internet protocols, instead of the custom-tailored simulator messages. While this approach reduces the amount of abstraction from an actual implementation, it has two drawbacks: Firstly, a certain emulation is specific to the inserted implementation. In the aforementioned case, it will produce meaningful results only if the prototype runs on FreeBSD. Secondly, computational complexity, side effects arising from a full-fledged operating system, and timing (which is influenced by the former two) are still not accounted for.

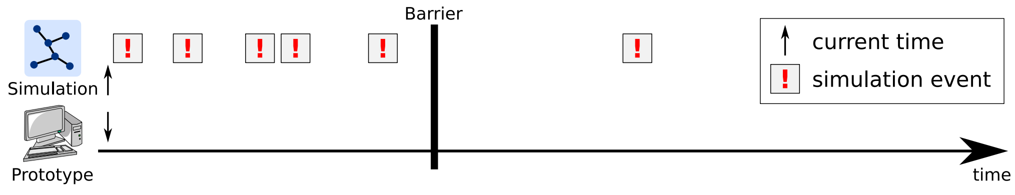
The other approach is called *network emulation*, and this thesis will focus on this field. Figure 2.2 shows a conceptual diagram of its set-up. Rather than to insert the prototype into the simulation, the two components stay separate, connected via a standard network connection, and it is the emulator's task to create a gateway to translate between the two entities and provide the functional coupling this way. Whenever a packet is sent by the prototype to a node inside the simulation, the emulator will convert the real packet into a message that is understood by the simulation and contains the original payload, and vice versa. A typical way to create such an emulator is described in Kevin Fall's groundbreaking work [27]: On the prototype's side, all the packets that are addressed to a simulated node are captured and sent to the simulator's side of the emulator. There, they are translated into messages understood by the simulation, and inserted at a certain point in the network topology that, to the other nodes inside the network, does not look different from any other simulated node.

2.4 Synchronization

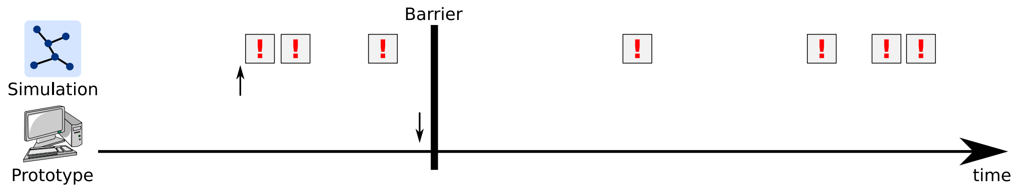
However, there is one big problem with network emulation: prototype and simulation handle the passing of time in totally different ways. While for the former, time passes in a continuous fashion at natural speed, a discrete event simulation jumps from point to point in time to whenever an event is scheduled. This means that prototype and simulation will rarely agree on what time it is. If the goal is to get meaningful information about the performance of a prototype, this is a serious drawback. For example, if a simulation has few events to process during a certain period, it will run much faster than real time. If now the prototype is supposed to act as a hub for simulated nodes, it will receive their packages at a much faster rate than would happen under real circumstances, and conversely, will seem slow in its reaction to the simulated nodes. Fortunately, this is a solvable problem. Since the simulation itself runs as an application on a real computer, and therefore can ask the underlying operating system about the current time, it can slow itself down to never run faster than real time. The time then passes at the same rate in the simulation and the prototype: they are *synchronized* to each other.

The real problem is if the opposite happens. If the simulation has many events to process, it will slow down and not be able to advance in real time. Now, every simulated node will seem slow to the prototype. What is even worse is that this can form a vicious circle: If nodes seem unresponsive to the prototype, it may send out retransmissions of packets, or reason that the node is down and try to connect to another one that offers the same service. When those packets are translated and inserted into the simulation, the amount of events to handle increases, which will slow down the simulation even more. This situation is called *overloading* or *livelocking*. The latter describes the situation in which the simulation cannot progress in time any more at all because there are more events incoming that need immediate processing than can be handled. In contrast to the first problem, this one is much harder to solve. Fall already recognized it and noted, 9 years ago, that "[a]t present, there is no simple solution to this issue" [27]. To the knowledge of the author, nobody has come forth with a simple solution so far, although the essential idea is easy enough. To get prototype and simulation to agree about the current time, there are two solutions: speed up the simulation or slow down the prototype.

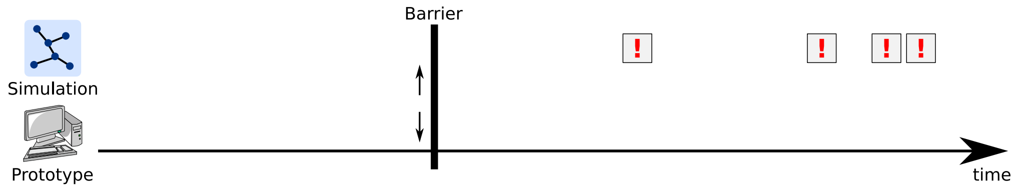
Speeding up the simulation can be realized in two ways. The easiest is to just buy faster hardware for the simulator. Failing that, one can distribute the DES over several computers to allow for parallel processing [51]. This *parallel DES*, however, opens up another class of problems: Now these simulations have to be synchronized against each other. DES relies on the fact that the events in the event queue are processed in order. If two events that are scheduled at different times both change the global status of the simulation, the latter must not be processed before the earlier. Otherwise, causality errors will occur (future events influencing the behavior of past ones). Also, an event can create a follow-up event, which may or may not change the simulation and influence later events. In short, the decision when to parallelize processing of events and when not is not an easy problem. Besides, it is as futile as buying new hardware: In both cases, for every given amount of computing power, there exist simulations which are large and/or complex enough to break the real-time constraint.



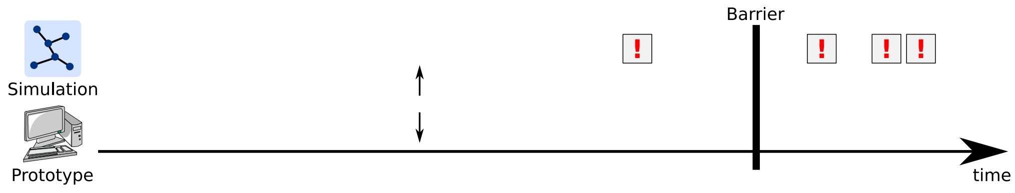
(a) Situation at starting time.



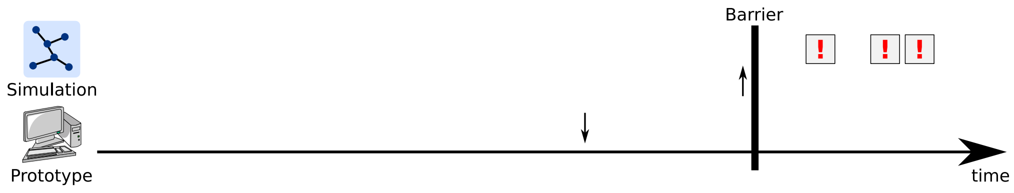
(b) Prototype waits at the barrier.



(c) Simulation catches up.



(d) Barrier is lifted, new barrier is set.



(e) Simulation waits at the barrier.

Figure 2.3 An example of Conservative Time Window synchronization.

Slowing down the prototype may sound easy, but simply reducing the speed at which a system runs will work only for the most simple systems. Every halfway complex one has a real time clock, and every operating system has a way to measure the passing of time and will not be easily tricked. Therefore, every approach to solve the problem will have to find a way to either slow down or halt these hardware clocks at will. For this thesis, the author has discarded this as infeasible because in an x86 computer system there are too many hardware time sources that are nigh impossible to reach, and done the next best thing: disconnect the system from the hardware timers by using virtualization. Section 2.5 will give insight into the technique of virtualization, while Section 2.7 will give an introduction into the timekeeping facilities of a x86 computer.

Finally, with a solution to the problem of how to slow down one part of the network emulation to the speed of the other if needed, and vice versa, a decision has to be

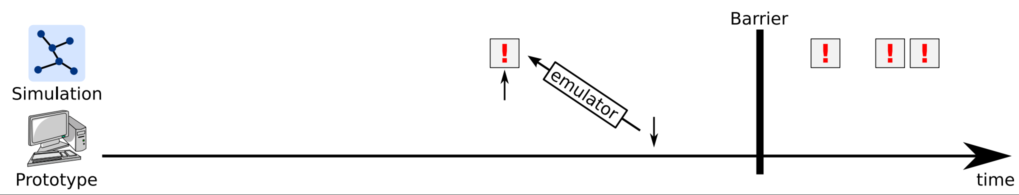


Figure 2.4 A message sent from the faster component to the slower may appear to have traveled backwards in time.

made about how to synchronize the two. Fortunately, the aforementioned parallel DES community has tackled the problem of synchronizing simulations running in parallel for a long time, and come up with several solutions (for an overview, see [29]). Unfortunately, most of them will not work in our special case, where one of the synchronized entities is not a simulation, but a prototype implementation running on a real computer system. The reason is that virtually all algorithms for parallel DES rely on the fact that it is possible to look into the future at distinct events, and decide whether they will influence each other (the conservative approach), or even process events first and roll back later if some events did turn out to influence each other (the optimistic approach). Rollback is not possible on the prototype without taking regular checkpoints of the full system state. This is a large amount of data on an x86 system with a decent amount of RAM. The amount of checkpoints that would have to be created to make rollbacks work properly does not seem feasible. Furthermore, the analysis whether an event influences another will work properly within one system only. In our network emulation setup, where at any given point, a message can be inserted into the simulation by the emulator because a packet was sent by the prototype without the simulator being able to know beforehand, deciding what events are safe to execute is impossible.

One algorithm, however, will work in our case too. The Conservative Time Window (CTW) algorithm allows every synchronized component to run for a certain amount of time, after which it will block until all others have also reached the barrier time that was set. Then the barrier is lifted and set to another point in the future. This means that every component is assigned a *time slice* of equal size; a simulation will then process all events with a scheduled time before the end of that slice, and a prototype will run for the length of the slice before it is stopped again. Figure 2.3 gives an example of how the Conservative Time Window algorithm works. The small arrows denote the current time as it is witnessed by simulation and prototype, respectively. Figure 2.3(a)) shows the state of the synchronized components at the start of the synchronization. In Figure 2.3(b), the simulator was not able to process its events in real time, and while the prototype has already finished its assigned time slice, the simulator has not. The CTW algorithm ensures that the prototype waits for the simulation at the barrier. When the simulator has also reached the barrier (Figure 2.3(c), the barrier is lifted and set to another point in the future (Figure 2.3(d)). Both simulation and prototype start execution again. If now there are few enough events so that the simulator can process them faster than in real time, it is its turn to wait at the barrier (Figure 2.3(e)).

In parallel DES, deciding on the size of the time window will influence the amount of possible parallelization [29]. Typically, the larger the window the smaller the amount of possible parallelization. In our case, however, the amount of parallelization is fixed to simulation and prototype running in parallel. This raises the question what

Ethernet Network Speed	Barrier Size
10 MBit	51.2 μs
100 MBit	5.12 μs
1 GBit	512 ns
10 GBit	51.2 ns

Table 2.1 Time a minimum-size (64 byte) Ethernet package takes to transfer over the cable at different speeds.

the appropriate window size for our case is. To decide this, it is necessary to first determine what potential errors too large time slices can generate. Figure 2.4 shows a *causality error*: If a fast simulation A is synchronized to a real system B, B will reach the barrier before A. If it sends a packet late during the barrier time, there is a high chance that, after it is translated by the emulator and inserted into the simulation, it will appear to have traveled backwards in time. Therefore, the most straightforward solution is to choose the time slices small enough so that they are just sufficient for one packet to be sent over network cable that connects prototype and simulation. This way, packet arrival times will always be aligned with the end of a time window, and therefore no erratic time travel of packets is possible. Since the time a packet spends on the wire is determined by the size of the packet, the assigned time slices have to be as small as the time the shortest packet spends on the wire. For the typical case of an Ethernet connection, the shortest packet is defined by the standard to be 64 bytes long. The time such a packet spends on the wire can be approximated by its size divided by the line speed. Table 2.1 lists times for common Ethernet speeds.

However, in the case of network emulation, even longer time slices cannot introduce causality errors, because of two reasons. First, a reply to a message can never reach the sender before the original message has been sent out. If the sender and receiver both reside inside the simulation, the event queue will ensure that the order is kept. If, on the other hand, the sender is a simulated node, and the receiver the prototype (or vice versa), the original message has to traverse the emulator first before it reaches its destination, and the reply has to traverse it again. Therefore, the order of related messages will not change, and a reply cannot influence the request that it answered. Second, even the order of unrelated messages is never changed. Again, the order of messages sent between two simulated nodes is ensured by the simulator's event queue, and in the case of communication between simulation and prototype, the emulator will translate the messages in the order it received them.

Nevertheless, while no causality errors can occur, the travel time of messages can still be skewed, such as in the example depicted in Figure 2.4. Note, however, that the size of the time slice (the time between two barriers), constitutes an upper bound to the amount of skewing that can occur. Even in the case of a message sent from one side (simulation or prototype) at the very end of its slice to the extremely slow other side, which is still at the very beginning, the skewing can never be more than the time slice. The opposite is also true: if the very slow side sends out a message early during its slice, but due to its slowness, the receiver has already finished and waiting at the barrier, the message will be delivered at the very beginning of the next slice, still holding the upper bound.

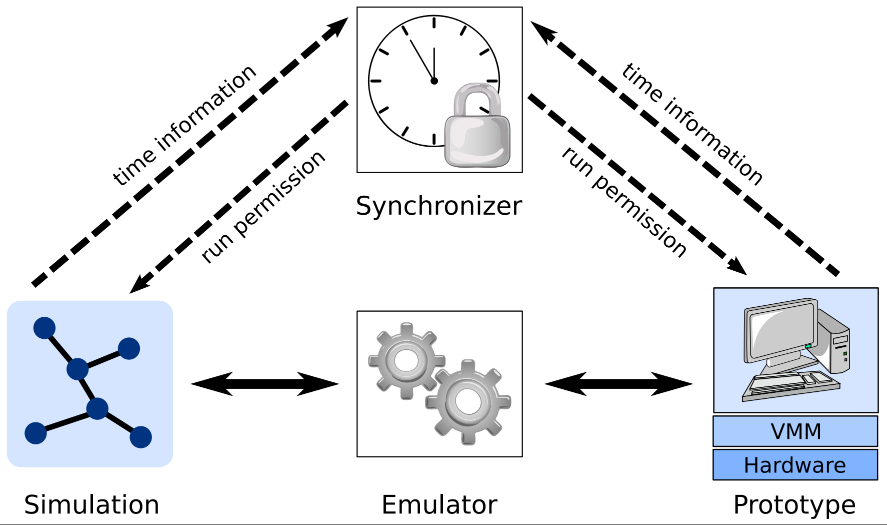


Figure 2.5 Synchronized network emulation

Nevertheless, since the system presented in this thesis has been built to analyze network implementations, and no protocol exists that requires timing accuracy down to micro- or even nanoseconds (nor would it be prudent to design one that is supposed to be usable on the Internet), some skewing can be deemed acceptable, and therefore higher barrier times than the theoretical ones that Table 2.1 suggests are feasible. The reader should always keep in mind that the goal of this thesis is to create a tool to analyze network protocols, not timing down to almost CPU instruction-level. The limitation should still be realized when running analysis: some protocols may report timing information down to the single-digit microsecond range. The standard Linux ICMP ping utility is a good example (examples of this will later be seen in Chapter 4). The CTW synchronization's limitation means that timing is only guaranteed to be correct down to slice size. Every measurement that includes timing data with sub-slice resolution has to be taken with a grain of salt, because the numbers are not guaranteed to be correct.

Finally, note that the CTW algorithm requires an entity in the network, the *synchronizer* or *synchronization server*, that sets the barriers. To do so, it will require information from the synchronized components about what their local time is. In turn, it will send out run permissions to all components to run up to the next barrier time whenever all have reached the current barrier. The concept of *synchronized network emulation* is shown in Figure 2.5. In addition to the concept of network emulation depicted earlier, the synchronizer has been added as an additional component. While the emulator creates and maintains a functional coupling between simulation and prototype and translates communication between the two nodes, the synchronizer creates a synchronous coupling by receiving time information from the two components, and sends out permissions to run for a specified amount of time.

2.5 Virtualization

To facilitate synchronized network emulation, the author has employed virtualization of real systems. The reasons for this have been already been explained in Section 2.4. Virtualization is a concept by which a program, called the virtual machine monitor

(VMM), allows several other programs (or operating systems) to run on a computer at the same time. It generally does so by giving the other programs the illusion of a full computer system (the virtual machine) at their exclusive disposal, while in reality, the access to hardware is shared between them. The idea of virtualization is all but new: It had already surfaced by the late 1950ies to early 1960ies as a means for time-sharing on large mainframes [20, 65], and IBM used it by the mid-1960ies in the form of CP/CMS [1] (after earlier efforts, only involving partial virtualization, such as CTSS and M44/44X [22], had proven insufficient).[3] In 1974, Gerald Popek and Robert Goldberg published groundbreaking work on the field of virtualization [56]. They classified all CPU instructions into three groups:

1. *Privileged instructions*: Instructions that trap, i.e. a switch to kernel mode if it happens outside of kernel mode
2. *Control-sensitive instructions*: Instructions that change configuration of parameters, i.e. processor registers
3. *Behavior-sensitive instructions*: Instructions that behave differently depending on configuration of parameters, i.e. processor registers

For example, a test for zero is a control-sensitive instruction if the result is saved in a specific “zero register”, and a jump if zero is a behavior-sensitive instruction if it looks at this register for its decision to jump or not. Furthermore, they required a VMM to fulfill three properties:

- *Efficiency*: All nonsensitive instructions must be directly executed on the hardware, i.e. without intervention from the VMM.
- *Resource control*: The virtual machine must not be able to affect the resources outside of those assigned to it.
- *Equivalence*: Overall, the behavior of a program inside the virtual machine must be the same as if it was run natively.

Popek and Goldberg proved that it was possible to construct such a VMM if the set of sensitive (both control- and behavior-sensitive) instructions is a subset of the set of privileged instructions. However, for many computers, this condition is still not true. Most important, the x86 CPUs do no meet these requirements and therefore are not virtualizable in this classic sense [61].

One help that x86 CPUs, starting from the 80386 and its new “protected mode”, did bring with them is the concept of *protection rings*. While many architectures have only two modes, privileged and unprivileged, the developers of the 80386 took the concept of 4 of those rings from the VAX architecture, with ring 0 being the most privileged and ring 3 the least. The design idea even went so far as to construct the rings in a way that allowed to run old pre-80386 applications that only made use

[3]The interested reader can find a very accessible overview over the development of IBM virtual machines, with an emphasis on the 1960ies and 1970ies, in an article written by Melinda Varian [70].

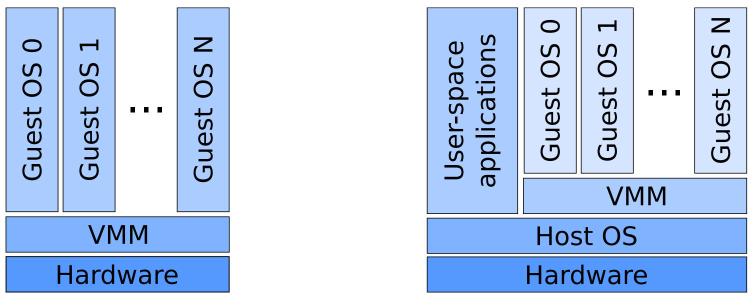


Figure 2.6 The two types of virtual machine monitors. To the left, a type 1 VMM, to the right, a type 2 VMM.

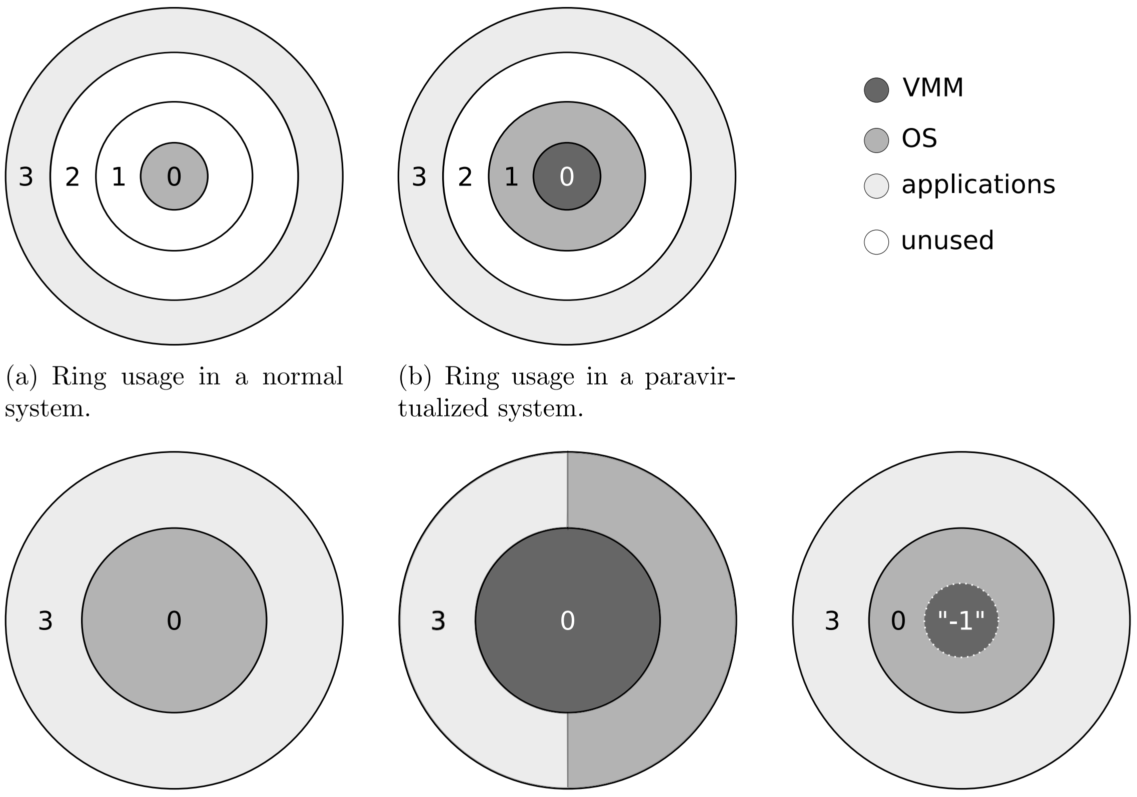
of the previously existing so-called "real mode", in a virtual environment, basically allowing to virtualize an old real-mode DOS operating system on the new 80386 CPU. However, this technique never saw much use. Newer operating systems that operated in protected mode still only made use of two rings, with ring 0 being used for the privileged and ring 3 for the unprivileged mode (see Figure 2.7(a)). Ring 1 and 2 fell into disuse, to a point where they were not even included in the 64 bit specification x86-64 [18].

Although these protection rings lend themselves to the idea of virtualization, even most VMMs available today forego them in favor of other techniques. One of the most widely used ones involves changing the code in a way that keeps the equivalence to the original one (and in fact directly executes most of it unchanged), but replaces the offending instructions. Keep in mind that instructions that are sensitive, but not privileged, break virtualization. This approach makes sure, in a most direct way, that no such instruction is executed, but rather replaced by a functionally equivalent (set of) instruction(s). This can be done while the machine is running and is named "binary translation" [63]. (This on-the-fly translation is not to be confused with an older concept by the same name, that fully converts programs from one computer's binary code to another before their execution, as in [64].) Recently, the situation on the x86 front has changed somewhat, to the point where x86 CPUs can be fully virtualized. This concept of hardware virtualization, sometimes also named "hardware-assisted virtualization", or "hardware virtualization mode" (HVM), is discussed in Section 2.5.3.

VMMs can be classified into two types. A *type 1* VMM runs directly on the hardware; any operating system running on a machine that uses this type of virtualization is by definition a virtualized guest operating system, i.e. it runs on top of the the VMM. A *type 2* VMM runs as an application inside an operating system. Therefore there is a distinction between the natively-running host operating system, which the VMM runs on, and which has direct access to all hardware, and the guest operating system(s), which is (are) virtualized. Figure 2.6 illustrates the difference between the two concepts.

2.5.1 The Xen Hypervisor

The work of this thesis has been done on the Xen hypervisor [9], for reasons discussed in Section 2.5.4. First of all, it should be noted that the term "hypervisor" has



(a) Ring usage in a normal system.

(b) Ring usage in a paravirtualized system.

(c) Ring usage in a normal x86-64 system.

(d) Ring usage in a paravirtualized x86-64 system.

(e) Ring usage in hardware virtualized x86-64 system.

Figure 2.7 Usage of protection rings on the x86 and the x86-64 systems. Note that there is no x86 system that supports hardware virtualization. (after a similar figure in [18])

no unanimously accepted definition. In some cases, it is used interchangeably with "virtual machine monitor", in other cases, it might only apply to VMMs that employ paravirtualization[4], or only to one type of VMMs. Since this work mainly deals with Xen, starting from chapter 3, the term "hypervisor" will be used as a synonym for "Xen", and "virtual machine monitor" or "VMM" to denote the technology as a whole.

Xen comes in the form of a small kernel that is booted instead of a standard operating system, and will act as a layer of indirection between the virtualized operating systems and the hardware. It therefore is a type 1 VMM. The design choice in Xen's case was to produce a kernel as minimal as possible, and delegate most of the hardware interfacing via drivers as well as the control over Xen to a *privileged domain*. (Domains are Xen's name for virtual machines.) Thus it resembles a micro kernel architecture, with process and memory management inside the kernel, and drivers separate from it.

While efforts to port Xen to other hardware platforms, such as ARM, are underway [37], it originally was developed for the x86 series of CPUs. As has been described in Section 2.5, this platform comes with certain limits to the amount of virtualization (in the pure sense) that can feasibly be done. Binary translation has already been introduced there. Instead, Xen uses two other approaches to virtualize on the x86.

[4]See Section 2.5.2 for an explanation of the concept of paravirtualization

2.5.2 Paravirtualization

The privileged domain, also called *dom0* (because it is the first started domain and therefore receives the numerical identifier 0), is generally a Linux that is aware that it is running on top of a Xen hypervisor. As such, it has been modified to work in a very efficient way with the hypervisor. Such a way of virtualizing an operating system is called *paravirtualization*. In this case, not the original OS has been virtualized, but instead, changes in the kernel where necessary have made the interfacing between the virtualizer and the operating system more efficient. Paravirtualization is not limited to the privileged domain, but normal, unprivileged domains can also be run in this fashion. This way of using modified operating systems was the original way to work with Xen. As has been hinted at before, Xen makes use of the protection rings of the x86 architecture. An operating system on this architecture expects to run in ring 0. Since the hypervisor needs control, it should run in a higher privilege level. Since there is no “ring -1”, Xen does the next best thing: It runs in ring 0 itself, and runs the kernel in ring 1 (see Figure 2.7(b)). So the operating system has not only been changed for more efficiency in running on top of the hypervisor, it also has to be fit into its new place. Specifically, code in ring 1 isn't allowed to run privileged instructions. It therefore has to hand over control to the hypervisor whenever it wants to do something that involves these. Xen introduces a concept that is very similar to what applications do when they want to run operations they are not allowed to execute themselves: In Unix, they invoke a *system call* by pushing the relevant data and a number that identifies the requested call onto the stack (or into special registers), and raise an interrupt. The operating system's interrupt handler reacts to it, pops the data, executes the operation on behalf of the application, and finally passes control back to it. If a paravirtualized OS wants to execute a privileged operation, it does something similar: It invokes a *hypercall* that works in exactly the same way, with the difference that the call is serviced by the hypervisor. A hypercall literally is a system call for operating systems. The paravirtualization approach is slightly different on x86-64 machines [52]. Since these have only two rings (see Figure 2.7(d)), the guest kernel has to share a privilege ring with its applications. This has effects on system call handling by the guest OS and memory management, but for this work, the difference is of no importance.

The fact that the operating system is customized into its role has several advantages. It was already pointed out that it can make the virtualized OS faster compared to other virtualization techniques because it can work in unison with the virtualizer. For example, since Xen already has to keep track of the passing of time for its own purposes, a paravirtualized domain can save itself this rather complicated task and just receive time information directly from the hypervisor. In addition, several features become available only in the case of paravirtualization. For instance, operating systems for personal computers generally do not expect the amount of memory to change (although swap space size may change over time, e.g. Linux has the `swapon` command for this). A paravirtualized system can be notified by the Xen Hypervisor that it has received more memory to work with; conversely, the OS can yield excess memory to the hypervisor so that it can be assigned to another domain.

However, there is one obvious drawback to this approach: The OS must have its source code readily available, there must be a (legal) way to distribute the changed sources, and developers must have invested the time to change the operating system for paravirtualization. In the case of Linux, this is no problem: sources are readily available, they are licensed under the GNU General Public License (GPL) which allows for redistribution of modified sources, and the original Xen developers implemented the modifications so they had a privileged domain to run with Xen. This is obviously not true for an operating system such as Windows XP: While during the early stages of the Xen development, there was an in-house port for Windows paravirtualization, the licensing agreements prevented the developers from ever publishing it. As such, while paravirtualization is a very interesting concept, it is only suitable for a subset of virtualization tasks.

2.5.3 Hardware Virtualization

For these reasons, the Xen developer community has recently invested much time into supporting unmodified operating systems [58]. This development was facilitated by the advent of x86 CPUs that support so-called *hardware virtualization* (HVM). The current technologies are called AMD-V [5] in the case of AMD CPUs, and Intel-VT [68] in the case of Intel CPUs. This allows the operating system (in this case Xen) to run other operating systems (Xen domains) unmodified. Conceptually, these technologies implement a virtual "ring -1" that was desired but not available in the case of paravirtualization. The hypervisor runs in a special mode in which it is invisible to the operating system and is allowed to perform a few new additional operations, such as setting aside memory to save CPU states when leaving a virtualized domain, as well as for starting, stopping, and entering it. Thus, the OS can run unmodified and no operations will fail because they are transparently handled by the hypervisor if necessary. In the concept of privilege rings, the setup looks similar to Figure 2.7(e).

The main advantage is that, at least in theory, every operating system ever created for a x86 computer can be virtualized. The disadvantage is additional overhead: Every action that is sensitive as per the earlier Popek-Goldberg definition has to be handled by the hypervisor. The operating system, which is generally expecting to execute those instructions itself, cannot optimize them properly. This means that a context switch has to take place between the virtualized domain and the hypervisor every single time, and context switches are always expensive. In contrast, paravirtualization can employ tricks such as batch hypercalls (multicalls) to reduce the number of context switches.

Also, in order to be usable by the HVM domain, every piece of hardware has to be virtualized, i.e. its behavior remodeled in Xen so that the domain can access it the same way it would access actual hardware. This means that some generic pieces of hardware are modeled that are wide-spread and old enough so that it can be expected that every operating system will have driver support for them. Xen supplies, among others, virtualized versions of a RTL8139 network card and a generic VGA capable display adapter to the HVM domains. The major drawback is that the emulation overhead reduces the performance compared to the paravirtualized case. Paravirtualization employs a concept called "split driver model". As mentioned before, device drivers are not a part of Xen; instead, Xen relies on the privileged domain to provide drivers for hardware access. For hardware access, paravirtualized domains can therefore interface directly with the privileged domain and its drivers via means of shared memory and event notification, which means that the stub *front-end* drivers in a paravirtualized domain are exceedingly simple, straightforward, and (generally) fast. On the other hand, for hardware virtualization, a device access has first to traverse the full driver inside the virtualized operating system's kernel, be handed off to Xen for emulation, and from there to the privileged domain to another driver.

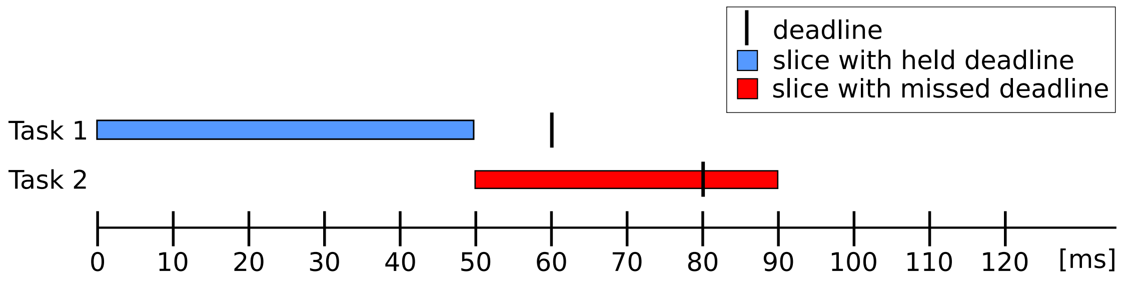
2.5.4 Comparison of Virtual Machine Monitors

Nowadays, users have the choice between several solutions to run a virtual machine on their computer. Therefore, an educated decision has to be made which one to use, which in this case lead to the Xen hypervisor. One of the most popular producers of VMMs is VMWare, Inc. [72] with their line of VMWare products. VMWare Workstation is a type 2 VMM that employs binary translation, while VMWare ESX Server is a type 1 VMM. Microsoft's Virtual PC is a type 2 VMM that is free of charge. None of these have their source code freely available, which ruled them out from the beginning, since it would not have been possible to make the necessary changes to the VMM. Recently, Linux's KVM [44], also a type 2 hypervisor that exclusively works with hardware virtualization, has reached levels of maturity where it can be used for production systems. It would have been another viable choice since the source code is available, but Xen has the advantage of being bundled with a scheduler that can be harnessed more easily for the goals of this work (see Section 2.6). In addition, KVM relays its I/O emulation to (a modified version of) QEMU, while Xen does it inside the hypervisor, which makes it easier to change the values the emulated hardware timers report to the hardware virtualized domains. (Xen's I/O emulation is based on the QEMU sources, but since it resides inside the hypervisor, it is easy to base the time warping on scheduling information from the scheduler).

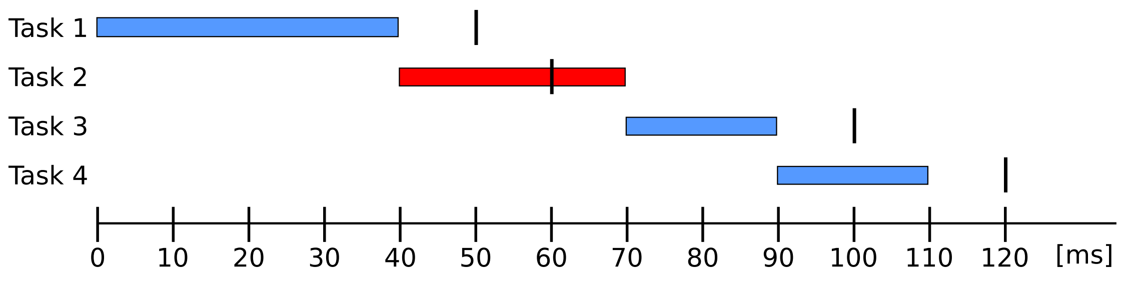
A totally different approach would have been to use a CPU emulator, such as QEMU [12] or Bochs [47], or a full-system simulator, such as Simics [50]. While the scheduling would have been more accurate with them (down to instruction or even cycle level), they are necessarily much slower in their execution, because they do not run any instructions natively on the CPU. It will be shown later in Chapter 4 that Xen's scheduling is accurate enough for network emulation purposes, and instruction- or even cycle-accuracy is not needed.

2.6 The Xen sEDF Scheduler

Xen runs directly on the hardware, and the guest domains run on top of Xen. This is similar in concept to an ordinary operating system, which runs on the hardware, with the user-space applications on top of the OS. Therefore, just as scheduling is important to operating systems, it is important to Xen, with the difference that it is not user-space applications that are scheduled for multi-tasking purposes, but operating systems. Xen comes with several schedulers and leaves it to the user to decide which one to use. For this work, the *simple earliest deadline first* (sEDF) scheduler has been used, for reasons shown later.



(a) Task 2 missed its deadline.



(b) Task 2 misses the deadline. However, task 3 and 4 can hold their deadlines.

Figure 2.8 Two examples of missed deadlines for different values of slice s and deadline d.

Earliest deadline first (EDF) schedulers are typically used in real-time scenarios, because they ensure that every task, if at all possible, will be run for the assigned time (called slice) before the deadline runs out [15]. In the case of one-shot tasks with a certain computation time (slice) s and a deadline d, it can be calculated for each task k whether it can be fully executed before the end of the deadline. This is the case if

$$\sum_{i=1}^{k} s_i \leq d_i$$

Figure 2.8 gives examples of missed deadlines. In the first example, task 2 misses its deadline because $\sum_{i=1}^{2} s_i > d_2$. In the second one, task 2 again misses its deadline ($\sum_{i=1}^{2} s_i > d_2$), but task 3 and 4 can still hold theirs ($\sum_{i=1}^{3} s_i \leq d_3$ and $\sum_{i=1}^{4} s_i \leq d_4$).

In the case of static (i.e. not changing over time), periodic deadlines and tasks that are continuously rescheduled whenever their deadline runs out (this is the way the sEDF scheduler schedules Xen domains), these deadlines are equal to periods. Whenever a deadline is reached, the task is rescheduled in the system with a deadline in the future of $now + period$. In the simplest case, all the scheduler has to do now is run the tasks in order of their deadlines: earliest deadline first (hence the name), then next earliest, etc. Note that it is possible to miss deadlines if the constraints set by the run time (slice) and deadline (period) of each task are unsatisfiable. This is obviously the case if

$$\sum_{i=1}^{n} U_i > 1$$

with the utilization factor of each task $U_i = \frac{s_i}{p_i}$ the ratio between slice and period, since the sum of all fractions of computation time the tasks request is greater than

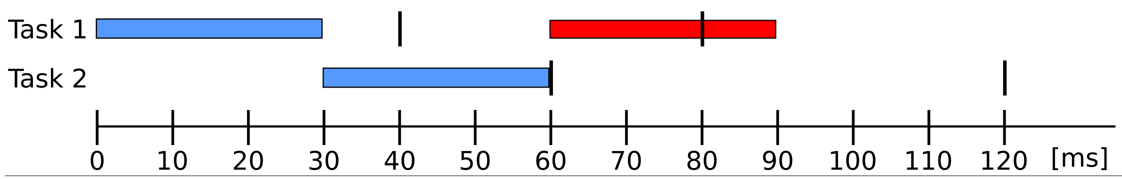


Figure 2.9 Continuously rescheduled tasks may miss their deadline later on. Here task 1 can hold its first deadline, but misses the second.

the amount that is available. Conversely, Liu and Layland proved in [49] that the opposite is also true: a set of n tasks is schedulable if

$$\sum_{i=1}^{n} U_i \leq 1$$

although this abstracts from the fact that the scheduler itself will also consume computation time, and requires that the tasks are preemptible. In many cases, it is not the duty of the scheduler to check for this constraint, but of the user that chooses the tasks to be run and their slices, deadlines, and periods. Figure 2.9 shows an example where task 1 can hold its first deadline, but misses the second (and all further ones—the same is true of task 2) because $\sum_{i=1}^{2} U_i = \frac{30}{40} + \frac{30}{60} > 1$. Figure 2.10 gives an example of deadlines that can e held only if the tasks are preemptible. $s_1 = 20$ms, $d_1 = 60ms$, $s_2 = 10$ms, $d_2 = 20$ms, so there are never 20ms continuously available during which task 1 could be scheduled without task 2 missing its deadline.

In the case of the Xen sEDF scheduler, the tasks that are scheduled are not the actual operating systems themselves, at least not directly. Whenever a domain is created to run an operating system, Xen creates with it one or more *virtual CPUs* (VPUs). These VCPUs are visible to the guest operating system as CPUs, with the effect that a domain with several VCPUs looks like a SMP system from the operating system's point of view. The scheduler then schedules those VCPUs as tasks on the physical CPU(s). It maintains four linked lists for each physical CPU, called PCPU from here on:[5] a *runqueue*, a *waitqueue* and two *extra queues* (the *penalty* and the *utility queue*). These queues contain pointers to the different VCPUs. A simple EDF scheduler is not work-conserving. This means that if all tasks have used their slice at some point, but no deadline has been reached (at which point a task would be rescheduled), the scheduler will have no tasks to run, and spin or run an idle task. This can be seen in Figure 2.10 between 50ms and 60ms. A work-conserving scheduler, on the other hand, will choose one of the tasks and run it in this extra time. Generally, for fairness reasons, it will make sure that over time, every task gets a fair amount of this extra time. sEDF's utility queue is for this purpose, and a domain can choose whether its VCPU(s) are allowed to run during extra time or not.

- The runqueue contains all VCPUs that have not used up their slice this period. They are ordered by their deadline, earliest deadline first.

- The waitqueue contains all VCPUs that have already used up their slice this period, and wait for their deadline so that the next period starts. They are ordered by the start of the next period (i.e. their deadline), earliest first.

[5]While PCPU, as opposed to VCPU, is not a Xen term, we will use it to make clear the differences between the two throughout the text

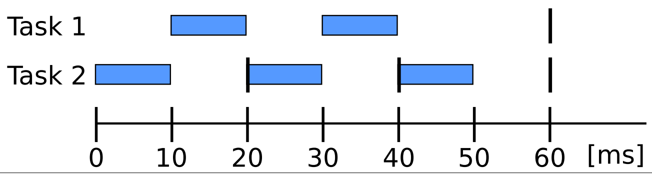


Figure 2.10 Sometimes deadlines can be held only if the tasks can be preempted.

- The penalty queue contains all VCPUs that were not able to run their full slice in an earlier period. While this cannot happen in strict real-time systems, it can in Xen. For example, a VCPU can block if the guest operating system has nothing but the idle task to run, and when that happens, it yields the PCPU back to Xen for use by other domains. The blocked VCPU is then woken up again at a later point when there is work to do, but by that time, it might have missed its deadline. The penalty queue allows domains to catch up on that lost time during extra time. The exact circumstances under which this happens are rather complicated, and are of no concern for the work presented in this thesis. The interested reader can find detailed information in [24].

- The utility queue contains all VCPUs that are aware of (i.e. are allowed to run in) extra time, ordered by scores calculated by a special scoring algorithm. For the same reasons as in the case of the penalty queue, detailed information is not given here and can again be found in [24].

A VCPU can be on several queues at the same time (a typical example is on the runqueue and the utility queue), but never on the runqueue and waitqueue at the same time. Furthermore, while a VCPU can be migrated from one PCPU to another, it can never be on queues of two different PCPUs simultaneously.

Whenever the scheduler is invoked, it will check how long the currently running VCPU has run, and subtract that number from its slice. If this reduces the slice to 0, the VCPU is moved from the runqueue to the waitqueue. It then checks whether there are VCPUs on the waitqueue whose next period has started by now, and moves them from the waitqueue to the runqueue. Finally, it takes the first VCPU on the runqueue and schedules it either for the length of its remaining slice, or until the next VCPU on the waitqueue is ready to be moved to the runqueue, whichever happens first. If the runqueue is empty, it takes the first domain from the penalty or utility queue to run it in “extra time”. This makes the sEDF a work-conserving scheduler since it will never idle, even when all VCPUs’ demands have been met.

To come back to this thesis, the reason why the sEDF scheduler has been chosen over other options that are shipped with Xen is as follows: While many schedulers base their decisions on priority levels of tasks, an EDF scheduler directly operates on time values. Since one of the aims of this work was to let domains run for exactly specified amounts of time, it comes as a natural choice. Scheduling (at least in theory) becomes as easy as setting the slice size in the scheduler to the slice size of the synchronizer.

2.7 Timekeeping

Every computer system from a certain size upwards, most definitely a x86 system, needs to be able to keep track of the passing of time. On a macro scale, users expect to be informed about the time of day, and services that perform tasks such as defragmentation, backup, or indexing need to be started once a day, week, or month, preferably at off-times. This time is called the *wall clock time*, since, just like a clock hanging on the wall of a room, it measures the passing of time down to a resolution of seconds, and up to hours, days or years. On a micro scale, a system must be able to measure the passing of time down to micro- or even nanoseconds—for example for exact scheduling—and interfacing with hardware with timing constraints. This time information is typically saved (if not measured at this accuracy) in nanoseconds since booting, and is called the *system time*.

Wall clock time is relatively straight-forward to measure and keep track of. Every x86 system has a real-time clock chip (RTC) that contains a clock with a visible resolution down to seconds (internally, a 32.768 kHz gives the RTC a resolution of $\frac{1}{2^{15}}\mathrm{s}$). It is battery-buffered to keep time while the computer itself is turned off. This time can be read from the RTC, or it can be set by the computer to update it, for example from information by the network time protocol.

System time is much more complicated. There is not only one hardware device to aid in timekeeping on this micro scale, but (depending on the production date of the computer) up to five. All of these do provide a clock that measures the passing of real time, but are either counters that are increased at high rates, or timers than can be set and signal an interrupt to the system when they expire.

The first one is the time stamp counter (TSC). Since the Intel Pentium, processors have an internal counter that is increased with every clock signal (while the clock signal itself would be a way to measure time, too, it is not accessible by any soft- or hardware) and can be read via the `rdtsc` instruction. While this counter can in theory produce very accurate results (with a 1 GHz processor speed, it increases every nanosecond), in practice, the results have to be taken with a grain of salt. The reason is that the system has to calibrate the TSC results against real time to measure how many TSC increments are equal to a certain period of real time. For instance, newer processors might not increase the TSC with every clock cycle, but only every 2 to 4 clock cycles. Furthermore, if there is any kind of power management that can decrease CPU speed, this will mirror in the TSC values: They will now increase at lower speed than before. And on symmetric multiprocessing systems, each CPU has its own TSC, and there might be significant differences between the TSCs on the different CPUs.[6] While newer processors try to remedy these shortcomings by having TSCs synchronized between all CPUs, and increasing at a steady rate independent of the current CPU speed, this is of little help to operating systems, since they have to be able to also cope with older TSC implementations. Nevertheless, since the TSC is the counter with the finest granularity in most x86 systems, it is still used in some situations.

[6]To remedy the problems with nonuniform TSCs on SMP systems, both Linux and Xen will do a calibration during bootup (see `smpboot.c`) during which all CPUs will run at the same time, for the same amount of time, and the differences in the counters are measured.

The second time source, the ACPI Power Management timer (ACPI PMT or simply PMT) is another counter device that is available on all computers that support ACPI. Compared to the TSC, it has a fixed frequency of 3.58 MHz at which it increases its counter. This makes it preferable for most applications, since the frequency cannot change, as can be the case with the TSC. Nevertheless, in a select few instances, the resolution might not be high enough.

Third is the programmable interval timer (PIT). As the name suggests, it is a timer that can be set to a certain interval, and will then issue an interrupt to the system periodically whenever it expires.[7] Its availability in all x86 systems and reliance on a quartz crystal (which makes the intervals independent from anything else that may happen in the system) make it one of the most-used timers. For example, Linux uses it as its main timer interrupt in many configurations.

The local APIC (advanced programmable interrupt controller) can provide another timer that can be programmed for periodic interrupts. The time source is not a quartz crystal, but the bus frequency, which makes programming somewhat more complicated, since it is not the same on all x86 computers. However, it allows for much longer intervals between timer interrupts because the counter is 32 bits in length, compared to 16 bits for the PIT. The main advantage, however, is that in multiprocessor systems, each CPU has a local APIC, and the timer interrupt will therefore always be handled by the same CPU. This allows to set up timers for each CPU, which for example allows to handle per-CPU scheduling via this timer. Furthermore, the dependency on the bus frequency facilitates a much higher resolution (typically around 1 μs). This is the basis of the Linux “high resolution timers”. It is also used by Xen and one of the cornerstones that facilitate the synchronization presented in this thesis, by allowing scheduling down to very small time slices.

Recently, the High Precision Event Timer (HPET) has been introduced and starts to be included in x86 systems. It was designed to replace the PIT and has several advantages over it, most notably more timers (while the PIT has three programmable timers, only one can generally be used by the OS) that can be programmed to different intervals, and a much higher resolution.[8] The downside is that most computers still do not have a HPET, therefore the operating system has to provide other timekeeping ways on most systems.

Finally, the RTC also supports a periodic timer mode, which in resolution and precision is almost equal to the PIT.

It should be clear from this list that timekeeping is all but an easy job for an operating system. It has to probe which sources are available, choose from them, calibrate them against each other, and possibly correct values that drift from each other, always deciding which source to consider the more reliable.

Timekeeping is a two-fold issue for Xen. On the one hand, it has to work with all these time sources to keep track of time itself. On the other hand, it has to

[7]The PIT is by far the oldest timer hardware and has existed since the first incarnations of the IBM PC. Its age shows from several design decisions: The frequency of the quartz crystal was derived from NTSC television standard to aid the graphics adapter in output on a TV screen; furthermore, one of the channels inside the PIT is used to drive the PC speaker.

[8]The HPET specification [40] requires a counter that measures the passing of time in femtoseconds (1 ns = 10^6 fs). Unfortunately, clock drift for every measurement time lower than 100 μs is allowed to be up to 200 ns, or 0.2%, which can add up quickly.

provide time information to the virtualized domains. Paravirtualization is the more straightforward case here, since the domains are changed to work in unison with Xen. Wall clock time is exported by the hypervisor into a shared memory page, from which it can be read by the paravirtualized OS. The same happens with a value for “time elapsed since system started”, which can be used for time measuring. And finally, the periodic timer interrupt is replaced by a timer inside Xen that signals the OS periodically.

The hardware virtualized case is more complicated to implement, since the domain is oblivious to the fact that it runs virtualized. Therefore, all these timers have to be emulated in software[9] and their programmed intervals have to be kept track of via timer queues inside Xen. Since they all work differently in the way their interface is designed, their timing constraints, and so on, this is a considerable implementation effort. For this work, it also means that changing the way the timekeeping works (as will be described in Section 3.4.2) means changing how each of these operates.

[9]In theory, some of them could be just ignored, so the domain would get the impression that the system does not have them. However, Xen does emulate all of these.

3

Implementation

In this chapter, the design and implementation of the work done for this thesis will be explained part by part. The author has tried to find a middle ground between being too broad and too specific. As such, not every programming trick, variable, or function will be discussed. A more detailed hands-on list on what configuration variables are available for the different parts of the system and how to use them can be found in Appendix A.

Figure 3.1 shows the overall setup that was implemented for this thesis. The synchronization server will be explained in Section 3.1, the synchronization clients in Section 3.2.1 (Xen side) and Section 3.2.2 (OMNeT++ side), respectively. The way the data communication is facilitated between both entities is laid out in Section 3.3. Finally, changes done to Xen to drive the scheduler in the desired way, and to the time representation for synchronized Xen domains will be described in Section 3.4.

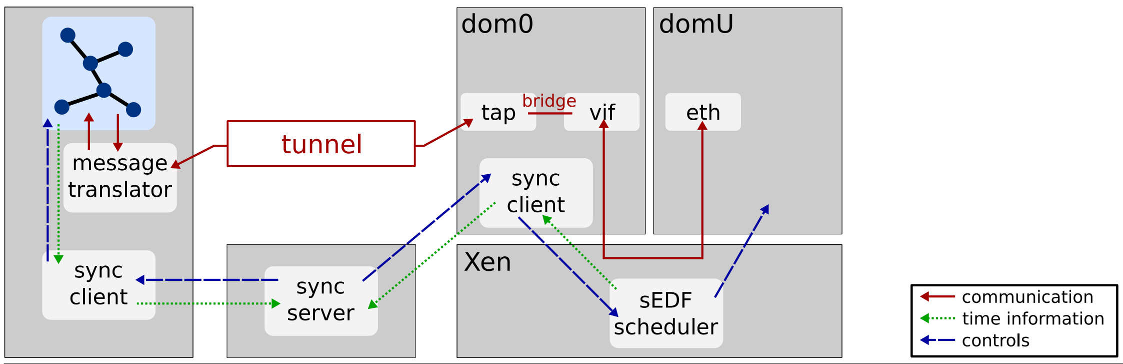


Figure 3.1 The synchronized network emulation setup that was implemented.

3.1 The Synchronization Server

While other pieces of work are much more involved and complicated, the synchronization server is the heart piece of the implementation. Its task is to keep track of all domains and simulations that have signed up for synchronization, and make sure that their clocks will be synchronized. The server can be run on any machine, including but not limited to one of the machines that partake in the synchronization (via a simulation or domain running on them). As has been explained in Section 2.4, CTW synchronization requires communication between the synchronization server and the synchronized components. To facilitate communication between them, there is a synchronization client running on every machine that is part of the synchronized network (see Section 3.2), and a simple communication protocol is used. There are only four different messages sent between server and clients, and they are sent as UDP packets:

- A "register" message (`reg`) is sent by the client to the server when it signs up to be part of the synchronization.
- An "unregister" message (`unreg`) is sent by the client to the server when it leaves the synchronized network.
- A "run permission" message (`run`) is sent by the server to all the clients when it starts a new time slice.
- A "finished" message (`fin`) is sent by the client to the server to notify it that its domain or simulation has finished the assigned time slice.

The process is therefore as follows: When the synchronization server starts, it will wait until the first client connects. When this happens, it will send out a `run` message. The client receives it, instructs the domain or simulation it controls to run for the assigned time, and afterward will report back with a `fin`. At this point, there is only this one single client registered at the server. Consequently, after receiving the message, it will immediately send out another run permission. This means that at this point in time, the client is synchronized to only itself, which is hardly useful.

However, as soon as another client registers at the server, meaningful synchronization starts: The new client will have to wait until the first one has finished the currently assigned slice and reports back to the server by means of a `fin` message. The server will now instruct both clients to run. Then it will wait until both have reported back. Only then will it allow both clients to run again. So, if one of the clients drives a Xen domain (or a fast simulation), and the other a simulation that is very complex and slow, the former will now be required to wait for the latter. This achieves the CTW synchronization as described in Section 2.4.

There is one twist to how `run` messages are sent: They are not addressed to the IP address of the computers housing the client. Rather, the server will send one broadcast message. This has a twofold benefit: First, it means that network traffic is kept low, since only one message has to be sent over the line, regardless of the number of clients. Second, it allows each computer to receive the message at the same time. It has been pointed out in Section 2.4 that the CTW algorithm cannot prevent skewing of packet travel times as they are perceived by the node, if the skewing is lower than the slice time (which was established as the upper bound of possible skewing). While this is a fundamental problem, and any measurements that try to rely on or gauge values to a precision below slice time should be taken with a grain of salt, sending out one broadcast message allows for at least some amount of best-effort precision compared to necessarily sequential sending to each client individually, in that all client start at roughly the same time. “Roughly” because there are still a plethora of factors that influence this: The packet has to traverse a network stack, to be handed to the synchronization client and interpreted, and the client then has to start the domain or simulation it drives. This is therefore purely best-effort and might or might not improve results at sub-slice accuracy; the better way will always be to reduce slice time according to the desired synchronization precision.

For this work, two synchronization servers have been written. The original, standalone one was written by Elias Weingärtner before the author started his thesis. Additionally, the kernel module that houses the Xen synchronization client (see Section 3.2.1) can also be used as a server.

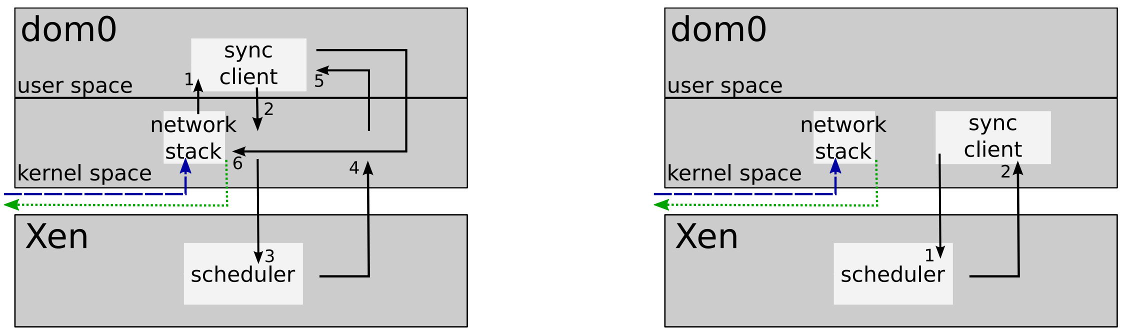
3.2 The Synchronization Client

For this work, two clients have been written: one for driving Xen domains, and one for the OMNeT++ discrete event scheduler. First of all, their common behavior will be explained, before the implementation of each is described in more detail.

The task of the synchronization client is to form the interface between the synchronized domains or simulations and the synchronization server. Between client and server, communication is done via a simple protocol on top of UDP, as described in Section 3.1. Whenever a `run` message is received, the client will instruct the scheduler that drives the respective domain or simulation to run it for the specified amount of time. It will then wait for notification that the slice was fully used up. As soon as that happens, the client will send a `fin` message to the server.

3.2.1 The Xen Synchronization Client

First of all, it should be noted that the synchronization client for Xen domains has been written as a Linux kernel module that resides in the privileged domain’s (dom0’s) kernel. The reason for this are efficiency and speed considerations: When the server sends out the `run` message, it will enter the Xen machine via the network adapter, and then the network stack inside dom0 . If the synchronization client was implemented in user space, as depicted in Figure 3.2(a), a received `run` packet would then traverse the stack, and finally be handed over to the client, which has opened a socket on the respective port that the packet was sent to. This is the first necessary context switch. Then, the client has to interpret the contents of the message, and afterward instruct Xen to run whichever domain is synchronized on the machine. This instruction is done via a hypercall (for the concept of hypercalls, see Section 2.5.2), and is only possible from kernel context. This means that another two context



(a) Sync client in user space (b) Sync client in kernel space

Figure 3.2 Placing the synchronization client into kernel space noticeably reduces the number of context switches.

switches are necessary for entering Xen's context: From user-space to kernel space (2), and from there on to Xen's space (3). Also, when the synchronized domain has run its assigned time, Xen must notify the client. This is done via an interrupt, and interrupt handlers have to reside in kernel context. So again, two context switches would be necessary here: from Xen to kernel context and the interrupt handler (4) and onward to the user-space client (5). Finally, to inform the synchronization server that the assigned time has been used up, a packet has to be sent, which requires another context switch back to kernel context and the network stack (6). Implementation of the synchronization client as kernel module therefore saves four context switches (see Figure 3.2(b)). In this case, only two context switches are necessary: from kernel context to Xen when the scheduler has to be instructed (1), and back to the interrupt handler when the domain has run its assigned time (2). While the exact effect of context switching on the performance is hard to measure, since it depends on many factors (architecture, CPU speed, CPU cache size, just to name some), they are generally expensive operations. Saving context switches therefore improves performance.

The fact that the client resides in kernel space makes a few things a bit different to handle. Most notably, everything surrounding sockets, such as their creation and sending and receiving messages, is different. This is because the functions `socket()`, `sendto()`, etc. are syscalls that cannot be issued from the kernel. Rather, the generic back-end functions have to be used. Also, since standard C libraries are not available, some functions have to be copied over and statically linked into the module.

When the kernel module is loaded, it will first open a socket for communication with the synchronization server. Then, it will send a hypercall to Xen to set domains that it will control into synchronized state. Instead of creating a totally new hypercall for this, it rather makes use of the generic hypercall DOMCTL (domain control). DOMCTL is a very diverse hypercall which can execute a plethora of different commands based on a cmd option that is handed over with the other hypercall options. Two commands, `XEN_DOMCTL_set_synced` and `XEN_DOMCTL_get_synced`, were added. After this is done, the kernel module will register an interrupt handler. Finally, it will start a kernel thread that contains the actual client code. From there, it will then send a `reg` message to the server, and wait until it receives the first `run` permission.

Whenever it receives one, it will use another hypercall to interface with the scheduler. Again, the DOMCTL hypercall was reused, but in this case, instead of adding another command, an already existing one was expanded for our purposes: `HYPERCALL_domctl(scheduler_op)` originally enabled the privileged domain to set and adjust scheduling options for all domains. For the synchronized case, this was changed so that a call to this will not only set these scheduling options, but also signal a new slice. Besides the fact that this minimally invasive change to Xen will probably make it easier to maintain the code for later releases,[1] this has the beneficial side effect that every slice can be of different length compared to its predecessors and successors, if desired. After issuing the hypercall, it will yield the CPU, so that the synchronized domain can start immediately.

After Xen has scheduled the domain for the specified time, the scheduler will send an interrupt to dom0 (see Section 3.4.1.2 for more details). The interrupt handler that was registered will be called, and trigger the sending of a `fin` message to the synchronization server. Then it will wait again for the next `run` permission. This loop will continue infinitely, or until the kernel module is unloaded. In this case, the exit function will close the kernel thread and the socket, unregister the IRQ handler, and use the `XEN_DOMCTL_set_synced` hypercall to reset the sync flag, which releases the domain from synchronization.

3.2.2 The OMNeT++ Synchronization Client

The modular design of OMNeT++ allows the user not only to create new nodes with custom behavior for simulation, but it also gives the opportunity to implement and use new basic classes. To create the synchronization client, a new scheduler was written. In contrast to the Xen side, synchronization client and scheduler form a unit; the scheduler performs all the tasks of the client in its scheduling functions. Like all other components in OMNeT++, it is implemented by creating a new class (in this case named `cSyncScheduler`) as a subtype of a more general, predefined class (`cScheduler`). Similar to how the Xen scheduling subsystem works, OMNeT++ will call the chosen scheduler's functions at the appropriate times.[2] For example, at the beginning of the simulation, (i.e. when OMNeT++ is started), the method `startRun()` is called. Here, the tunnel for the external client (see Section 3.3 on the functional coupling via the emulator) is initialized, and the simulation instance is registered at the synchronization server.

After Initialization, OMNeT++ will repeatedly call the main scheduling function `getNextEvent()`. As the name suggests, it is the scheduler's task to hand over the next event that is to be processed. In the case of the cSyncScheduler, it will block

[1]The main reason that adding a new hypercall might break in later releases is that every hypercall is identified by a unique number that correlates to an offset in Xen's hypercall vector. For example, when the DOMCTL hypercall is to be executed, the system notifies Xen that it wants it to execute hypercall no. 36. If a new hypercall is added, it has to be uniquely identified with a number. If in a later version of Xen, a hypercall was added by the developers that uses the same number, this conflict had to be resolved manually.

[2]The only difference is that OMNeT++ calls a method of a class, while this object-oriented concept is not available in C, in which Xen is programmed. Xen uses function pointers set in a globally declared `struct` to achieve the same behavior, a typical approach in C for this sort of functionality.

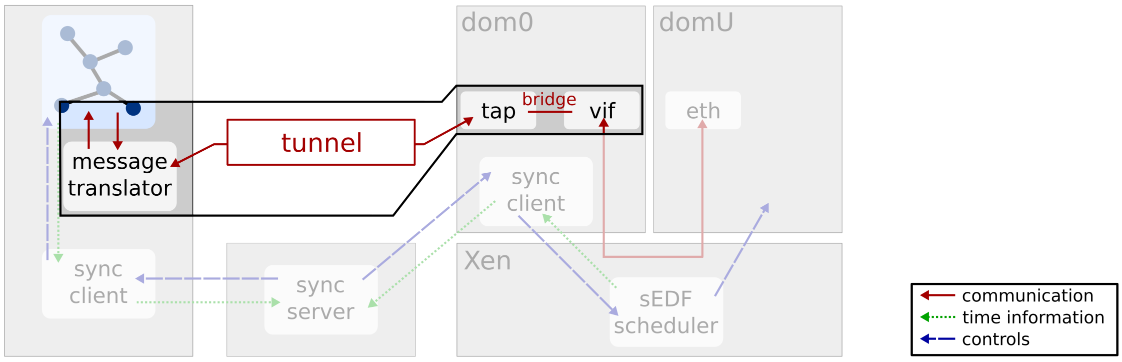


Figure 3.3 The parts of the synchronized network emulation setup that constitute the emulator.

and wait for a `run` message from the synchronization server. Once it receives it, it will check whether the next event's scheduled processing time in the event queue is earlier than the barrier set by the run permission. If so, it will return this event to OMNeT++. The simulator will process it and ask the scheduler for the next event. This will continue until the next event's scheduled processing time is later than the barrier. The scheduler will then send a `fin` message to the synchronization server and wait for the next run permission, blocking the simulation again until this happens. This cycle will repeat ad infinitum, or until the user stops the synchronized network emulation. If the simulator is closed, it will call the function `endRun()`, in which the scheduler sends an `unreg` message to the server to notify it that it is leaving the synchronization.

3.3 The Emulator

The emulator performs the tasks that facilitate the functional coupling between simulation and Xen domain, i.e. it converts Ethernet frames sent out by the domain's network driver into simulations messages and vice versa. It comprises several parts, which have been marked in Figure 3.3 and will be described in this section: a bridged TAP device, a tunnel, and a message translator.

On the Xen side, acquiring the Ethernet frames sent by the synchronized domain is done via a TAP device. Whenever an unprivileged domain, named "domU" in short, is started in Xen, and it has been defined to have one or more Ethernet devices, a corresponding so-called "virtual interface" (vif) is created within the privileged domain dom0. All traffic to and from the domU is also visible on this device; in fact, it is a vital part of the interface setup in Xen. It is bridged with the actual physical network device in dom0 to facilitate connection to the outside world.

For emulation, this is changed so that the virtual interface is bridged with a TAP device. TAP devices [45] are front ends to a virtual network driver that allows to tap into (hence the name) the communication on Ethernet frame level. The TAP driver creates a special device file from which the frames can be read, or to which frames can be written to insert them. The emulator uses this for communication with the synchronized domain. The frames are read from the device and encapsulated into a UDP packet, so they can be tunneled to a different host; in our case, this is the host the simulation is running on. This is done because the IP address of a simulated node is not known to the physical network, and packets addressed to it would not be routable. In contrast, the machine that runs the simulation has an IP address that packets can be sent to. Conversely, UDP packets that are received over the tunnel are decapsulated, and the Ethernet frames inside the packet are inserted into the TAP device, and therefore bridged to the virtual interface and by extension the synchronized domain.

The simulator side of the emulator is more complex, since all the actual translation takes place there. When a packet is received over the tunnel, it is decoded step by step and transformed into an OMNeT++ message. The conceptual design of an OMNeT++ message in the INET framework is similar to that of a real packet or Ethernet frame: A packet consists of a header and a payload, and the latter may contain another packet type, again with a header and payload. Just like TCP is encapsulated into IP when it traverses a TCP/IP network stack, an INET framework message consists of several types encapsulated into each other: A message of type `EthernetIIFrame` will contain a pointer to an encapsulated message, typically of the type `IPDatagram` or `ARPPacket`, and so on. While the INET framework ships with some serializers to do this translation work, in practice they are incomplete and unsuitable for our tasks; for example, ping data payload that an ICMP `echo request` packet is padded with is silently discarded. This means that an `echo reply` sent by a node inside the simulation cannot return this data payload, which is a violation of the respective RFC [57] and can lead to problems, such as the original sender not properly recognizing the `echo reply`. Consequently, a new translator was implemented by the author.

After the translation has finished, the message has to be inserted into the simulation. To facilitate this, every synchronized domain is represented in the simulation by a node of the newly created class `cExtHost`. Whenever a packet is translated into an OMNeT++ message, it is inserted at this node, and from there handled exactly the same as all other INET framework messages. In other words, to simulated nodes in the network, `cExtHost` does not behave any different from a fully simulated one. When a message is received by the `cExtHost` node, it will be handed over to the emulator for translation in the opposite direction, from message to packet. The work for this direction of translation was already done a few years ago by another student in [59] and adopted. For the most part, it could be used in the state it was in. However, support for ping data payload was again missing and added.

At this point in time, the emulator can translate the protocols ARP, UDP (and consequently IP), ICMP and Ethernet (in the common format of version 2). The interface is extendable and should accommodate additional protocols without any problems.

3.4 Modifications to Xen

For this thesis, Xen has been modified in two fields: the scheduler and the time-keeping. These changes comprise the main part of the work, both in complexity and amount of time invested. They can be roughly split into two areas. Changes to the scheduler allow to precisely start and stop the synchronized domain, in fulfillment of requirement 1. The sEDF scheduler was modified for the special treatment of synchronized domains. To improve scheduling performance, the main scheduling loop that wraps the sEDF scheduler was also modified. Changes to the timekeeping allow to mask the passing of time from a synchronized domain, to the point where it will not notice time has passed while it was descheduled, fulfilling requirement 2. How this is done depends on whether the domain is para- or hardware virtualized, and whether counters or timers are modified.

3.4.1 Modifications to the sEDF Scheduler

As described in Section 2.6, the sEDF scheduler is an earliest deadline first scheduler that does its work by scheduling the virtual CPUs (VCPUs) that belong to a domain. It uses four queues per physical CPU (PCPU): a wait- and a runqueue, as well as two extra queues. A VCPU will be assigned to a PCPU and never migrate to another (save for manual intervention), and switch between run- and waitqueue there, depending on whether it still has time to run during its slice, or is done and waiting for its next period to begin. If VCPUs are "extra-aware", they are also put on the utility extra queue. If the run queue is empty (i.e. all runtime demands have been met for the time being), sEDF will take a VCPU from the utility queue, so that the PCPU does not idle.

To understand the changes to the scheduler, and their reasons, remember the first requirement of the Xen implementation:

1. The Operating System must be stoppable and startable at any point in time, and run for precisely the amount of time assigned.

To synchronize an operating system, it is therefore important to make sure that it will never run during this extra time. That means that its domain's VCPU(s) must be removed from the utility queue. Fortunately, this is very easy to ensure, since the sEDF scheduler already comes with an option to make VCPUs extra-aware or not.

It is also important to make sure that the VCPU of the synchronized domain is placed outside the normal switching between run- and waitqueue. Normal domains fulfill the constraints laid out in Section 2.6: They have a constant slice and relative deadline, and are constantly rescheduled when their deadline is reached. In contrast, in this implementation, there is no immediate rescheduling when a synchronized domain has finished its slice. Rescheduling occurs only when the synchronization server gives the instruction to do so. Therefore, this implementation introduces another queue which is called the "syncqueue". VCPUs of domains that are synchronized are put onto this queue, which functions as an alternative to the waitqueue. They are kept on this queue until the synchronization server assigns a new time slice. Then, the slice is set to the time that was assigned, and the VCPU is put onto the runqueue. After it has run its course, instead of onto the waitqueue, it is put back onto the syncqueue.

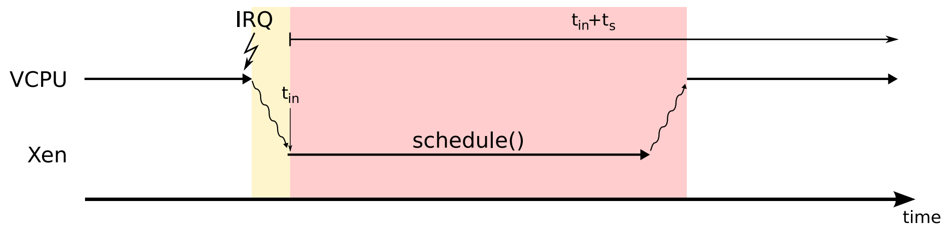
This has two advantages: First, synchronized domains are kept outside of the scope of the scheduler while they are dormant, which means that they cannot influence the actions of the scheduler during this time. This was done so that the changes to the scheduler are kept small, which hopefully will mean that the changes to Xen will be easily portable to newer versions that will be released in the future. Generally, the code was written in a way that tried to extend Xen in a way that changed as little of the already-existing code, and rather added new code specifically designed for synchronized domains only. Second, keeping the synchronized domains and their VCPU aside in a separate queue allows for fast checking and finding of them. If a system had many domains running, but only few of them were synchronized, a separate queue will allow to traverse the synchronized VCPUs much faster than checking every single one on the waitqueue. In retrospect, the speed advantage is probably negligible, but it allowed for a convenient partition into synchronized and unsynchronized domains. In addition, it fit into the multi-queue concept of the sEDF scheduler.

3.4.1.1 Misattribution of Runtime

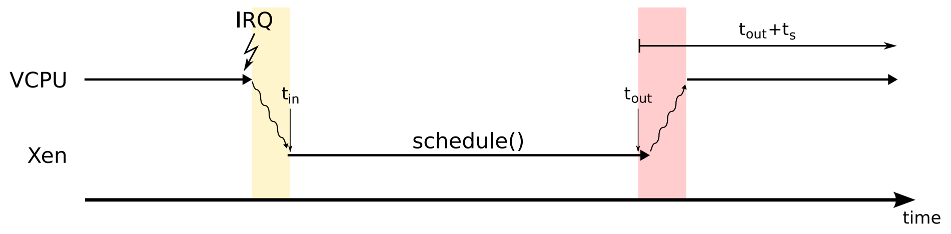
Xen gives the administrator the choice between several scheduling algorithms. As has been stated numerous times by now, this work adopted the sEDF scheduler. This choice is implemented by having a main scheduling loop that contains fixed code, and that calls the chosen scheduler only when the actual scheduling decision (which VCPU, for how long) has to be made; all the other work (for example, context switches, and maintaining the timekeeping system) is done at appropriate times inside the main loop, but before control is given to the chosen VCPU.

One of the main problems of this main scheduling loop, and that the sEDF scheduler therefore inherits from it, is that it attributes time spent in the scheduler to the VCPU chosen for scheduling. So, if scheduling decisions and all the additional upkeeping inside the scheduling system take $2\,\mu$s, the domain will run $2\,\mu$s shorter than the time it is scheduled for. Under normal circumstances, this effect is negligible, since the scheduler will rarely run a domain for less than $500\,\mu$s; besides, the effects will affect all domains more or less the same. Consequently, this inaccuracy was ignored by the Xen developers. However, in our cases, slices may be as small as $10\,\mu$s, which exacerbates the effect, and requirement number 1 explicitly states that domains are supposed to be run for exactly specified amounts of time. Therefore, the scheduler must be made aware of the time it consumes itself.

Here lies a fundamental problem. It is impossible to ever make the scheduler 100% accurate in measuring this time. The reason is illustrated in Figure 3.4. The behavior of the original scheduler is depicted in Figure 3.4(a): It takes a timestamp t_{in} at the time it enters the main scheduling function. It then calls the sEDF scheduler to reach a decision on which VCPU to schedule, and for how long (t_{s}; not depicted in the figure). Then, it will do various administrative work connected to this VCPU. Finally, it will set a timer to $t_{\text{in}} + t_{\text{s}}$, which will give control back to the scheduler; after returning there, it will attribute t_{s} as time the VCPU ran, which is obviously inaccurate on two accounts. First, because the time spent in the scheduler is attributed to the scheduled domain (the red area). Second, because the time between the scheduler interrupt occured and t_{in} was taken is attributed to the preceding domain (the yellow) area. However, when trying to remedy the misattribution by introducing another value t_{out} (see Figure 3.4), a problem occurs: At best, the timer can be set to $t_{\text{out}} + t_{\text{s}}$. This means that the timestamp has to have been made before the instruction to set the timer, which in turn has to be made before the final context switch between scheduler and VCPU occurs. Therefore, there is always a small



(a) Before: Time in the scheduler attributed to the following domain.



(b) After: Time in the scheduler not attributed to any domain.

Figure 3.4 Misattribution of runtime. Note that the figure is not to scale.

amount of time that will be incorrectly attributed to the scheduled VCPUs due to the context switches. The solution to add a timestamp t_{out} to the already existing t_{in}, however, alleviates the problem by a great amount. Note that the two figures are not to scale. In reality, the time spent on the context switches is much shorter in relation to the time spent in the scheduler. It has been exaggerated here for reasons of clearness in the figure. In the source code, the two timestamps are saved in the C structure `struct vcpu`, as `sched_start` and `sched_end`, respectively. Those two variables play an important role in the timekeeping changes that are described in the next section.

Several changes in the sEDF scheduler stem from this change. Most important, whenever the sEDF is called from the main scheduling loop, it will update its information, so it can keep track of which VCPU has already run for how long. It will not simply assume that the time it assigned to the VCPU is the time it actually ran, since there are several situations under which the two are not the same (e.g. Interrupts occurred, or the domain blocked and yielded the PCPU). For these purposes, sEDF calculates the difference between the previous time it was called (which is equal to t_{in}), and the current time. This had obviously to be changed, too. While the above change to the timer that recalls the main scheduling loop makes the VCPU run longer, it has to be ensured that the extra time gained this way is not deducted from the VCPU's time slice. Otherwise it will be punished by the sEDF next time around for seemingly running too long. The change is, similar to above, a switch of t_{out} for t_{in}.

One final thing that has to be kept in mind is that timers are not 100% accurate. If a domain is scheduled for 100 μs of time, it is not uncommon to see it return to the scheduler shortly before or after (on our test computers, it was generally in the sub-μs range, but could sometimes peak at 3-4 μs difference). Therefore, some self-adjustment was added. If the scheduler returned too early, the VCPU is generally simply rescheduled in the normal way for the remaining time. However, if it had run too long, the scheduler takes note of it and will reduce the next slice by the appropriate amount. Furthermore, if the VCPU had returned too early, but barely so (less than 1 μs too early), it is not rescheduled; rather, the lost time is added as a bonus next time it is scheduled. The reason for this is two-fold. On the one hand, since t_{out} is not exactly the time the VCPU starts running again, there is always a small error introduced (as has been already explained). For slices of a size this tiny, the error would become rather noticeable; therefore, it is preferable to add it to the next slice to reduce the overall error. On the other hand, timers in Xen with expiry dates very close to the current time have the tendency to fire immediately. In this case, the following happens:

1. The rescheduling timer is set.
2. Immediately, it fires, and the main scheduling loop is called again.
3. It in turn invokes the sEDF.
4. A very small amount of time has passed since the previous invocation; this is subtracted from the scheduled VCPU's time slice. The VCPU is chosen to continue to run.
5. The main scheduling loop sets the timer.
6. The cycle continues from step 2, until all the VCPU's remaining time has been used up.

In this case, the VCPU loses all its remaining time, does not run at all, and the system produces meaningless scheduler overhead. Since this obviously is undesirable, the above solution was chosen.

3.4.1.2 The sEDF Interface

When the domain has used up its assigned slice, the scheduler needs to signal this, so that Xen can inform the synchronization client in dom0, which in turn will relay this information to the synchronization server. Since the knowledge about when this happens resides in the sEDF, it will signal this to the main scheduling loop. The design choice was again to make this signaling as simple, nonintrusive and efficient as possible. Therefore, inside the C structure `struct domain`, which already existed and contains domain management variables, a counter was added that is initialized to 0. When a synchronized domain has finished its assigned time, the counter is incremented and therefore odd. The main scheduling loop will always check the counter for oddness, and if the case, increment it again (to make it even), and send out a so-called "virtual interrupt" (see Section 3.4.3) to the dom0, in which the synchronization client resides. From the view point of a Xen domain, a virtual interrupt is no different from normal interrupts, i.e. the synchronization client has registered an interrupt handler, and therefore immediately is noticed whenever a domain is done.

Conversely, the synchronization client, whenever it receives instructions from the synchronization server to let a domain run for a specified amount of time, has to have a way to interface with the sEDF. For this end, a hypercall is used. For details, see Section 3.2.1.

3.4.2 Time Warping: Modifications to the Timekeeping Subsystem

It has been explained in Section 2.7 that every x86 system makes sure that it can keep track of the passing of time. For synchronization that is truly transparent to the synchronized OS, the time information that can be received by it must be changed to fulfill requirement 2:

2. The Operating System must not notice that, during the time it did not run, any time passed. In other words, although it will run only intermittently, it must seem to it as if time passed continuously without any gaps.

Also in Section 2.7, it was explained how timekeeping hardware comes in the form of either counters (the real time clock being a special form of a counter) or programmable timers. To achieve what will be called time warping from here on, i.e. proper modification of the timekeeping information, those two types have to be dealt with in different manners.

Counters must not increment while the domain is not running. This means that in contrast to the normal way they are handled by just replicating the values of the actual hardware counters, for synchronized domains, a special value Δt has to be calculated that is the cumulated amount of the time the domain has not run since it was put into synchronized state (therefore the "big delta"). Then, whenever information about these counters is relayed to the domain, Δt is subtracted from the actual value.

Timers must, on the one hand, not expire while the domain is not running. This can be ensured by stopping all of them whenever the domain is descheduled. On the other hand, their expiry date must also be adjusted. If they were simply stopped when a domain was descheduled, and then restarted when it was scheduled again, they would expire at the wrong time, with a high chance to do so at the very beginning of the time slice. Therefore, every time a synchronized domain is scheduled, a value δt has to be added to the expiry date of all timers, where δt is the time since the domain has been last run, i.e. since it was descheduled (the "small delta").

To calculate Δt and δt, the timestamps `sched_start` and `sched_end` are saved during scheduling. These are the variable names used in the Xen implementation for the two timestamps t_{in} and t_{out}, as defined in Section 3.4.1.1. In addition, the main scheduling loop saves a third timestamp, `sync_time_last_run`. While it contains the same value as `sched_start`, it is saved in a different location and serves a different purpose: The first two values are saved with the VCPU that is going to be scheduled, while the third one is saved with the VCPU that has just been descheduled. This makes it possible to calculate the time since a domain has last been run as the difference `sched_end`−`sync_time_last_run`, which is the definition of δt. Δt is updated every time a domain is chosen for scheduling. Whenever this happens, δt is added to Δt. Often during the implementation, two more values are used for time warping: `sync_total_run` is the counterpart to Δt in that it contains the cumulated time the domain has run (in contrast to the time it has not run), and `sync_original_start_time` is a timestamp taken at the time the domain was put into synchronized state. Note that `sync_total_run` is directly calculated from the values of `sched_start` and `sched_end` inside the scheduler: When a domain is descheduled, `sync_total_run` is incremented by `sched_end` − `sched_start`, where `sched_end` still contains the old value, but `sched_start` already the new.

In theory, time warping is therefore hardly a complicated task. The main problems are that the approach to time warping is completely different between para- and hardware virtualized domains, since their timekeeping architecture is likewise dissimilar. In addition, the hardware virtualized case requires full emulation of all hardware counters and timers, which work very differently internally. Finally, it is important to base all the calculations on as few timestamps as possible. Whenever a timer is warped, the current time value, in short “now” from here on, is important to know. Nevertheless, it is better to acquire one timestamp for “now” first and then do all warping with this value, even if there are many timers to be warped. Even though every warping takes a few nanoseconds, and therefore “now” is actually already in the past, acquiring a new timestamp for every counter would only achieve two things: First, it would increase the overhead, since every call to acquire a new timestamp costs time. Second, it would mean that timers would, even if very slowly, start drifting from each other.

Take as an example two timers that both fire every second. They both had 0.5 seconds left when the domain was descheduled. Before the first one is warped, a timestamp is acquired, and the calculations are based on it. If for the warping of the second, a new timestamp is obtained, the second timer will, after warping, be scheduled to fire slightly after the first. This inaccuracy will accumulate, to the point where the two timers will noticeably drift apart. This drifting must be prevented, because it would be an artifact introduced by the synchronization. This is the reason why all time warping is based on a common “now” timestamp, and why most of the calculations are based on few values, typically, `sync_original_start_time`, `sync_total_run` and expiry values based on synchronized time.

3.4.2.1 Paravirtualization

One of the main concepts of paravirtualization is playing on the strengths of knowing that the domain is virtualized. A paravirtualized Linux kernel for Xen therefore foregoes a complete timekeeping subsystem and rather acquires time information from the hypervisor. For system time, this means that it does not try to read values from hardware counters and calibrate them to find out how many increments constitute a nanosecond. It rather relies on Xen to deliver this information. Xen does this by periodically writing several values into a memory page shared between hypervisor and domain: a current timestamp of the form “nanoseconds since boot time” (`system_time`), the value of the time stamp counter at the same time, and a set of two scale/shift values that contain TSC calibration information. The domain, on the other hand, will be able to read those and copy them to an internal C struct. Whenever it needs to know the current system time, all it has to do is read the current TSC value, calculate the difference between it and the TSC timestamp value, use the calibration information to find out how many nanoseconds the difference constitutes, and add this value to the `system_time` timestamp. This relieves it from most of the timekeeping overhead.

As a side effect, it also makes it easy to warp the value according to requirement 2. Whenever a domain is scheduled, Xen will check whether one of the periodic updates to the time information has happened, and then copy the new values to the shared memory page. In the synchronized case, the nanosecond value is changed accordingly. If an update has taken place, `system_time` is set to `sync_original_start_time` + `sync_total_run`. This alone does not suffice, however. Since the structure also contains a TSC timestamp, the domain would be able to notice any time it has not been run since the update when it calculates its time values. The TSC timestamp itself should stay untouched, however, because otherwise the scale/shift values might have to be updated too, and recalibrating them is a comparatively time-consuming progress. Therefore, the difference between the current time and the timestamp is subtracted from the new values of `system_time` instead. If, on the other hand, no update has taken place, then the value has already been warped earlier per the above explanation, so only the additional time the domain has not run has to be subtracted. All that is to do is therefore subtract δt from `system_time`.

Periodic timers are implemented in paravirtualization by handing them over to the hypervisor. For example, the periodic Linux "tick" that generally happens at a rate of 100–1000 Hz[3] is actually handled by Xen at the request of the domain's kernel. Whenever it expires, an event is sent to the kernel by the hypervisor. All timers that belong to a certain VCPU are accessible via Xen's standard management structure `struct vcpu`. Furthermore, the standard Xen implementation already stops the timers when a domain is descheduled, and restarts them when it is rescheduled. Consequently, all that is to left do is to warp them accordingly before their reactivation by adding δt.

3.4.2.2 Hardware Virtualization

In the case of Hardware virtualization, the timekeeping architecture works very different from the paravirtualized way. First and foremost, the guest cannot take advantage of Xen's timekeeping, and rather has to establish its own. For this reason, all hardware timers are emulated in the Xen HVM code. The only exception is the time stamp counter. Since this counter is part of the CPU, both Intel-VT and AMD-V provide ways to virtualize it. This is done by allowing the virtual machine monitor to set an offset that is added to or subtracted from the actual value whenever a domain executes the `rdtsc` (read TSC) CPU instruction.

Xen's timer emulation code is peculiar compared to most other parts in that it is still under heavy development, and has several rough edges. The most obvious one is that the code is less organized in a centralized, reusable manner. For example, at this point in time, PIT, RTC, and local APIC timers share some code in the form of a "virtual platform timer", while this is not the case for the ACPI PMT and HPET. Furthermore, the virtual platform timers are driven by the TSC, which means that the timekeeping that Xen does for itself to ensure monotonic and precise time is not put to use, and that timers may show nonmonotonic or imprecise behavior in

[3]Newer, so-called "tickless" kernels do not have a periodic timer any more that ticks at a fixed rate. However, the latest paravirtualized Linux kernel available for Xen at the time of writing is 2.6.18.8, which still works in the old, standard way.

SMP systems. This means that the changes for HVM timekeeping are relatively complicated and diverse.

As mentioned above, processors that support x86 hardware virtualization already provide a way to virtualize the TSC. Xen makes use of this by setting a TSC offset for every HVM guest domain that is running. For time warping, this means that the approach is very similar to the warping of the system time for paravirtualized guests. However, the time is not measured in nanoseconds, but in TSC ticks. Therefore, if a HVM domain is synchronized, at the time it is put into synchronized state, not only `sync_original_start_time` is saved, but also, for every VCPU, a TSC timestamp named `sync_original_start_tsc`. Whenever a domain is scheduled, a TSC value is calculated from this TSC snapshot and `sync_total_run`, which is converted from nanoseconds to TSC ticks. The difference between this value and the current actual TSC value is then given as an offset to the Intel-VT or AMD-V virtualization architecture.

Similar to the paravirtualized case, Xen implements timers that are set by the guest in the emulated timer hardware as timers in its main timer queue. However, in contrast to the paravirtualized case, it does not stop all of these when the domain is descheduled. Time sources that use the virtual platform timer framework will be generally stopped, but Xen will keep note of missed ticks, i.e. expires that happened while the domain was descheduled. There are several ways that Xen will cope with these, depending on a per-domain setting that can be set by the user at domain creation time, including ignoring them or delivering them, or changing the behavior of the timer, which can lead to them not being stopped any more during descheduling. Time sources that are not part of the virtual platform timer will never be stopped. Finally, which timers are actually used and which one are idle depends on the operating system that is being virtualized, and which of the emulated timers it decides to make use of.

For proper time warping, all timers have to be stopped whenever the domain is descheduled, not only the virtual platform ones. However, there has to be a way to keep track of which timers are actually used by a domain. First of all, each and every timer that is part of the HVM timekeeping architecture was therefore fit with an additional variable: a flag named `active`. Whenever a domain is descheduled, every timer is checked. If it is active at this point in time, `active` is set to 1, and the timer is stopped. Otherwise, `active` is set to 0. This allows to recognize which timers have to be restarted when the domain is rescheduled again, so unused timers do not suddenly start running and confuse the domain when they signal their expiry.

In addition, all timers keep a time value *in synchronized time* that is updated when they expire and their handling function is called. For virtual platform timers, this value contains the time they expired, while for the other timers, it contains the time they will next fire at. (This difference stems from which time variables are available at which point during the execution of the various functions). In both cases, this gives the time warping the necessary information to modify the timer expiry values correctly. All that is to be done is use the time value of current real time, current time as it appears to the synchronized domain, and either the last time the timer fired and the timer period (virtual platform timers) or the next time it will fire (non-virtual-platform timers). The calculation is therefore:

$$\texttt{now} + \texttt{period} - (\texttt{sync\_total\_run} - \texttt{last\_fired})$$

or:

$$\texttt{now} + (\texttt{fire\_at} - \texttt{sync\_total\_run})$$

This warping is done when the synchronized domain is rescheduled, right before the timers are reactivated again. Warping and reactivation will be done only if the `active` flag denotes that a timer was active when the domain was descheduled, so as not to introduce rogue timers that have no use and are not expected by the domain.

A special case is the real time clock: The periodic timer that can be programmed is part of the virtual platform timers and handled by their time warping. However, the RTC emulation also contains two more timers. These emulate the time behavior of a hardware RTC. The first timer fires every second, and increases the time saved in the emulated RTC by 1 second. To correctly emulate the classic Motorola MC146818 RTC chip, it also introduces a second timer. This RTC chip was only readable at certain times; when an update of the values was in progress (once every second), reading produced undefined values. Therefore, an additional flag called “update in progress” was used to signal when it was safe to read correct values from the RTC. This safe window was 244 microseconds long. The second timer in the emulation is used to correctly emulate the “update in progress” flag behavior.

For normal (unsynchronized) domains, these timers are never stopped. The RTC always reflects the real time this way. For synchronization, this has to be changed; both timers are warped according to the above system for non-virtual-platform timers. In addition, the update to the RTC time values is not left to the timer that fires each second. To make extra sure that no time drifting can occur from cumulated precision errors stemming from the constant stopping, warping and restarting of the timers, the time value that the RTC signals is synchronized against the value of `sync_total_run`. This way, it will always produce time information that is exact, even over long periods of time, without any fear of it drifting off.

3.4.3 Miscellaneous Changes to Xen and the dom0 Kernel

Communication between the synchronization client and Xen relies on hypercalls and interrupt handlers, as has been described in Section 3.2.1. The client uses two different hypercalls to either notify the start or end of synchronization of a domain, or to relay run permission information. These changes to the hypercalls have already been described in detail in that Section, so it will be skipped here. For the other direction of communication, Xen uses an interrupt to notify the client of domains that have finished their assigned run time, and reached the time barrier.

This interrupt is one of Xen’s so-called virtual interrupts. From the hypervisor’s angle, a virtual interrupt is similar to a signal that is sent to a specific domain. To the domain, it will appear as a standard hardware interrupt. For this work, a new virtual IRQ with the number 13 and the symbolic name `VIRQ_SYNC_DONE` was defined in Xen. Signaling the synchronization client from Xen is therefore as simple as registering an interrupt handler in dom0, and raising the interrupt in Xen whenever signaling is necessary.

Finally, a few more cosmetic changes were applied to the paravirtualized Linux kernel that is used for the privileged domain dom0 and paravirtualized unprivileged domains. Their role was to make development easier and provide a cleaner interface for possible future development. First, since the domctl hypercall was not called from kernel context before our changes, there was no function to encapsulate the more generic call syntax. This is in contrast to other hypercalls that were already used in the kernel, and that received such a cleaner way of calling to the hypervisor. Consequently, a function `HYPERVISOR_domctl()` was added to the interface to abstract from the more generic `_hypercall1()`. Second, a C preprocessor definition was added to give the name `VIRQ_SYNC_DONE` to the interrupt no. 13 in the dom0 too, so that it is clear in the code of the interrupt handler which interrupt it has been registered to. Furthermore this abstraction ensures that if at some point in the future, the interrupt number has to be changed for compatibility reasons, it has to be changed only at one central location.

4 Evaluation

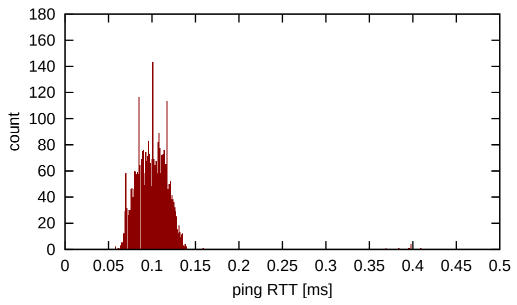
To analyze the proper functioning of the implemented synchronization system, several tests have been performed. The evaluation analyzes the following fields and tries to answer the following questions:

- Timing accuracy: Do results acquired under synchronized network emulation closely resemble real-world results?
- Overhead: How does synchronization to small time slices influence the overall run time of a synchronized network emulation setup?
- Performance: What are the effects of synchronization on performance, as witnessed by the synchronized domain itself?

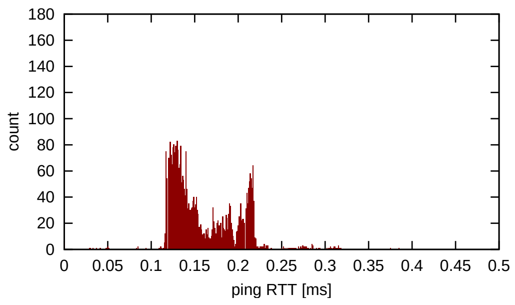
The evaluation setup consisted of two computers. Computer A was an AMD Athlon 64 3800+ with a speed of 2.4 GHz, with 512 KB cache, 1024 MB RAM, and a 1 Gbit network adapter. It was used in all of the tests, single-machine as well as multi-machine. Computer B was an Intel Pentium 4 (with hyperthreading) at 3 GHz, with 2048 KB cache, 1024 MB RAM, and a 1 GBit network adapter. It was only used in multi-machine tests.

4.1 Timing Accuracy

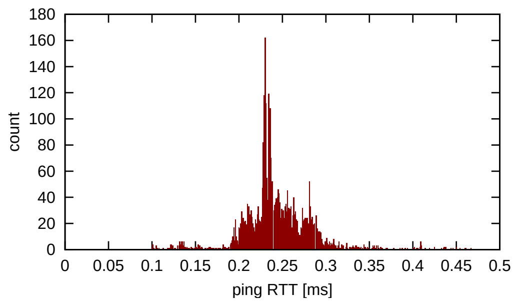
To analyze timing accuracy, an evaluation method had to be devised to measure the timing behavior at very small scales. An easy way to acquire such numbers is the standard `ping` utility. It sends an ICMP echo request to another machine, which will immediately be answered by an echo reply. The `ping` program then measures the round-trip time (RTT), which is the time from the sending of the request to the receiving of the echo reply. The standard Linux ping utility will provide numbers down to microsecond resolution if the overall RTT is less than 1 millisecond.



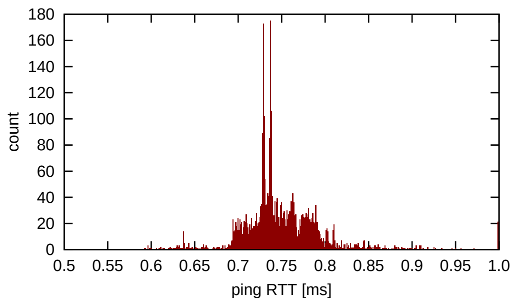
(a) RTTs for two Linux PCs



(b) RTTs between a paravirtualized Linux domain and an OMNeT++ simulation



(c) RTTs between a hardware virtualized Linux domain and an OMNeT++ simulation

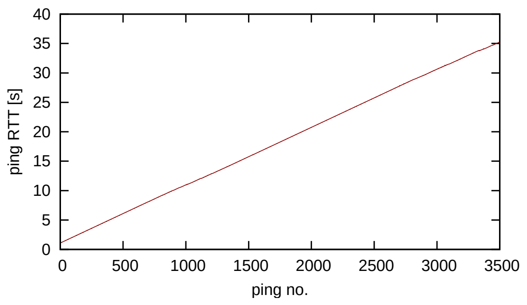


(d) RTTs between a hardware virtualized Linux domain and an OMNeT++ simulation, with delay

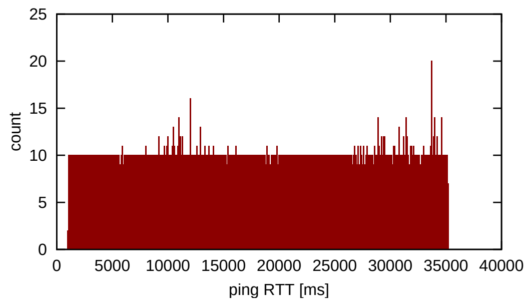
Figure 4.1 ICMP echo (ping) round-trip times under different circumstances. Histograms of 3500 echo requests/replies each.

The results of the accuracy test are shown in Figure 4.1. To produce comparison data, both computer A and computer B were booted into an Ubuntu Linux. Then, A pinged B 3500 times. For this and all other test setups, both computers were connected directly with an Ethernet cable and no other network hardware in between, such as hubs or switches. Figure 4.1(a) shows that the RTTs are for the most part between 50 μs and 150 μs, but there are some higher results, especially around 400 μs.

In contrast, Figure 4.2 shows the results of a real system connected to an OMNeT++ simulation via the emulator, but without synchronization between the two. The simulation was overloaded to show a scenario that this work tries to remedy. If the simulation had been able to cope with the incoming packets from the real system, the results would have been similar to Figure 4.1(a). However, in the overloaded case, results such as the one shown are inevitable: If the simulation is consistently slower than real time it will fall behind the real system's time base more and more, which in turn steadily increases RTTs. In this case, the ping output showed every ping taking roughly 10ms longer than the preceding one, which leads to the graph in Figure 4.2(a) if the pings are plotted on the x-axis, and a histogram of the form shown in Figure 4.2(b), for comparison with the Figures from 4.1. The results are obviously useless for any further analysis that even remotely relies on timing information. In this case, if there had been any TCP connection, for example, the real system would have eventually witnessed packet timeouts because the simulation took longer and longer to produce replies.



(a) With an overloaded simulation, RTTs are constantly increasing



(b) The same results plotted as an histogram for comparison with Figure 4.1

Figure 4.2 ICMP echo (ping) round-trip times against an overloaded simulation.

Figures 4.1(b) and 4.1(c) show the results that were received using the implemented synchronizer. In both cases, computer A was running the modified Xen and a domain that was synchronized to an OMNeT++ simulation which was running on computer B. The slice size was set to 100 μs. First, a paravirtualized Linux was run as synchronized domain. The results in Figure 4.1(b) show that almost all of the RTTs fall in an interval between 120 μs and 220 μs. Note that this corresponds with the chosen slice size. Again, there are some higher results. Note that there is a small "tail" that extends for about another time slice up to 320 μs, and very few higher results (again around 400 μs). What might be surprising at first is that there also are results that are lower than 120 μs, down to RTT numbers that are actually lower than in the real-world case. This might be explained by packets that are sent out by the synchronized domain just before its assigned time slice ended. The packet then travels over the Ethernet wire while the domain is waiting for the next time slice. This means that this time is not accounted for in the RTT calculations. Likewise, the aforementioned "tail" might be explained by packets sent out at the very beginning of a slice. It then can take three time slices for the traveling alone: The packet is sent out by the synchronized operating system during the first time slice. It is then received, processed and the reply sent out by the simulation during the second time slice, and not received by the originating domain until the third time slice.

The results acquired from synchronizing a hardware virtualized Linux domain show a similar cumulation of results in the interval between 200 μs and 300 μs, if not as massively as in the paravirtualized case. There are some faster RTTs that are bounded by the size of another slice, i.e. between 100 μs and 200 μs, and again a tail that here is slightly longer than a single slice. The slightly inferior performance probably stems from the fact that a hardware virtualized system has to rely on fully emulated interfaces, which means that there is an additional overhead in sending and receiving.

Overall, the analysis shows that results are very comparable between a domain synchronized to an overloaded simulation and two normal operating systems. There is a slight drop in performance, as shown by the shifted distribution of RTT, but comparability is still assured, especially compared to the meaningless results acquired without synchronization.

Finally, there are a few things to keep in mind while interpreting these results. First of all, the conservative time window synchronization does not give any timing guarantees inside its slices. Second, the difference in RTT results between original and synchronized system is constant. The analysis setup was chosen in a way to clearly show the difference between the three cases. While for RTTs as low as in the example, the difference might seem huge (the average RTT more than doubles between Figures 4.1(a) and 4.1(c)), for more typical ping times in the millisecond range, 100 to 200 μs difference is much less obvious, from a relative point of view. Figure 4.1(d) shows RTTs that were acquired by introducing a 250 μs delay on the line that connected the nodes inside the simulation (the simulated node that was pinged, and the `cExtHost` node that represents that synchronized Xen domain inside the simulation). Since it had to be passed twice (once for the ICMP echo request, and once for the reply), the delay added up to 500 μs. The results are exactly as expected, with most RTTs between 700 and 800 μs, exactly the added 500 μs longer than in the original test whose results are depicted in Figure 4.1(c). Third, note the distinct peaks and pits in the first three Figures. They might be artifacts stemming from the fact that while the `ping` utility gives results to a resolution of 1 μs, the accuracy might not be as high. This is a *caveat emptor* sign: It seems that the results are close to or even slightly beyond of what granularity is feasibly achievable with this test setup. So while the overall trend leads to the conclusion that the synchronization is working correctly, the results should not be over-interpreted especially in respect to cumulation and distribution in the sub-slice range because it may easily lead to a mere reading of tea leaves.

This analysis covered the timing accuracy on a micro-scale level. It is also interesting know whether there is any inaccuracy on a macro-scale level, i.e. whether there is any drift between the synchronized domain's clock and the time assigned by the synchronizer. To test this, a paravirtualized Linux, a hardware virtualized Linux, and a hardware virtualized Windows were run for extended periods of time (at least 5000 second each), at time slices of 100 μs. There was no measurable drift even after this amount of time. Since the tests were done by reading the current time of day, which the operating systems provided at a resolution of 1s, it can therefore be concluded that if there is any drift at all, it has to be below 0.02%.

4.2 Overhead

It is also interesting to know how the synchronization influences the performance of the synchronized network emulation setup. That is, how long it takes for the synchronized domain to finish an assigned slice. To test this, computer A was booted into the modified Xen, and a domain was started. Then, the synchronization server was also started on this machine, and the synchronization client synchronized the domain with only itself, communicating with the synchronization server over the local network loopback device. The domain was then run for 600 seconds (of synchronized time), and it was checked how long this synchronization took in real time. The ratio between the two times is the *synchronization overhead*, which itself can be divided into *scheduling overhead* and *messaging overhead*. The former is the overhead introduced by the synchronization client and the scheduling instructions relayed to the Xen scheduler, and the context switches that are associated with it.

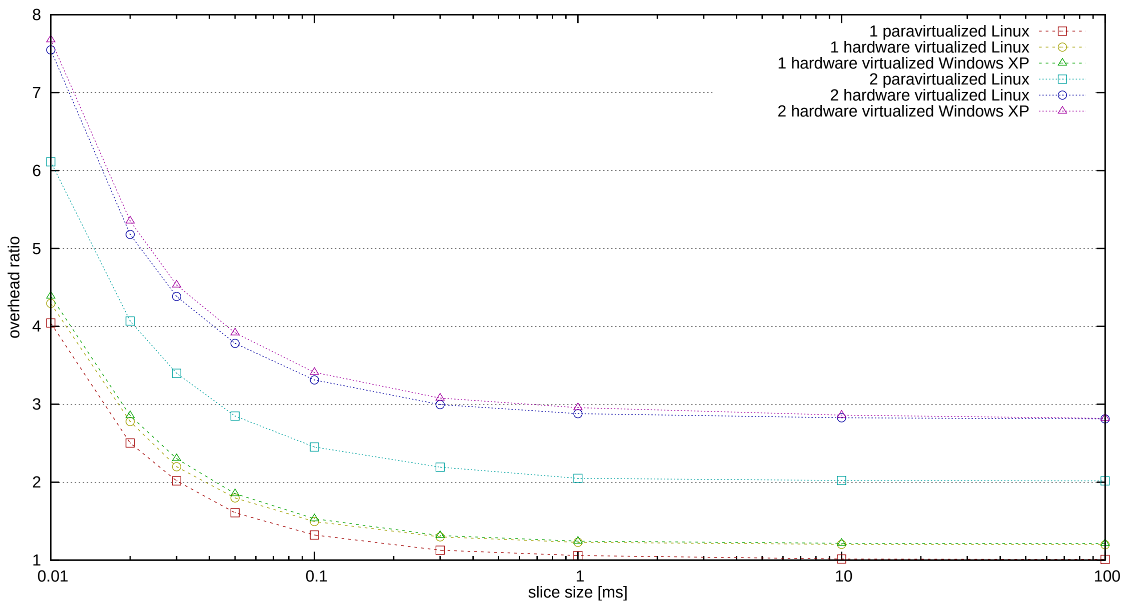


Figure 4.3 Overhead introduced by synchronizing a Xen domain, measured in the overhead ratio of actual wall-clock time and synchronized time.

The latter includes the work that is done by the synchronization server, and its context switches.

The synchronization overhead is obviously dependent on the slice size because the number of issued slices increases with decreasing slice size. This leads to more communication between synchronization client and server, more scheduling instructions, and more context switches. The test therefore was run several times, with different slice sizes each time.

Figure 4.3 shows the overhead measured during analysis. It can be seen that for slice sizes down to 1ms, the overhead is almost constant, at as little as 25% (HVM Windows) to less than 6% (paravirtualized Linux). The overhead rises from here, to about 30-50% at 100μs (the slice size that was used to produce the ping results in the previous section). From there, it rises sharply, to more than 400% overhead at slice sizes as small as 10μs.

The analysis was also done with two synchronized domains running on the same computer at the same time, synchronized to each other. Here, the overhead ratio can never fall below the factor 2, because the domains both have to run their assigned time slice, which already means that at least double the amount of real time has passed. The results lead to three interesting observations: First, for large slices, overhead doubles almost exactly for paravirtualized domains, while it increases to more than double the amount for HVM domains. Second, for small time slices, the paravirtualized domains "catch up", i.e. their over ratio rises slightly more sharply (this also is true, although to a smaller extent, of the test cases with one domain only). This result is somewhat puzzling and might need some more analysis, especially in light of the fact that several domains synchronized to the same server are scheduled strictly sequentially, the second only being started after the first is finished. Third, at small slice sizes, the overhead less than doubles compared to running only one domain. This is because the synchronization client collates several domains and only notifies the synchronization server when all domains have consumed their assigned time slice. This means that the messaging overhead is constant, regardless of the number of synchronized domains on a single computer.

The analysis shows that the Xen implementation was done efficiently, creating only minimal overhead at slice sizes of 1 ms and above (an accuracy already sufficient for many applications), and still less than doubling the execution time for slices down to 50 μs. It also shows that it is possible to synchronize to even smaller slices, if one is willing to put up with the increasing overhead. Also, implementation was done in a way that allows to run several domains on the same computer, and increasing the overhead only roughly linearly at the same time (depending on the slice size).[1]

4.3 CPU Performance

Synchronization can have negative results on the performance of the scheduled domain. This is different from overhead in that overhead measures the amount of real time in relation to the assigned synchronized time. It does not make any statements about the inside behavior of the synchronized domain, it is only a gauge of outside performance. And while the first section showed that the domain's timekeeping is in fact fooled into only perceiving synchronized time, it is also important to analyze how synchronization affects performance inside the domain. Synchronization is done by frequent de- and rescheduling. At smaller slice sizes, this happens much more often than under normal circumstances. This, however, means that the CPU has to change what it is working on more often than normal. This leads to suboptimal usage of caches. Also, pipelines cannot be used to their fullest, especially since branch prediction is useless.

To test this effect, the author used OMNeT++. This tool, while typically used as a simulator, as in this work, can also be used to benchmark a CPU. In fact, because of the large amount of integer operations, OMNeT++ is part of the SPEC CPU 2006 benchmark suite. For these tests, the simulator was compiled and linked with all visualization turned off; the same was done for the INET framework. Then, one of the INET framework example setups (named "very large LAN") was started, run for 60 seconds of simulation time, and every 1 million events, OMNeT++ was instructed to output the number of processed simulation events per second for these 1 million events. The test was repeated another nine times to get more reliable results, since they tended to vary by several percent from run to run. As a comparison, this test was done on computer A running Ubuntu Linux. Then, it was executed several times on computer A running the modified Xen, inside a synchronized domain. The test was done for different slice sizes, and paravirtualized as well as HVM domains.

Figure 4.4 shows the performance in events per second for different slice sizes. The black horizontal line denotes the performance that was reached on the same computer when running an Ubuntu Linux. The reader will note that the performance numbers actually increase for large slice sizes, compared to the unsynchronized case. The reason for this is that synchronization masks descheduling from the domain.

[1]Preliminary last-minute tests with three domains running on one machine reinforce the trends described here, on all accounts.

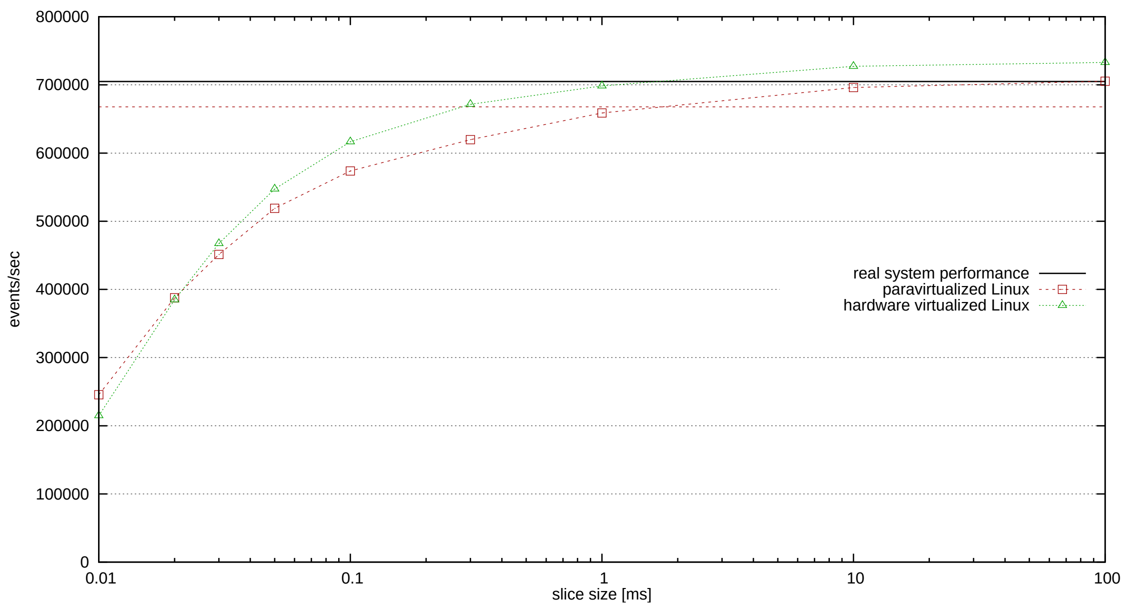


Figure 4.4 CPU performance decreases in relation to slice size. Very frequent de- and rescheduling leads to degrading results in this CPU integer benchmark.

Time that has been spent on other duties (such as interrupt handling, which is done in Xen) is not taken into account when calculating the numbers. This slight advantage is outweighed by the performance decrease introduced by the constant scheduling at around 1 ms slice size. The performance then decreases steadily, although the numbers are still at more than 50% at 20 μs. Interestingly, hardware virtualization fares better for the most part. This might be due to the fact that hardware virtualization can employ some techniques to improve virtualization performance, while paravirtualization is not able to do so. For comparison, the dashed line denotes the performance of an unsynchronized paravirtualized domain Linux domain. It shows that the performance increase for large slices is almost identical in comparison to that of hardware virtualized domains.

However, it should be kept in mind that these numbers reflect only one specific benchmark; they only measure the amount of integer operations that can be done per second. Since network-related tasks, such as traversing packets through the network stack do not purely rely on computing power, the results probably over-emphasize the performance reduction in more realistic scenarios. However, if the investigation for which this implementation for synchronized network emulation is used contains stress tests that lead to high CPU usage, this limitation might influence results. In this case, it is better to choose larger time slices that affect CPU performance much less.

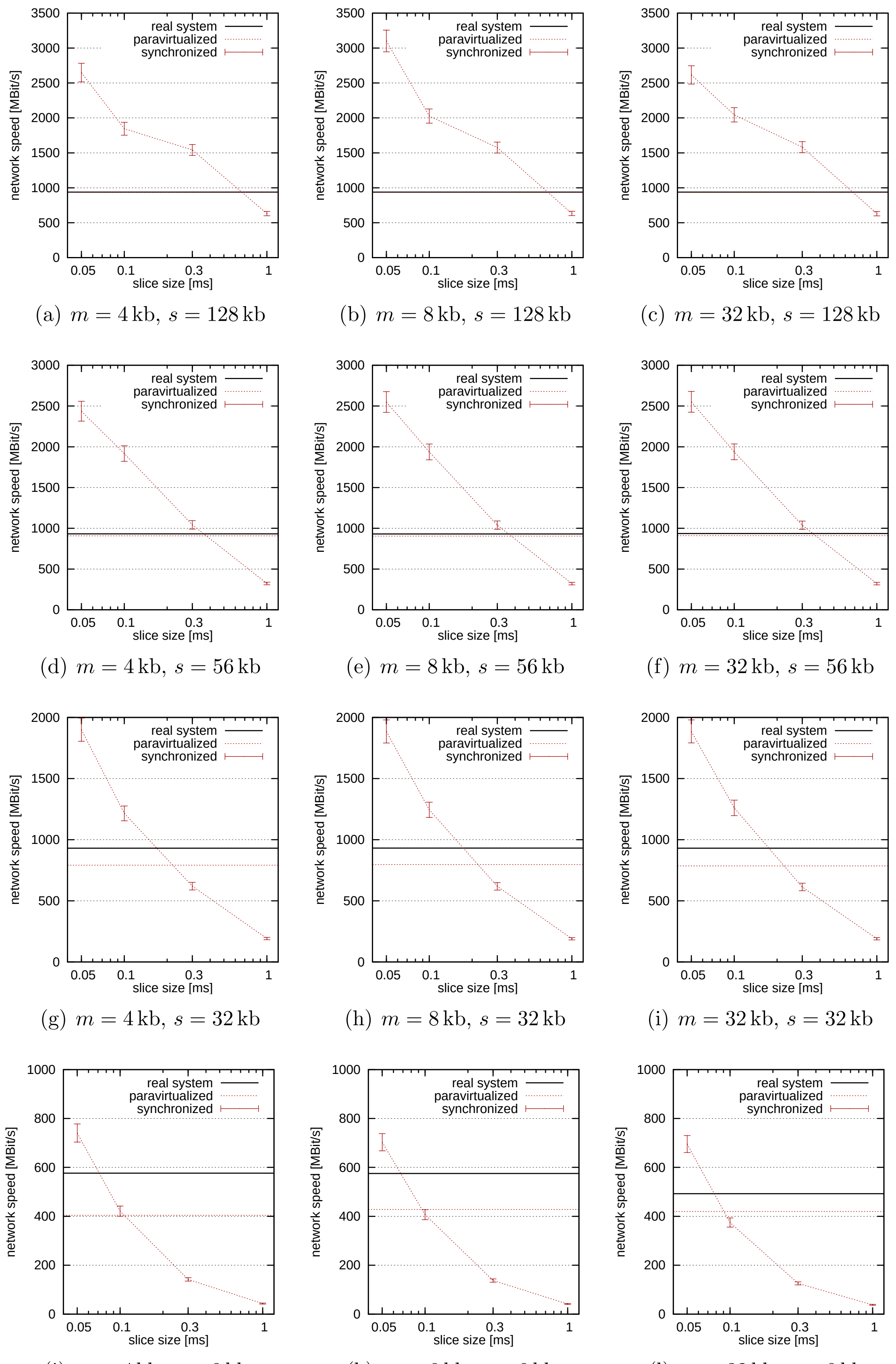
4.4 Network Throughput Performance

While it has been shown that packet travel times are accurately modeled in Section 4.1, another aspect is how virtualization and synchronization affect the throughput performance of the network device as witnessed by the synchronized domain. The graphs in Figures 4.5 and 4.6 show results created with the netperf network performance benchmark [42]. Netperf consists of two pieces of software: The netperf software itself is run on the computer that is benchmarked, while the netserver software is run on the machine that forms the endpoint of the network connection that is being tested. In particular, the `TCP_STREAM` test was used to measure bulk TCP data transfer rates. Computer A ran the netperf software, once on a Linux without Xen for comparison, and then on a paravirtualized and a hardware virtualized Linux, respectively. The tests were done at several synchronization intervals, and unsynchronized for comparison. Computer B was running the netserver endpoint software for all tests. Both computers were connected directly to each other, with no network switch or any other networking hardware in between, in the same way it was done for the timing accuracy tests.

Each netperf test at each level of synchronization, and also in the unsynchronized cases, consisted of 12 different tests, combined from three different message sizes and four different buffer sizes. The message sizes determine the size of the payload that is being sent over the network with each packet, while the socket buffer size has a direct influence on the TCP window size. While the latter would typically be chosen by the network stack and adapted to the connection speed and latency, they are set to static values in these tests to analyze the network behavior in certain situations. While this makes the tests somewhat synthetic, the choice of four different buffer sizes can give insight into typical as well as more extreme and rare situations. The combination of these two values can already have a poignant influence on the benchmark results in the standard case on an unvirtualized Linux. Each of the 24 graphs in Figures 4.5 and 4.6 depict a combination of message size, socket buffer size, and virtualization type (para- or hardware virtualization).

The graphs are read in the following way: Each graph has the performance of the unvirtualized Linux system in the corresponding test denoted by a black horizontal line, for comparison. For the same reason, a second, dashed horizontal line denotes the performance of the virtualized system without any synchronization. For each virtualization technique, each test was run at four different time slice sizes (1 ms, 300 μs, 100 μs and 50 μs for paravirtualization, and 1 ms, 100 μs, 50 μs and 20 μs for hardware virtualization). Those four test points are shown in the graph with their 95% confidence intervals, and are connected by a dashed line.

The results are very different between para- and hardware virtualization, which stems from the driver models used by each technique. Paravirtualization employs the earlier mentioned “split driver” model. Here, the data that are to be sent over the network device by the virtualized domain are simply written to a shared memory area, from which the privileged domain dom0, which controls the network adapter, reads the data and actually sends them. Obviously, writing to memory is much faster than interfacing with the network adapter, and memory is always faster by a large margin than the network connection. Therefore, an effect that at first seems very strange can be noticed in the netperf results for paravirtualization: with decreasing size of the time slices that are assigned, network throughput performance increases (or rather, seems to increase from the viewpoint of the synchronized domain), to the point where it not only beats the performance of an unvirtualized Linux by a large margin, but also transcends the limits of the physical line. The network throughput can seemingly go to several Gbit/sec, which is obviously not realistic. The reasons for this, and possible solutions, will be discussed in Section 6.2.



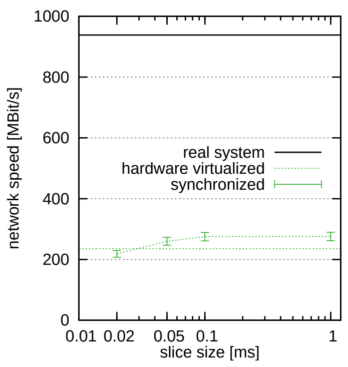
(a) $m = 4\,\text{kb},\ s = 128\,\text{kb}$ (b) $m = 8\,\text{kb},\ s = 128\,\text{kb}$ (c) $m = 32\,\text{kb},\ s = 128\,\text{kb}$

(d) $m = 4\,\text{kb},\ s = 56\,\text{kb}$ (e) $m = 8\,\text{kb},\ s = 56\,\text{kb}$ (f) $m = 32\,\text{kb},\ s = 56\,\text{kb}$

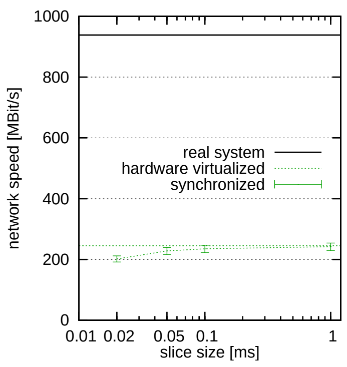
(g) $m = 4\,\text{kb},\ s = 32\,\text{kb}$ (h) $m = 8\,\text{kb},\ s = 32\,\text{kb}$ (i) $m = 32\,\text{kb},\ s = 32\,\text{kb}$

(j) $m = 4\,\text{kb},\ s = 8\,\text{kb}$ (k) $m = 8\,\text{kb},\ s = 8\,\text{kb}$ (l) $m = 32\,\text{kb},\ s = 8\,\text{kb}$

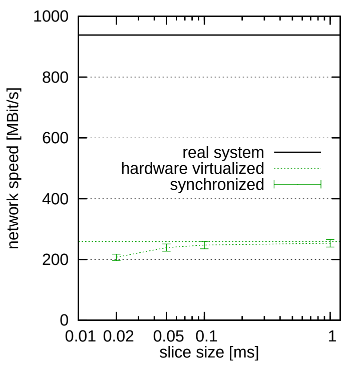
Figure 4.5 Netperf TCP throughput performance tests for different message sizes m (option `-m`) and socket buffer sizes s (options `-s` and `-S`). Paravirtualizated domain synchronized to several chosen time slices.



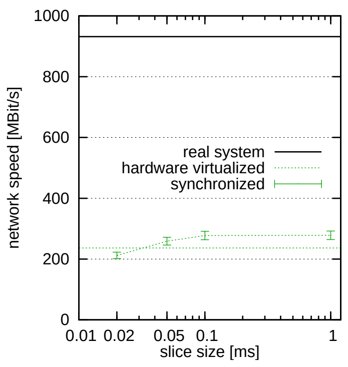
(a) $m = 4\,\text{kb}$, $s = 128\,\text{kb}$



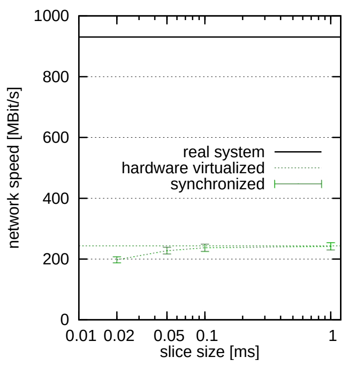
(b) $m = 8\,\text{kb}$, $s = 128\,\text{kb}$



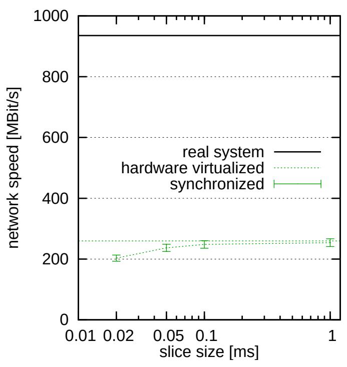
(c) $m = 32\,\text{kb}$, $s = 128\,\text{kb}$



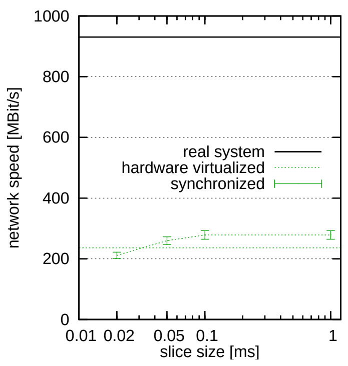
(d) $m = 4\,\text{kb}$, $s = 56\,\text{kb}$



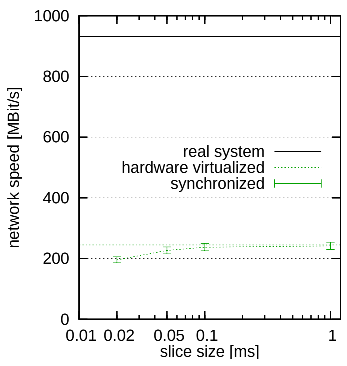
(e) $m = 8\,\text{kb}$, $s = 56\,\text{kb}$



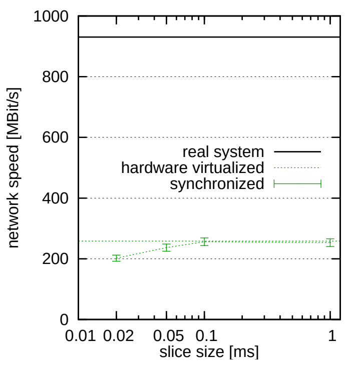
(f) $m = 32\,\text{kb}$, $s = 56\,\text{kb}$



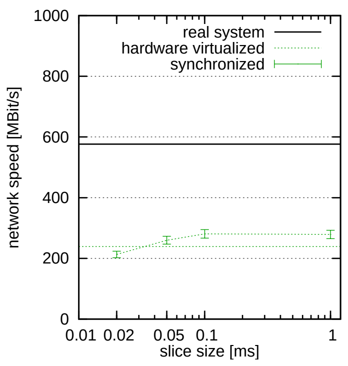
(g) $m = 4\,\text{kb}$, $s = 32\,\text{kb}$



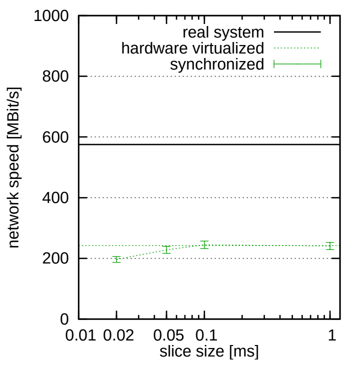
(h) $m = 8\,\text{kb}$, $s = 32\,\text{kb}$



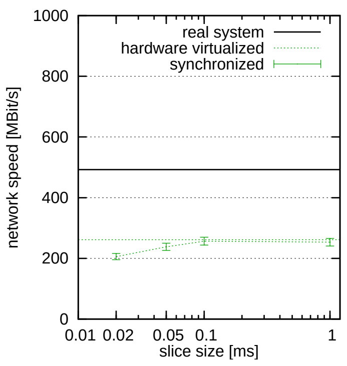
(i) $m = 32\,\text{kb}$, $s = 32\,\text{kb}$



(j) $m = 4\,\text{kb}$, $s = 8\,\text{kb}$



(k) $m = 8\,\text{kb}$, $s = 8\,\text{kb}$



(l) $m = 32\,\text{kb}$, $s = 8\,\text{kb}$

Figure 4.6 Netperf TCP throughput performance tests for different message sizes m (option `-m`) and socket buffer sizes s (options `-s` and `-S`). Hardware virtualized domain synchronized to several chosen time slices.

On the other hand, the results for hardware virtualization are somewhat impaired by the fact that a virtual network device has to be fully emulated and provided to the hardware virtualized domain. This full emulation of a hardware device in all its details leads to results that are about 75% lower than without virtualization.[2] However, the influence that synchronization has on the results is much more in line with what one would expect: a slight decrease in performance with decreasing size of time slices, similar to what is witnessed for CPU performance. At this point, further analysis on several different computers to check whether CPU speed or other factors increase or don't increase the network throughput for hardware virtualized domains compared to the unvirtualized case would provide valuable insight.

4.5 Summary

The analysis presented here shows that the modifications to the Xen hypervisor have been successful in creating an environment that allows synchronized execution of operating systems. It also shows that there is a certain trade-off between two different types of accuracy: The smaller the size of the CTW synchronization time slices the higher the timing accuracy, but the lower the CPU performance accuracy. A user that wants to employ the synchronization system developed for this thesis will want to run her tests at different slice sizes and compare results, or decide beforehand about the necessary timing or CPU performance accuracy and decide on a certain size, accepting potential inaccuracies in the other area.

From the numbers presented in this chapter, it seems as if a slice size of 1ms might be a "sweet spot" that will be useful in many cases. The timing accuracy is high enough to produce meaningful information for most evaluation scenarios, because few network protocols will rely on accuracy in the sub-millisecond range. On the other hand, the CPU performance (at least the integer crunching one) is almost identical (HVM) or at least still very close (paravirtualization) to that on a real operating system. And finally, the overhead introduced by synchronized execution is still low. This will ensure that generally, the Xen implementation will not be the bottleneck compared to a complex simulation. Depending on the simulation complexity, it will be even be possible to run several synchronized domains on one computer and still not bottleneck the synchronization.

Finally, network performance is still somewhat of a problem. In the hardware virtualized case, it behaves as one would expect it to, but performance is somewhat low because of the necessary full emulation. In the paravirtualized case, the split driver model creates unrealistic results. Solutions to this problem will be discussed in Section 6.2.

[2]Interestingly, note how the unvirtualized system reaches its limits in throughput for small socket buffer sizes (8 kilobyte in the test cases). The hardware virtualized case can catch up here and reach almost 50% of the unvirtualized performance.

5 Related Work

This chapter will give an overview over other publications in the field of work of this thesis. It should be noted that there is little to no work that tries to exactly achieve what has been done in this thesis. Therefore, work related to several aspects of this thesis are discussed one after another.

5.1 Discrete Event Simulation

The simulator used for this work is OMNeT++, in connection with the INET framework. As laid out in Section 2.1.1, OMNeT++ [69] is a modular discrete event scheduler. Its simulation modules are written in C++, and can be combined to form larger compound modules, which are written in a simple script language named NED (network description). Nearly every part of the simulation model is easily replaceable by a custom class that serves the same purpose and has the same interface. This is done by creating subclasses that inherit the interface from generic classes that are delivered with OMNeT++. A good example of how expansion of OMNeT++ works is given by the plethora of simulation models that have been developed by other people for the OMNeT++ simulation system. One of those is the INET framework [38], which has been used for this work and facilitates the use of the common internet protocols (IP, TCP, UDP, ICMP, etc.), or the mobility framework, which allows modeling wireless communications of node that move in and out of communication range of each other.

However, there is a nearly uncountable number of network simulators in existence, even if only discrete event simulators are taken into account, including, but not limited to ns-2 [28], parsec [8, 75], SFF [21], and JiST [10]. ns-2 was also mentioned before. In contrast to OMNeT++, ns-2 is more monolithic, as it brings all standard internet protocols with it in the standard installation. As the name says ("ns" stands for "network simulator"), it was also specifically designed for network simulation, while OMNeT++, although it is also primarily used in this area, supports additional simulation models due to its large amount of modularity.

One of the developments in the simulator community is to create simulation systems that are programmed by the use of standard programming languages, instead of custom simulator languages developed for the system. The idea is that this will increase the acceptance of new simulators in the simulation community, since users will not have to learn a new language to use the simulator. On the other hand, those implementations have to show that they can compete in performance with simulation systems that employ custom programming languages to drive them.

OMNeT++ goes halfways along this way: It lets the user create the basic building blocks, the simple modules, in the form of C++ classes, but description of compound modules and the overall network is done in a special, albeit very simple, description language. ns-2 uses C++, and network descriptions are done in Tcl which is not a language specific to ns-2, although it is not very widely used any more today and has mostly been replaced by more powerful scripting languages. However, the simpleness makes learning Tcl not a very time-consuming process. SFF is less a simulator than a very generic simulation framework that has to be expanded to fulfill specific roles; its minimalistic interface of a grand total of five classes and a few dozen methods is available with C++ and Java bindings. Simulators such as Parsec, which was designed with parallel DES in mind, bring with them their own specific programming language, in the case of Parsec the homonymous programing language. At the opposite end, simulation environments such as JiST go through complicated steps during the compilation phase to allow using standard Java with few limitations to write simulations models.

JiST is interesting in another aspect: The fact that it uses Java means that the code is executed in the Java virtual machine. The idea was therefore to modify the virtual machine in a way that lets the applications inside run in virtual time instead of expose them to the real time of the outside world. In that way, there is an obvious similarity to our work, especially since the limitations placed on the Java code are rather small and often facilitate the use of already existing programs as simulation models with little change. However, there are certain differences that make JiST more limited in use. First, the fact that the changes were made to the Java virtual machine means that it only works with Java programs. This not only rules out prototype implementations that were written in other languages, but it also means that it is not possible to analyze changes to the operating system kernel. Second, while our solution keeps the timing behavior of the real system intact (for the most part), JiST decided to forego a time model that closely resembles the original one. Instead, it mimics the behavior of more traditional simulators by enforcing that no instruction takes any time to execute except for `sleep`. Therefore, the code execution can be modeled by a sequence of separate events that all execute immediately, therefore creating a discrete event model. So while we brought the arbitrary starting and stopping ability of classical simulations to the prototype implementation, JiST removed the normal timing behavior of an application in lieu of the discrete event time concept of simulations.

Along a similar line, Kunz [46] presented a system that allows to plug real-world C code (in this case network stack code) into OMNeT++ for debugging and simulation. The C code is wrapped into C++ and converted so that for example, retransmission timers are modeled as self messages. As is the case with JiST, code has to fulfill a few rules so as to be convertible, and each module that is created from a predefined block of C code will run without any time consumption when it is called.

Discrete event simulation systems can also be categorized by the level of detail they work on—that is, what constitutes an event. Most of the previously mentioned simulators are packet-based, so events are closely related to sending and receiving of packets between network nodes, and their encapsulation and decapsulation in a network stack. SSF, however, is generic enough not to specify what level of abstraction its simulations work on. Also JiST, as a tool which works closely on almost-stock source code, and is not primarily concerned with network simulation, is not packet-based, but method-based: Each method forms an event that is executed at a certain point in virtual time, without this time progressing from instruction to instruction; events are scheduled for a future point in simulated time by calling the `sleep` function, followed by a method call.

Since packet-based simulation models networks on a very low-level basis, and events have to be scheduled whenever a packet is sent, received, or processed in any way, this way of simulating networks naturally becomes computationally expensive with the growth of the network, its complexity, and the rate at which data are exchanged between the nodes. A way to speed up simulation is therefore to abstract from packet-level. One approach is to model a cluster of packets that is exchanged between two nodes into a single so-called "packet train", which reduces the number of events, as presented in [2]. Another, more radical approach is fluid-based simulation [6], in which the simulation model fully abstracts from single packets, and rather models the rate of packets that are exchanged as a fluid flow with a given flow rate. Events are then based on the change of the rate, and this way a single event that defines a flow can replace thousands of events in a packet-based simulation, while still maintaining the ability to measure many performance characteristics [54, 62]. However, fluid-based simulation is not without its own problems: An issue named the "ripple effect" [48] can massively increase the number of events, to the point where performance can drop below that of a packet-based simulator. While there have been solutions proposed [73], they introduce even more abstraction inaccuracies.

Moreover, all of these abstraction approaches are not useful in our field of interest. For a simulation to be able to communicate with a prototype implementation, it is necessary that it can send replies to single packets that have been inserted into the simulation. If it abstracts from single packets, it will not be able to communicate properly with the prototype any more.

5.2 Opaque Network Simulation

Another set of tools has been developed to model networking behavior with a much higher level of abstraction than any typical discrete event simulator. These tools do not simulate any network nodes; rather, they only model the behavior of a link between real machines that house prototype implementations. This modeling mimics typical properties of internet connections, for example, a certain amount of link delay and limited connection speed. This approach is called "opaque network simulation" because the data packets sent through the simulator are not interpreted in any way. Each packet is seen as a black box, and the only property that can be changed is the speed at which it is released from the application after it was received.

It should be noted that often, this opaque network simulation is also called "network emulation". While it shares this term with the network emulation that was described in Section 2.3, the two are fundamentally different. In our case, network emulation means the translation of the representation of data from packets to simulator messages and vice versa, while opaque network simulation mimics (emulates) the behavior of a long-distance connection. However, in this section, "network emulator" will be used with the meaning "opaque network simulation".

The simplest network emulators just tap into the networking stack of the operating system on which they run, and introduce an additional queue that delays packets by a predefined amount of time, and possibly randomly drops some of them. Hitbox [3] and Delayline [39] belong to this class. Slightly more complex are systems that use multiple queues, and also model connections between them that are assigned certain properties, such as bandwidth. Two of those network emulators are dummynet [60] and ONE [4]. Dummynet comes in the form of a patch to the FreeBSD kernel, and taps into its network stack to intercept packets. To model network routers, it uses queues in which it saves packets and keeps them for a predefined time, with the possibility to drop a random amount of them. The link between those routers is modeled by copying packets from one inqueue to another outqueue, at a certain maximum rate. Dummynet is used by having at least two network adapters inside the computer it is running on, and connecting the prototypes (at least two) that are to be tested to one of them each. All communication therefore goes through the dummynet system as a gateway, and thus facilitates the introduction of the described effects on every packet. ONE, the Ohio Network Emulator, works almost identically; the main difference is that it runs on Solaris, and does not require changes to the operating system, since it runs in user space.

A newer generation of network emulators comes with additional capabilities. For example, NIST Net [16], developed at the National Institute of Standards and Technology, a Linux kernel-based software, also facilitates packet duplication for testing scenarios, in addition to the standard delay and packet loss settings of the earlier emulators. Also, it can model the behavior of the link using trace files, that is, live recordings of how network traffic behaved over a certain connection, and the modeling of congestion-dependent loss, instead of a fixed random percentage for each package. In addition, it makes use of higher-resolution timer hardware in the system it runs on, so it can control packet delay more exactly, which is very helpful with faster connections, such as 1 GBit Ethernet.

Emusocket [7] also allows the user to set connection properties in a more detailed way than in earlier emulator software. While it does not have the advanced timer resolution capabilities of NIST Net, it has the advantage of running in user space (which is the main reason of this drawback). However, it is designed for use with Java applications only, by inserting itself into the socket API. One advantage it has over many other emulators, such as the previously mentioned ONE, is that it is not necessary to use a dedicated computer with at least 2 network adapters to use Emusocket. Rather, all software that is to be tested can be executed on one machine, since the network emulation work is already done at socket level.

While Emusocket allows users to run all software on one machine, RAMON [35] is a rather complicated setup consisting of software that is based on NIST Net, and dedicated hardware that is driven by the emulator. RAMON was designed to do network emulation in wireless networks and model mobility without having to move the prototypes that are tested. To reach this goal, RAMON uses several hardware nodes, each of which is connected to an omni-directional antenna via an attenuator and functions as an access point to the actual prototype. A control connection between the central network emulation entity and those nodes allows to set the signal strength and signal-to-noise ratio of the connection between them and the prototype. By changing these two factors according to the control information sent by RAMON, the connection can be modeled in a way that mimics mobility of the prototype in a wireless network. While this works well, the complex setup and necessity of much more hardware than most other network emulators makes this a costly and somewhat complicated solution.

All in all, it has to be said that opaque network simulation excels in the area of speed over packet-based discrete event simulation. The fact that packets are not processed in any way beyond what is done in any network router facilitates much higher throughput through the simulation, since not every packet has to be processed through all network stack layers. Opaque network simulators generally are fast enough to service at least a 1 GBit ethernet connection. For comparison, NIST Net was used with a a 200 MHz Pentium computer, and was able to cope with a saturated 100 MBit link [16]. However, this abstraction means that it is not possible to model a full network with nodes that the prototype can connect to and communicate with; all of these opaque network simulators can simulate only a network connection, or a set of those, with specific behaviors. It is therefore generally necessary to have at least two prototypes for analysis, and if more complicated network setups are to be tested, the number may rise quickly. In contrast, simulators such as OMNeT++ can model all nodes in a network, complete with the possibility to communicate with them, and analysis of an implementation therefore generally requires only a single prototype in hardware if the solution presented in this thesis is used.

5.3 Emulation

The "classical" emulation, as defined in Section 2.3, is subdivided into two categories that depend on how the two entities that are to be combined—that is, simulation and prototype implementation—are connected. One way is to take the software that constitutes the implementation, and insert it into the simulation model. This is called environment emulation, or sometimes also software-in-the-loop (SiL) simulation. In the field of network simulation, it is often used by taking an implementation that is already known to work, and is in widespread use, and insert it into the simulation to make that simulation's behavior more realistic. This has been done for the FreeBSD network stack, which has been inserted into the OMNeT++ discrete event simulation system [14]. The "Network Simulation Cradle" [41], a wrapper that can be used with several different operating systems' networking stacks (FreeBSD, OpenBSD, Linux), facilitates insertion into the ns-2 network simulator. Both of these increase the credibility of simulation results that deal with network throughput.

This usage of environment simulation is the counterpart to what it is typically used for in many other fields, in which the prototype implementation is to be tested for correct behavior. However, if the changes in the prototype implementation are done inside an operating system's network stack, for example when developing a new protocol, it is possible to use an already existing environment emulation system such as the ones described above, and patch the changes into the network stacks used there.

An approach that is somewhat of a hybrid between environment emulation and opaque simulation is the ENTRAPID Protocol Development Environment [36]. Running in user space, it creates "virtual network kernels" (VNKs) that are based on the FreeBSD networking stack, although the authors argue that with little work, any arbitrary network stack can be used in the same way. The stack can then be modified for the addition and subsequent testing and analysis of newly developed network protocols. Every VNK can be understood as a network node, and in this way ENTRAPID can be seen as a simulation environment, with the difference that all nodes necessarily contain an environment emulated network stack, whereas in a simulator, one could run a subset of nodes with an integrated network stack, and some with the simpler (and faster) standard model. To test those network stacks, application software is necessary that sends and receives data, and those are assigned to a VNK. In this way, ENTRAPID is similar to opaque network simulation, in that no sophisticated simulation nodes exist that are simulated. Also, simulation is not event-based, but rather works with real time values. Meaningful performance evaluation is, however, somewhat impaired by the high overhead introduced by the VNKs.

The second type of emulation as defined per Section 2.3 is network emulation, or sometimes also named "hardware-in-the-loop". Kevin Fall was the first to present such a system, which was developed for the predecessor of the ns-2 simulator, ns [27]. It already contained the vital part of a network emulator: a translation unit that transformed network packets into simulator messages, and vice versa. Despite its age, it has (in a more recent, updated version) still been in use recently for network emulation research, such as in [33]. Network emulation has also been used with other network simulators, such as the commercial OPNET [17] simulation system. [13] used a two-stage process, in which the communication between real systems and their network packets, and OPNET and its message format is facilitated by translating both into a standardized message format that is part of the HLA (High Level Architecture) [23], which was specifically designed for interconnection of and communication between simulation systems.

For OMNeT++, the network simulator that was used in this thesis, work has recently been done to create a full-fledged network emulation interface. [67] uses a translator between simulation messages and network packets that is very similar to the one created for this thesis. However, their solution is slightly more general due to the fact that their translator does not represent every prototype implementation with a counterpoint in the simulation model. Rather, they have the translator behave like a router to the simulation, and a gateway to the prototype implementation(s). Also communication between the prototype and the emulator is not done via tapped and UDP-tunneled packets, but rather by using the standard routing infrastructure of every network. Packet data is written using libpcap [66], which allows the gateway to send packets that bear the addresses of the simulated hosts in their headers instead of the gateway's.

Of the two types of emulation, environment emulation is unsuitable to reach the goals set for this thesis. As it inserts an implementation into a network simulator, it suffers from similar problems as JiST [10], presented in Section 5.1. The implementation loses its notion of wall-clock time, and instead uses the simulated time that is presented by the simulation system. As such, while it is possible to analyze the correct functionality, meaningful performance analysis cannot be done any more. Network emulation, on the other hand, will also be unable to produce meaningful timing information in scenarios with complex simulations unless it employs synchronization. None of the solutions so far have done so, and so rely on the simulation always being able to run in real time. While all network emulation systems presented here can cope with simulations that run faster than real time, none will produce useful results when the simulation cannot keep up with the prototype implementations that are connected to it. To the knowledge of the author, this thesis is the first work that has tried to relieve simulations of this constraint.

5.4 Encapsulation of Real Systems

While virtualization and CPU emulation are both techniques that have been used for a long time, and implemented for a wide range of hardware, presented for example in [9, 12, 20, 37, 44, 50, 72], the idea to use the encapsulation that is attained this way for network emulation, or at least some sort of network performance analysis, is rather new, and has not found widespread use yet.

Gupta et al. presented a system in [32] that facilitates network performance evaluation for network speeds that are not available yet. Similar to this work, Xen is used to encapsulate an operating system as virtualized domain, and the timekeeping subsystem is modified to achieve what the authors call “time dilation”: The illusion to the virtualized domain that time is running slower than it actually is. This, in turn, can increase network throughput, as it appears to the domain. If a domain is run with a time dilation factor of 10, i.e. time is passing 10 times slower inside the system than it does in reality, a 1 Gbit Ethernet connection may appear as a 10 Gbit one, if it is saturated, because events arriving from outside the machine are not scaled. To also scale package round-trip times accordingly, dummynet [60] was used. The main difference in how timekeeping was changed for their work and for the work presented here is that time is slowed down uniformly, whether the domain was scheduled or not, and not stopped when the domain is descheduled. Also, the original system only supported paravirtualized domains, and did not change the scheduling subsystem. A follow-up publication [31] expanded on the original work by introducing more advanced managing techniques, support for hardware virtualization, and the scaling of hard disk I/O according to the dilation factor.

Virtutech Simics [50] is a full system simulator, i.e. it emulates the behavior of a CPU, as well as all hardware devices that are connected to the simulated system. In contrast to other CPU emulators, which often are built for a high level of accuracy down to instruction or even cycle level, Simics was designed to be a fast simulator, at least in comparison to other such applications. To reach this goal, it has to compromise in some respects. For example, hardware emulation is transaction-based, i.e. data that are sent over the emulated network adapters are simulated not as single bytes sent one by one, but rather the packets form atomic units. Moreover, Simics is only instruction accurate; it does not provide cycle accurate timing information. Simics has been used for network analysis. [26] presents a system to use Simics for evaluation of network performance of embedded devices. While Simics can simulate systems with more computing power, full system simulation is arguably too slow to make this a worthwhile effort. The solution presented for the embedded devices facilitates the execution of several simulated nodes on one machine, as well as the execution on several physical machines, which are then synchronized against each other, much in a way that parallel distributed simulation works. While this work is very promising, as full system simulation allows exact synchronization down to very small time slices, less than what is feasibly achievable with the approach presented in this thesis (see in specific Section 4.3) and brings a full hardware emulation to the table, the performance is very low compared to a virtualization solution. A very simple test conducted by the author showed that a Windows XP took more than 8 times as long to boot, compared to a Xen domain (335 seconds vs. 40 seconds). Furthermore, some findings suggest that Simics' abstraction from an exact 1:1 emulation of x86 current processors, such as no emulation of out-of-order execution, can lead to inaccurate performance results compared to real processors [71], something that one would generally not expect from a full system simulator.

5.5 Extensions to Xen

The author's work on Xen is by far not the only extension that has been done in the scientific community. In fact, the freely available code, its publication under the GNU General Public License, and its modular design have created possibilities to create a plethora of extensions that cater to different needs. A few of these will be presented as examples in this section. It should be noted that the work done by Gupta et al. [31, 32] could have also been presented in this section instead of the previous one, since it also constitutes a Xen extension.

One of the more obvious and straightforward applications is to use Xen to debug operating system kernels. Generally, kernel debugging is a tedious task, since the choice of tools is limited. Separating the operating system from the hardware and the ability to stop execution allows for more convenient debugging. A first step is to use an unmodified Xen and its already existing facilities. Kamble et al. presented in [43] how to use Xen for Debugging kernels with gdb [30] and its gdbserver-xen extension, which is supported by any standard Xen without changes to the code.

Another step forward has been done with the creation of the Xenprobes framework [53]. Probing allows the user to set a breakpoint on any instruction in the (virtualized) kernel. Whenever this instruction is to be carried out, a special debug interrupt is raised, which returns control to Xen. Xenprobes supports two types of probes. One is suitable for single instructions that are to be probed, by calling a pre-handler and a post-handler function immediately before and after the instruction is executed, respectively. The other type is used for probing function calls. The interrupt is set on the function entry point, and whenever the function is called, an entry-handler is executed before control returns to the function, and an exit-handler when the function is about to return. The probes and their associated interrupts can be dynamically set at run time, and since each handler is a normal function, the choices of what to do when a breakpoint is reached are near limitless. A simple debug output via printk is just as possible as incrementing counters for profiling, or snapshots of values of certain variables.

XenAccess [55] is a tool for memory introspection (primarily; other applications, such as disk monitoring, are in development, but not as mature). It facilitates the reading of domain memory, and uses the kernel symbols to find the requested data structures. They can then be read and presented to the user. XenAccess comes with examples to, for example, read the current list of running processes on a Linux kernel, or the currently loaded kernel modules, but it can be easily extended to provide other information. Introspection is not limited to paravirtualized domains, but can also be used with hardware virtualized ones. Besides the fact that this tool facilitates the collection of statistical data about a running system, it can also be used as an intrusion-detection system (IDS), by monitoring an operating system's system call table for changes, or the integrity of kernel modules. A downside is that XenAccess needs detailed information about the kernel memory layout of a system, so it can find the entry points for linked lists and other data structures that it wants to introspect, so it has to be updated whenever this layout changes in a new version of the operating system kernel.

Another very interesting extension for Xen is PTLsim/X [74], a cycle accurate full system simulator that can drive a Xen domain. Its author asserts that it is the only such simulator that it available as open source. It simulates a x86-64 processor, complete with modeling of pipelines and out-of-order execution. A very interesting property is that PTLsim can be switched between its cycle accurate mode and native execution of instructions for higher speed through areas of code the user deems uninteresting. When a domain is created, a small amount of memory is set aside to load PTLsim into it. The simulator then works as another layer of indirection between the hypervisor and the domain. Because cycle accurate mode is much slower than native execution, livelocking due to incoming network packets poses a problem: Processing a packet will take much longer, during which several new packets might have arrived for processing, giving the illusion of an extremely fast network, similar to what was done on purpose in [31, 32]. Here, however, this behavior is not desired. The same can happen with other hardware devices. Therefore, PTLsim checkpoints the system when it is set into cycle accurate mode, runs for another small amount of time in native mode, and logs all hardware device interaction in a trace file. Then, it returns to the check point and replays the hardware events at the appropriate time during cycle accurate simulation. The threat of livelocking is very real: Like other cycle accurate simulators, PTLsim runs extremely slow in this mode. In a test presented in [74], a test that took 0.7 seconds in native execution mode ran for more than 62 minutes when done in cycle accurate mode. This means the simulation ran at less than 0.02% of real time, an overhead factor of well over 5000.

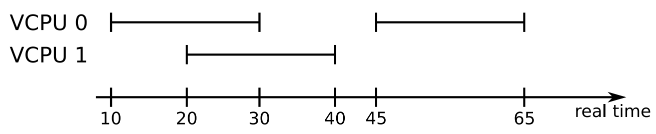
6

Future Work

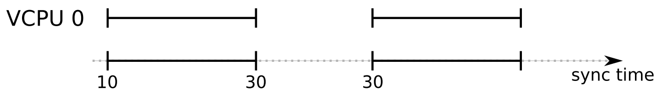
In this chapter, the author wants to present a few fields of work that might benefit from more detailed research, analysis or further development. Three topics will be discussed here. The first deals with how to expand synchronization onto virtualized domains that have been created with more than one VCPU because this case is different from the uni-virtual-processor case. The second discusses solutions to the problem of unrealistic network throughput reports in paravirtualized domains (see Section 4.4), which can skew performance evaluation results for this virtualization mode. Finally, the third topic discusses how to expand the synchronization system presented in this thesis to facilitate more detailed debugging and analysis of the prototype implementations running in synchronized domains. It will propose investigation into how this work can interact with XenAccess [55] and Xenprobes [53], and how these tools might benefit from each other and whether there might even be beneficial synergy effects improving the usefulness of each tool.

6.1 Synchronizing Virtualized SMP Systems

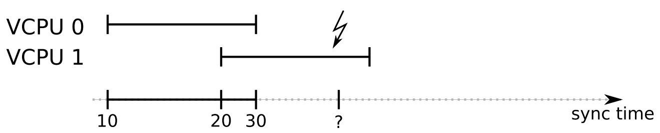
Xen allows domains to have as many virtual CPUs as the administrator desires (up to a certain hardcoded limit, but not limited by the number of physical CPUs present in the system). The changes to Xen done for this thesis have kept expandability to support for multi-VCPU domains in mind, and most of the code could actually work with such domains. However, some vital sections (especially the notification from sEDF to the main scheduling loop that a domain has finished its assigned slice, as described in Section 3.4.1.2), will need a few more changes. The reason that the author has not finished the work in this area and this work has focused on synchronizing single-VCPU domains is that synchronizing domains with more than one VCPU creates additional, more fundamental problems in respect to timekeeping. First of all, it is important to discriminate between two cases: The number of VCPUs can either be lower than or equal to the number of physical CPUs, or it can be higher.



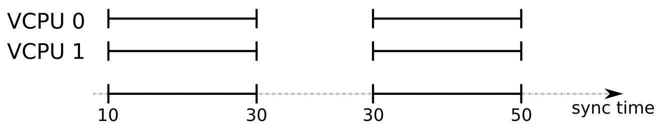
(a) SMP domain in regular mode. All domains agree on a common time base based on real time.



(b) For comparison: a one-VCPU domain in synchronized mode. Time only proceeds while the domain is scheduled.



(c) A synchronized SMP cannot run its VCPUs in the same way. What time is it at the lightning mark?



(d) A solution is to always schedule all VCPUs at the same time.

Figure 6.1 Synchronizing SMP domains creates new problems.

6.1.1 Case 1: Number of VCPUs Lower Than or Equal To Number of PCPUs

When the Xen scheduler schedules a multi-VCPU domain, it makes use of its timing information and relays it to the domain. In other words, the domain will be able to recognize that its VCPUs do not run 100% of the time. At times, none may run, or one, or several but not all, or all of them at the same time. This is not a problem because overall, the scheduler will ensure that the domain receives the time it was assigned by the administrator. When a domain is assigned a certain slice of time, it will receive an overall CPU time of $s \cdot v$, where s is the slice and v the number of VCPUs. For example, a domain with 4 VCPUs and an assignment of a slice of 10 ms every 100 ms will receive 40 ms of CPU time during those 100 ms, if possible. When this happens during the 100 ms period is up to the scheduler. As in the uniprocessor case, with a standard Xen installation, domain time correlates with real time. This is depicted in Figure 6.1(a). Time even passes when no VCPU is running.

This way of scheduling multi-VCPU domains—that is, scheduling each VCPU independent of each other—does not work for synchronized domains. There are three constraints that have to be met when scheduling a synchronized domain of several VCPUs:

1. Time progresses if and only if the domain is running.
2. Time to run is assigned by the synchronization server.

3. The total CPU time is the assigned time multiplied by the number of CPUs.

If the VCPUs are simply scheduled at some point during the deadline period, timing problems can happen, such as in Figure 6.1(c). VCPU 1 was started after VCPU 0. Time was frozen before VCPU 0 started, and then started progressing again. The VCPUs were scheduled for 30 time units (for the sake of this example it does not matter whether these are microseconds, milliseconds, or any time unit). After this time, VCPU 0 is stopped, but VCPU 1 is still running. Time-wise, the domain is now in a difficult position. What time is it at the point denoted by the lightning mark? If time continued to run, it might be around 40 time units. This, however, violates constraint 2, since the time progresses farther than the allowed 30 time units. Another approach would be to freeze time at 30 units. This, on the other hand, violates constraint 1 and can lead to problems for the programs running on VCPU 1, since time calculations will be based on a time that does not progress, and therefore invariably will be wrong. Other ideas include stopping all VCPUs when the assigned time has been used up (violates constraint 3), or giving each VCPU its own time base (which again disturbs time calculations, since time might jump back- and forwards if it is compared between different VCPUs, and so violates constraint 1).

This is a serious problem for synchronizing virtual SMP domains. So far, the author has found only one solution: All VCPUs of a domain have to be scheduled at the same time, as depicted in Figure 6.1(d). In this case, all three constraints can be held: Time progression coincides with running VCPUs, and the assigned time is accurately used up, in wall clock time as well as CPU time. However, this requires more logistic overhead than in the single-CPU case. In detail, a close cooperation of and communication between the schedulers running on the different physical CPUs is necessary. Whenever a domain is scheduled, and the scheduler is called on one of the PCPUs, it has to pick a VCPU of the synchronized domain, set a flag (atomically) that represents the VCPU, signals that it is ready to be scheduled, and is readable by all the other scheduler instances. Finally, the scheduler has to spin and check the array until all flags are set. The next scheduler that is called has to do the same with the next VCPU, and so on, until all VCPUs are ready to start running immediately. In this case, the schedulers will stop spinning, and all start scheduling the VCPUs, all simultaneously, therefore agreeing on a common timebase. This is similar to how both Linux and Xen measure drift between the TSCs of different CPU cores at system boot time on SMP systems. However, using this technique in a scheduler that is called many times per second will probably incur a very noticeable performance decrease.

6.1.2 Case 2: Number of VCPUs Greater Than Number of PCPUs

This case incurs even more problems than the first one. In this case, it is not possible to use the technique described above because there are not enough physical CPUs to schedule all VCPUs at the same time. A completely new technique would be necessary. While the author, at this point in time, cannot think of any elegant solution which would not violate one of the constraints for proper timekeeping laid out above, it is a field that might spur future work and will need more research and discussion to come to a well-founded conclusion about its feasibility. A solution to this problem would also probably produce a more efficient solution to case 1 than the proposed strict simultaneous execution.

6.2 Network Throughput Performance

The analysis presented in chapter 4 has shown that for the most part, the synchronized execution of the virtualized operating systems works very well. However, one area is somewhat problematic. The network throughput for paravirtualized domains (see Figure 4.5 on page 53) as it is witnessed by the synchronized domain is not realistic. This can be seen by the fact that a TCP throughput benchmark reports numbers as high as 3 GBit/s on a network adapter that only supports a maximum of 1 GBit/s. This is due to Xen's "split driver" model. The paravirtualized domain uses a very simple network driver front-end, which does not interact with the hardware at all, and rather puts data it wants to send out into a shared memory area. The dom0 contains the actual driver back-end that interfaces with the hardware, and will take the data out of the shared memory to send packets on behalf of the unprivileged domains.

Obviously, writing data into memory is a much faster operation than sending data to the network adapter. The synchronized domain therefore can put more data into the shared memory than it would be able to send to the network adapter, if it had direct access to it, up to the point of completely filling the shared memory. Whenever the domain is descheduled, the control domain dom0 starts to empty the shared memory buffer and send the packets over the wire. This happens while the synchronized domain is descheduled, and therefore during a time that it doesn't realize is passing at all. Since the dom0 is running much more often in between the synchronized domain for smaller time slices, this explains the seemingly increasing speed with decreasing slice size.

A solution to this is to change the back-end network driver in the privileged domain dom0 to be aware of the synchronization of domains that it sends and received packets on behalf of. Thus, a synchronized driver would send only the appropriate amount of data to the network adapter whenever a synchronized domain is descheduled. For example, if the size of the time slices was set to 100 μs, and the network adapter is a 100 MBit Ethernet device, the maximum amount of data to be sent would be 1250 bytes. Packets are sent out until this margin is reached.

Another way to tackle this problem could be to investigate PTLsim's [74] capabilities at doing a full emulation of hardware devices. Then again, the cycle accurate execution entails a crippling speed penalty which in all likelihood rules out this approach.

Finally, it should be noted that fully emulated drivers in hardware virtualized mode do not produce paradoxical results for network throughput performance, but rather behave as expected. However, network throughput is noticeably lower than on an unvirtualized system. If one found a solution to synchronize the paravirtualized back-end driver, it would be possible to eventually use special drivers in the hardware virtualized domain that take advantage of the knowledge that their operating system is virtualized. Such paravirtualized drivers for hardware virtualized domains have been in development for some time—for example, for Windows XP [34]. This way, the network throughput performance of hardware virtualized domains could be increased, and a combination of high performance and meaningful results could be reached.

6.3 Introspection and Probing

This work presents a set of tools that facilitates synchronization of an operating system with a prototype implementation to a packet-based discrete event simulator. One of the reasons for this was to lower the requirements for prototype analysis by using only one prototype together with a simulated network, instead of a testbed of many prototypes. In addition, it is much faster and easier to restructure a simulated network inside a simulation than it is to reorganize a network testbed constructed from real machines.

Another step forward in this field of both detailed and convenient analysis would be to expand the Xen implementation to make debugging potential software errors easier, while the prototype is running inside a Xen domain. Section 5.5 presented two extensions to Xen that can aid the user in debugging, namely XenAccess [55] and Xenprobes [53]. XenAccess allows introspection into the memory of a domain that is currently running, and Xenprobes places hook functions on instructions. Each of these tools on its own is already a helpful aid in analysis and debugging. Combined, and with the synchronization presented in this thesis, they can form a powerful combination for development and analysis. It would be possible to read the contents and state of the networking stack of a synchronized operating system via XenAccess whenever an assigned time slice has ended, or to probe certain functions in the code that was added.

While both tools introduce a considerable overhead and therefore slow down the execution, synchronization as presented in this thesis will ensure that the results are very close to the behavior of the uninstrumented system in regards to time behavior.[1] While the machine that Xen and the synchronized domain are executed on will take longer to execute one time slice, we have shown in Chapter 4 that time warping as presented in Section 3.4.2 accomplishes requirement 2 as set in the introduction. Therefore, time spent outside of the synchronized domain—that is, in XenAccess or Xenprobes—will not be attributed to the synchronized domain; the additional overhead that is created is only a nuisance, but it will not influence the results in any clearly noticeable way.

The next steps in this direction could be to test how well XenAccess and Xenprobes work together with the current version of Xen, and the synchronization changes. Then, XenAccess would have to be extended to read network stack status, and make it executable from either the Xen kernel or the dom0 kernel, so that it could be called automatically whenever a synchronized domain has finished running its time slice. Since XenAccess runs inside the privileged domain dom0, the timewarping

[1]In theory, the only differences will be due to more context switches between synchronized domain and Xen/dom0, and performance decreases due to cache misses and pipeline stalls. The effects of this can be seen in Figure 4.4 and page 51.

code should work with it without any changes. For Xenprobes to work properly, a small change to the debug interrupt's handler will be necessary, so as to stop the synchronized domains time while the interrupt is being serviced; this way, the interrupt handling will not be realized by the domain due to lost time.

7 Conclusion

This diploma thesis presented network emulation, that is, the connection of a real system running a prototype implementation with a network simulator, and focused on a specific problem of this field: The fact that while both have a notion of time, its representation is fundamentally different in a simulator and a real system on x86 hardware. It was shown that problems occur when these two representations of time do not agree on the current time. So far, this problem had generally been solved by using simulation systems on very fast hardware, or distributing them over several machines, to ensure that they will never be slower than real time, even in very complex simulation, and then slow them down to real time when they are faster. This ensured that simulation time would run in unison with real time, which is the basis of the clocks inside a hardware prototype.

However, the underlying problem is that it is not always possible to make a simulation fast enough to meet the goal of real time capability. For every system, no matter its number-crunching ability, there are simulations complex enough to slow it down so that this capability is lost. Therefore, this diploma thesis presented a way to slow down the real system that is connected in the network emulation scenario, and execute its operating system synchronously to the simulation, so that the real-time constraint is lifted from the simulation.

To reach this goal, the Xen Hypervisor was used to encapsulate the real system inside a virtual machine, so as to detach it from the hardware it would otherwise run on, while keeping all of the internal behavior intact, so that results from running the prototype implementation are credible. Two fields of work were identified to facilitate synchronous execution. First, Xen's scheduling subsystem was modified so that it allowed precise scheduling on demand of a virtualized OS for short amounts of time. This made it possible to have a real system wait on the simulator whenever the latter fell behind because it could not run in real time. Second, the interface between the OS and the hardware clocks that define its timing behavior was modified to mask passing of time while the OS is descheduled, so that the illusion of a continuous flow of time is upheld. This way, an operating system could be stopped for arbitrary amounts of time, without the OS ever noticing it was stopped.

A detailed insight into how these changes were implemented was given in Chapter 3. Changes to the scheduling subsystem involved modifying the Xen sEDF scheduler, but also the main scheduling loop, to make it aware of time it consumed itself during the scheduling. Changes to the timekeeping subsystem had to be done for both paravirtualized and hardware virtualized Xen domains, and involved using time information created by the scheduler and by Xen's own timekeeping facilities to stop timers that signaled the passing of time, and warp them and time counters in proper ways to keep up the illusion of a continuous flow of time to the Xen domain.

To construct a fully working setup in which synchronized network emulation could be used and its correct functionality be analyzed, a set of tools was developed and implemented besides the Xen modifications. First of all, a central synchronization server was necessary that kept a reference clock and instructed simulation and prototype to run for specified amounts of time, implementing a conservative time windows algorithm (see Section 2.4). It was also necessary to create a client that ran on the Xen machine and formed an interface between the synchronization server and the modified Xen. Finally, an emulator had to be constructed that made communication between simulation and prototype possible.

When the work for this thesis was started, it was not clear whether an approach to synchronized network emulation using the Xen Hypervisor was feasible at all. The results of the analysis presented in Chapter 4 showed that the solution presented works very well, providing a synchronization accuracy of down to 10 μs. Furthermore, at larger time slices that are nevertheless still useful for network protocol analysis in a network emulation scenario, the overhead introduced by the synchronization is very low, less than 50% at an accuracy of 100 μs, and less 25% for hardware virtualization and 6% for paravirtualization for an accuracy of 1 ms. Finally, the timing results measured are very comparable between a network emulation setup with a simulation that is not real time capable, and a setup consisting of two real systems. This means that the solution presented in this thesis can be used for detailed and meaningful analysis with simulations of arbitrary complexity, relieving network emulation of its constraint of real time capable network simulations.

The main problem with the implementation at this point in time is that, for operating systems virtualized via paravirtualization, network throughput results measured in the virtualized operating system are not credible. Suggestions on how to solve this problem were proposed in Section 6.2. The author assumes that a solution will not only be possible, but also realizable with relatively little investment of time and work.

Finally, while this work has focused on using synchronization of operating systems for synchronized network emulation, a fundamentally new field of application opens up when it is combined with tools for introspection and probing. As has been described in detail in Section 6.3, a combination of this work's synchronization with XenAccess and Xenprobes will allow to debug an operating system with convenient and powerful tools, while at the same time keeping its timing behavior intact. While generally, debugging introduces new delays, and breakpoints set for this purpose can totally change the timing behavior, synchronization will mask the time spent in debugging functions from the debugged operating system. This will allow to debug a system that behaves as if it was running without any debugging action, even in time-sensitive areas of the implementation.

Therefore, the author not only concludes that synchronization of operating systems created a system for feasible and powerful network analysis through synchronized network emulation, but he also anticipates exciting new use cases in the field of operating system debugging.

Bibliography

[1] Adair, R. J., Bayles, R. U., Comeau, L. W., and Creasy, R. J. A virtual machine system for the 360/40. Tech. rep., Cambridge Scientific Center Report 320-2007, 1966.

[2] Ahn, J. S., and Danzig, P. B. Packet network simulation: speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking (TON) 4*, 5 (1996), 743–757.

[3] Ahn, J. S., Danzig, P. B., Liu, Z., and Yan, L. Evaluation of TCP vegas: emulation and experiment. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1995), ACM, pp. 185–195.

[4] Allman, M., Caldwell, A., and Ostermann, S. ONE: The ohio network emulator. Tech. Rep. TR-19972, School of Electrical Engineering and Computer Science, Ohio University, August 1997.

[5] AMD. *AMD64 Virtualization Codenamed "Pacifica Secure" Virtual Machine Architecture Reference Manual.* AMD, May 2005.

[6] Anick, D., Mitra, D., and Sondhi, M. M. Stochastic theory of a data handling system with multiple sources. In *ICC '80: International Conference on Communications* (June 1980), pp. 13.1.1–13.1.5.

[7] Avvenuti, M., and Vecchio, A. Application-level network emulation: the emusocket toolkit. *Journal of Network and Computer Applications 29*, 4 (2006), 343–360.

[8] Bagrodia, R., Meyer, R., Takai, M., an Chen, Y., Zeng, X., Martin, J., and Song, H. Y. Parsec: A parallel simulation environment for complex systems. *Computer 31*, 10 (1998), 77–85.

[9] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM ymposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[10] Barr, R., Haas, Z. J., and van Renesse, R. JiST: an efficient approach to simulation using virtual machines. *Software—Practice and Expererience 35*, 6 (2005), 539–576.

[11] Baumgart, I., Heep, B., and Krause, S. OverSim: A flexible overlay network simulation framework. In *10th IEEE Global Internet Symposium* (May 2007), pp. 79–84.

[12] Bellard, F. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[13] Biegeleisen, E., Eason, M., Michelson, C., and Reddy, R. Network in the loop using HLA, distributed OPNET simulations, and 3D visualizations. In *MILCOM '05: Proceedings of the IEEE Military Communications Conference* (October 2005), vol. 3, pp. 1667–1671.

[14] Bless, R., and Doll, M. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNeT++. In *Proceedings of the 2004 Winter Simulation Conference* (2004), Winter Simulation Conference, pp. 1556–1561.

[15] Buttazzo, G. C. Rate monotonic vs. EDF: judgment day. *Real-Time Systems 29*, 1 (2005), 5–26.

[16] Carson, M., and Santay, D. NIST net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review 33*, 3 (2003), 111–126.

[17] Chang, X. Network simulations with OPNET. In *Proceedings of the 1999 Winter Simulation Conference* (1999), vol. 1, pp. 307–314.

[18] Chisnall, D. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2007.

[19] Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 3–12.

[20] Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. An experimental time-sharing system. In *AFIPS Conference Proceedings: 1962 Spring Joint Computer Conference* (1962), vol. 21, pp. 335–344.

[21] Cowie, J. H., Nicol, D. M., and Ogielski, A. T. Modeling the global internet. *Computing in Science and Engineering 1*, 1 (1999), 42–50.

[22] Creasy, R. J. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development 25*, 5 (September 1981), 483–490.

[23] Dahmann, J. S. Standards for simulation: As simple as possible but not simpler—high level architecture for simulation. In *DIS-RT '97: Proceedings of the 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications* (Washington, DC, USA, 1997), IEEE Computer Society.

[24] Diestelhorst, S. Scheduling operating-systems. Studienarbeit, Technical University Dresden, June 2007.

[25] Drytkiewicz, W., Sroka, S., Handziski, V., Köpke, A., and Karl, H. A mobility framework (MF) for OMNeT++. In *Proceedings of the Third International OMNeT++ Workshop* (2003).

[26] Engblom, J., Kagedal, D., Moestedt, A., and Runeson, J. Developing embedded networked products using the simics full-system simulator. In *PIMRC '05: Proceedings of the Sixteenth IEEE International Symposium on Personal, Indoor and Mobile Radio Communications* (September 2005), vol. 2, pp. 785–789.

[27] Fall, K. Network emulation in the Vint/NS simulator. In *Proceedings of the Fourth IEEE International Symposium on Computers and Communications* (1999), pp. 244–250.

[28] Fall, K., and Varadhan, K. The network simulator ns-2. `http://www.isi.edu/nsnam/ns/`.

[29] Fujimoto, R. M. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (1990), 30–53.

[30] GDB: The GNU debugger project. `http://www.gnu.org/software/gdb/`.

[31] Gupta, D., Vishwanath, K. V., and Vahdat, A. DieCast: Testing distributed systems with an accurate scale modeling. In *NSDI '08: 5th USENIX Symposium on Networked Systems Design and Implementation* (Aprilx 2008).

[32] Gupta, D., Yocum, K., Yocum, K., McNett, M., Snoeren, A. C., Vahdat, A., and Voelker, G. M. To infinity and beyond: time warped network emulation. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), ACM.

[33] Guruprasad, S., Ricci, R., and Lepreau, J. Integrated network experimentation using simulation and emulation. In *TRIDENTCOM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 204–212.

[34] Harper, J., and Grover, A. GPL paravirtualized drivers for windows. `http://xenbits.xensource.com/ext/win-pvdrivers.hg`.

[35] Hernandez, E., and Helal, A. S. RAMON: Rapid-mobility network emulator. In *LCN '02: Proceedings of the Twenty-Seventh Annual IEEE Conference on Local Computer Networks* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 809–820.

[36] Huang, X., Sharma, R., and Keshav, S. The ENTRAPID protocol development environment. In *INFOCOM '99: Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies* (March 1999), vol. 3, pp. 1107–1115.

[37] Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R. Xen on ARM: System virtualization using xen hypervisor for ARM-based secure mobile phones. In *CCNC 2008: Proceesings of*

the Fifth IEEE Consumer Communications and Networking Conference (2008), pp. 257–261.

[38] The INET framework for OMNeT++. `http://www.omnetpp.org/staticpages/index.php?page=20041019113420757`.

[39] Ingham, D. B., and Parrington, G. D. Delayline: a wide-area network emulation tool. *Computing Systems 7*, 3 (1994), 313–332.

[40] Intel. IA-PC HPET (high precision event timers) specification. `http://www.intel.com/hardwaredesign/hpetspec_1.pdf`, October 2004.

[41] Jansen, S., and McGregor, A. Simulation with real world network stacks. In *Proceedings of the 2005 Winter Simulation Conference* (2005).

[42] Jones, R. The netperf benchmark. `http://www.netperf.org`.

[43] Kamble, N. A., Nakajima, J., and Mallick, A. K. Evolution in kernel debugging using hardware virtualization with xenprobes. In *Proceedings of the 2006 Linux Symposium* (July 2006).

[44] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: The linux virtual machine monitor. In *Proceedings of the 2007 Linux Symposium* (Ottawa, August 2007).

[45] Krasnyansky, M. Universal TUN/TAP driver. `http://vtun.sourceforge.net/tun/`.

[46] Kunz, G. Entwurf und Implementierung einer PlattformabstraktionsschIcht für modulare Kommunikationsprotokolle. Diplomarbeit, RWTH Aachen University, December 2007.

[47] Lawton, K. P. Bochs: A portable PC emulator for Unix/X. *Linux Journal 29* (1996), article no.7.

[48] Liu, B., Figueiredo, D., Guo, Y., Kurose, J., and Towsley, D. A study of networks simulation efficiency: fluid simulation vs. packet-level simulation. In *INFOCOM 2001: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* (2001), vol. 3, pp. 1244–1253.

[49] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[50] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35*, 2 (2002), 50–58.

[51] Misra, J. Distributed discrete-event simulation. *ACM Computing Surveys 18*, 1 (1986), 39–65.

[52] Nakajima, J., Mallick, A., Pratt, I., and Fraser, K. x86-64 XenLinux: Architecture, implementation, and optimizations. In *Proceedings of the 2006 Linux Symposium* (Ottawa, August 2006).

[53] Nguyen, A. Q., and Suzaki, K. Xenprobes, a lightweight user-space probing framework for xen virtual machine. In *ATEC '07: Proceedings of the 2007 USENIX Annual Technical Conference* (June 2007), pp. 15–28.

[54] Nicol, D., Goldsby, M., and Johnson, M. Fluid-based simulation of communication networks using SSF. In *Proceedings of the 1999 European Simulation Symposium* (October 1999).

[55] Payne, B., de Carbone, M., and Lee, W. Secure and flexible monitoring of virtual machines. In *ACSAC '07: Twenty-Third Annual Computer Security Applications Conference* (December 2007), pp. 385–397.

[56] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[57] Postel, J. RFC 792: Internet control message protocol. `http://www.ietf.org/rfc/rfc0792.txt`, September 1981.

[58] Pratt, I., Fraser, K., Hand, S., Limpach, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A. Xen 3.0 and the art of virtualization. In *Proceedings of the Seventh Linux Symposium* (July 2005), vol. 2, pp. 65–77.

[59] Riedl, J. Time and packet synchronization of the OMNeT++ simulator. Studienarbeit, University of Tübingen, August 2005.

[60] Rizzo, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[61] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the Ninth USENIX Security Symposium* (Berkeley, CA, USA, 2000), USENIX Association, pp. 10–25.

[62] Ros, D., and Marie, R. Estimation of end-to-end delay in high-speed networks by means of fluid model simulation. In *Proceedings of the Thirteenth European Simulation Multiconference* (June 1999).

[63] Rosenblum, M., and Garfinkel, T. Virtual machine monitors: current technology and future trends. *Computer 38*, 5 (May 2005), 39–47.

[64] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. Binary translation. *Communications of the ACM 36*, 2 (1993), 69–81.

[65] Strachey, C. Time sharing in large fast computers. In *Proceedings of the International Conference on Information Processing* (1959).

[66] TCPDUMP/LIBPCAP public repository. `http://www.tcpdump.org`.

[67] Tüxen, M., Rüngeler, I., and Rathgeb, E. P. Interface connecting the INET simulation framework to the real world. In *SIMUTools '08: Proceedings of the First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems* (March 2008).

[68] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[69] Varga, A. The OMNeT++ discrete event simulation system. In *ESM '01: Proceedings of the European Simulation Multiconference* (June 2001).

[70] Varian, M. VM and the VM community: Past, present, and future. `http://www.princeton.edu/~melinda/25paper.pdf`, 1997.

[71] Villa, F. J., Acacio, M. E., and García, J. M. Evaluating IA-32 web servers through simics: a practical experience. *Journal of Systems Architecture 51*, 4 (2005), 251–264.

[72] VMWare homepage. `http://www.vmware.com`.

[73] Yan, A., and Gong, W.-B. Time-driven fluid simulation for high-speed networks. *IEEE Transactions on Information Theory 45*, 5 (July 1999), 1588–1599.

[74] Yourst, M. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS '07: Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software* (April 2007), pp. 23–34.

[75] Zeng, X., Bagrodia, R., and Gerla, M. Glomosim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation* (May 1998), pp. 154–161.

A

Synchronization System Setup

This appendix will give a more comprehensive overview over what configuration options exist for the different programs that together form the synchronization system developed for this thesis, and how to set up synchronized network emulation between an OMNeT++ simulation and a Xen domain.

A.1 Configuration Options

A.1.1 Stand-Alone Synchronization Server

The stand-alone synchronization server is a user-space application that is configured via its configuration file config.sync which the server expects to be in the same directory as the executable. It is in a standard INI file format, i.e. it consists of sections that contain variables and their assignments. Note that while the server comes with some rudimentary support functionality to run as client, there is little use for this feature at this point in time, as it can interface with neither Xen nor OMNeT++ in this function. Consequently, this setup has not been tested.

Section	Name	Description
GENERAL	mode	“client” or “server”
SERVER	srv_barrier_interval	An integer that sets the size of a CTW synchronization time slice, in μs.
	srv_brdcast_address	The IP broadcast address of the network, in dotted decimal notation.
	server_port	The port the server listens on.
CLIENT	client_id	A numerical identifier for the client. Must be unique over all clients participating in the synchronization.
	client_type	An identifier that can describe the type of client.
	client_sync_server	The IP address of the server the client will connect to, in dotted decimal notation.
	client_port	The port the client listens on, i.e. the port on which the broadcast `run` messages are sent.

A.1.2 Xen Synchronization Client

The Xen synchronization client comes in the form of a kernel module. All options are passed on the command line, or can optionally be added to the module options file. The location of this file depends on the used Linux distribution, but a typical location is `/etc/modprobe.d/options`. Note that the option to use the kernel module as synchronization server was not thoroughly tested.

Name	Type	Description
run_as_server	int	If set to 1, allows the module to function as server instead of client.
server_address	char[]	The IP address of the server, in dotted decimal notation.
server_port	int	The port the synchronization server listens on.
broadcast_address	char[]	The IP broadcast address of the network, in dotted decimal notation.
client_port	int	The port the client listens on, i.e. the port on which the broadcast `run` messages are sent.
cli_id	int	A numerical identifier for the client. Must be unique over all clients participating in the synchronization.
cli_descr	char[]	A description string that can be used to identify the client at the server.
sync_domain	int[]	A comma-separated list of domains that are to be synchronized on this machine.
barrier_time	int	In server mode: the size of a CTW synchronization time slice.

A.1.3 OMNeT++ Synchronization Client–Scheduler

The synchronization client written for OMNeT++ is part of the scheduler that drives the simulator according to the assigned time slices (see Section 3.2.2 for more information). As such, it is configured the same way as any other OMNeT++ component, i.e. by setting the options in the omnetpp.ini file that belongs to the simulation model. First of all, the option `scheduler-class` has to be set to `cSync-Scheduler` in the `General` section. The other options are set in a section named `Sync`.

Name	**Type**	**Description**
sync_enabled	boolean	If "no", the scheduler will not be synchronized and work like a normal scheduler.
client_id	int	A numerical identifier for the client. Must be unique over all clients participating in the synchronization.
client_description	string	A description string that can be used to identify the client at the server.
sync_server	string	The IP address of the server, in dotted decimal notation.
sync_server_port	int	The port the synchronization server listens on.
sync_client_port	int	The port the synchronization client listens on.
debug	boolean	If "yes", print debugging information to STDOUT while running.
sync_unit_scaler	double	A scale factor. The time slice that is received from the server is divided by this number. Since the synchronization server described in Section A.1.1 sends the time in μs, but OMNeT++ expects the time in seconds, it is set to 1000000 and should generally not be changed.

A.1.4 Emulator–Tunnel: Xen Side

The tunnel is set up by an application `tap-udptunnel` that expects three command line options: First, the IP address of the endpoint of the tunnel, i.e. the computer the simulation is running on. Second, a part number of the endpoint. Third, a flow-id that uniquely identifies this tunnel. If there are several Xen hosts connected to the simulation, they will all share the IP address and port endpoints, but they will each receive a unique flow-id. As next step, a new bridge has to be created, and the virtual interface `vifN.0` corresponding to the synchronized domain has to be bridged with the tap devices that was created by starting the `tap-udptunnel` program. Finally, ARP has to be switched off on the bridge.

A.1.5 Emulator–Tunnel: OMNeT++ Side

The configuration options are spread over two files. The general options are set in the omnetpp.ini file, in a section named `cExternal-Tunnel`.

Name	**Type**	**Description**
tunnel_enabled	boolean	If “no”, no functional coupling will take place. Consequently, no communication between simulation and Xen domain will be possible.
tunnel_destination	string	The IP address of the endpoint of the tunnel, in dotted decimal notation.
tunnel_port	int	The port of the endpoint of the tunnel.

The flow-id, which uniquely identifies each Xen domain’s traffic, is set in the network description file. Since each Xen domain is identified with a network node of the type `cExtHost` inside the simulation (see Section 3.3), the flow-id is set as one of its parameters with the name `flowid` and an integer as value.

A.2 Synchronized Network Emulation Setup

1. Boot one computer into Xen, and one computer into Linux.
2. Start the domain that is to be synchronized on the Xen computer.
3. Start the `tap-udptunnel` script on the Xen computer.
4. Bridge the virtual interface that belongs to the domain that is to be synchronized with the newly created tap device. Do not forget to switch off ARP on the bridge.
5. Start the synchronization server on the Linux computer (or the Xen computer, or any other third computer).
6. Start OMNeT++ on the Linux computer.
7. Load the kernel module on the Xen machine.
8. Start the OMNeT++ simulation.