Porting the QEMU virtualization software to MINIX 3

Master's thesis in Computer Science

QEMU
open source processor emulator

Erik van der Kouwe

Student number 1397273

erik@erisma.nl

Vrije Universiteit Amsterdam

Faculty of Sciences

Department of Mathematics and Computer Science

Supervised by dr. Andrew S. Tanenbaum

Second reader: dr. Herbert Bos

12 August 2009

Abstract

The MINIX 3 operating system aims to make computers more reliable and more secure by keeping privileged code small and simple. Unfortunately, at the moment only few major programs have been ported to MINIX. In particular, no virtualization software is available. By isolating software environments from each other, virtualization aids in software development and provides an additional way to achieve reliability and security. It is unclear whether virtualization software can run efficiently within the constraints of MINIX' microkernel design. To determine whether MINIX is capable of running virtualization software, I have ported QEMU to it. QEMU provides full system virtualization, aiming in particular at portability and speed. I find that QEMU can be ported to MINIX, but that this requires a number of changes to be made to both programs. Allowing QEMU to run mainly involves adding standardized POSIX functions that were previously missing in MINIX. These additions do not conflict with MINIX' design principles and their availability makes porting other software easier. A list of recommendations is provided that could further simplify porting software to MINIX.

Besides just porting QEMU, I also investigate what performance bottlenecks it experiences on MINIX. Several areas are found where MINIX does not perform as well as Linux. The causes for these differences are investigated. For practical usage, the difference is found to be small. Most bottlenecks can be resolved through minor changes and only one of the minor issues appears to be due to the microkernel design.

This thesis does not only provide a report on my research about QEMU on MINIX, but also aims to provide information to those who intend to port software to MINIX. It contains a detailed report on the issues encountered while porting QEMU, so that others may avoid the pitfalls I found. As such, it can serve as a manual for porting projects.

Brief table of Contents

1 - Introduction....8
2 - Virtualization....13
3 - Issues encountered and changes made....31
4 - How to use QEMU on MINIX....73
5 - Performance measurements....81
6 - Conclusions....101
Bibliography....102
Appendix A - Contents of the CD-ROM....104
Appendix B - Performance measurements....106

Detailed table of Contents

1 - Introduction....8
1.1 - Context....8
1.2 - Problem statement....8
1.3 - MINIX....9
1.4 - QEMU....11
1.5 - Structure of this thesis....12
2 - Virtualization....13
2.1 - Introduction....13
What is virtualization?....13
Why are virtual machines useful?....13
Theoretical background....14
2.2 - Possible approaches....16
Dynamic Binary Translation....16
Paravirtualization....18
Previrtualization....19
Operating system level partitioning....20
Application virtual machine....20
Hardware supported....21
2.3 - QEMU implementation....22
User-level memory management unit....23
Code generation....24
No kernel-level components....29
3 - Issues encountered and changes made....31
3.1 - General....31
3.2 - Changes made to MINIX....33
Addition of the setitimer function....33
Implementation of the pread64 and pwrite64 functions....36
Signal handling bug....38
Use of the select function with the /dev/eth device....38
3.3 - Porting QEMU....39
General remarks on porting software to MINIX....39
Changes related to compilation....44
Changes related to code generation....47
Changes related to networking....48
Miscellaneous changes....49
Missing functionality....51
3.4 - Features added in QEMU for MINIX....53
Curses support....53
Memory allocation recommendation....53
Networking....54
Opcode histograms....59
Running deterministically....60
Simple profiling of QEMU....62
3.5 - libSDL....62

The configure and configure.in files....63
Changes to SDL files....63
Build file....64
3.6 - Debugging QEMU....64
Causing crashes to occur early....65
Lack of double and triple fault....66
Logging system calls....66
Making the core file more readable....67
Parallel testing....68
Profiling supported by GCC....68
Using MINIX' information server....69
3.7 - Testing QEMU....70
3.8 - Discussion....71
4 - How to use QEMU on MINIX....73
4.1 - Installing QEMU on MINIX....73
What has to be done....73
Installing using the installation script....73
Installing manually....75
4.2 - Running QEMU on MINIX....78
Running the pre-made disk images....78
Setting up a new virtual machine....79
5 - Performance measurements....81
5.1 - Methodology....81
Measuring QEMU performance....81
Measuring impact of the HZ constant....84
5.2 - Results....86
Performance of MINIX as a guest operating system....86
Performance of QEMU itself....89
Performance of MINIX as a host operating system....93
Impact of the deterministic mode....95
Recursive emulation....96
Impact of the HZ constant....97
5.3 - Discussion....99
6 - Conclusions....101
Bibliography....102
Appendix A - Contents of the CD-ROM....104
Appendix B - Performance measurements....106
B.1 - Benchmarking guest operating systems running on QEMU....106
B.2 - Impact of the LLDT instruction....108
Changes made to MINIX....108
Performance impact....109
B.3 - Benchmarking the impact of the clock frequency....109

Table of figures

Figure 1: Structure of the MINIX 3 operating system, each box denoting a process isolated from the other processes; source: [21]....10
Figure 2: The virtual machine manager as a regular user process....13
Figure 3: Main loop for binary dynamic translation....17
Figure 4: Main loop for paravirtualization....18
Figure 5: Main loop for hardware-supported virtualization....22
Figure 6: MMU operation and terminology on x86....23
Figure 7: Function call with relative displacement on x86....25
Figure 8: Steps in the compilation of QEMU using the Dyngen tool....26
Figure 9: Translation of a x86 function using intermediate code....27
Figure 10: Processes involved in network emulation using qemu-vswitch....57
Figure 11: qemu-vswitch routing an outgoing DHCP discover packet....57
Figure 12: qemu-vswitch routing an incoming DHCP offer packet....58
Figure 13: Interaction between processes involved in measuring performance....82
Figure 14: MINIX vs. Linux as a guest platform....87
Figure 15: Slow-down caused by emulation....91
Figure 16: Amount of data sent in the io_network_throughput benchmark....92
Figure 17: MINIX vs. Linux as a host platform....94
Figure 18: Slow-down of deterministic mode compared to default mode on MINIX....96
Figure 19: Relationship between overhead caused by the operating system and timer frequency, on MINIX patched to have a variable clock frequency....98

Table of tables

Table 1: Possible schedule when two instances of QEMU read the same disk.....37
Table 2: Advantages and disadvantages of several virtual networking approaches supported by QEMU on MINIX 54
Table 3: Operations used by the guest to control the opcode histogram feature.....60
Table 4: Operations used by the guest to read the host time.....62
Table 5: Benchmarks performed by the Benchmark program.....83
Table 6: Performance of recursive emulation; time to recompile the MINIX image in seconds 97
Table 7: Average run-times (in seconds) for each benchmark on each configuration.....107
Table 8: Averaged performance measurements of various operating systems at various clock frequencies.....110

1 - Introduction

1.1 - Context

MINIX 3 is a free software operating system which was developed to show that operating systems can be made more reliable, more secure, smaller and more understandable. It achieves this by keeping the kernel, the part of the operating system that directly controls the hardware, as small and simple as possible. This reduces the likelihood of bugs being able to bring down the entire system. All other functionality is provided in the context of separate processes called 'servers'. Since the kernel isolates processes from each other, bugs in servers can do only limited damage. If something goes wrong, they can simply be restarted. The operating system itself keeps on running and users of the system can continue working without disruption.

Currently, although MINIX 3 is fully functional, only few major applications and hardware drivers have been ported to it. Sufficient availability of ported software is important both scientifically and practically. Scientifically, such software serves to test the ability and performance of a reliable microkernel operating system. By providing additional means to test MINIX 3, they point towards possible improvements. Practically, a lack of applications and hardware drivers holds people back from switching to MINIX 3 for everyday use. Since MINIX 3 is an open source project, it would greatly benefit from the contributions that new users can make. Software that assists in software development is especially important in this way, since it attracts exactly the group of people that is most important for further improving MINIX 3.

QEMU is a free software virtualization program. It emulates computer hardware to allow one operating system to run as a process inside another. This is useful for a number of purposes, including software testing and operating system development. For example, it allows one to easily restore the operating system to a previous state after a change that worked out badly, makes it possible to have the equivalent of a core dump that is useful for kernel debugging and it allows one to test programs on multiple operating systems without rebooting. Moreover, it allows one to run applications even if they have not (yet) been ported to host operating system. In the case of MINIX, it could for example be used to run a graphical browser, something which MINIX itself currently lacks. No general purpose virtualization program is currently available in MINIX 3, so if QEMU were to be ported that would benefit further development on MINIX 3.

1.2 - Problem statement

My aim is to port the QEMU virtualization software to the MINIX 3 operating system to evaluate what difficulties are encountered and what level of functionality and performance can be achieved. It is not clear in advance that this can be done, since MINIX 3 lacks some features that are common in other operating systems. Examples of such missing features are virtual memory and a high-resolution timer. It is also better secured against self-modifying code, which may be an issue because code generation is a core means of speeding up emulation. Moreover, microkernel systems sacrifice performance for reliability and security.

Speed is an important factor determining whether a virtual machine is useful or not. Hence, QEMU provides an opportunity to find performance bottlenecks.

One additional goal of this project is to provide a manual for porting complex software to MINIX. Such a manual is currently lacking and would encourage others to get started with porting. In particular, the Java Virtual Machine and the MONO framework use techniques similar to those in QEMU to be able to run Java and .NET code across platforms without recompilation.

This research is driven by the research question 'Can MINIX 3 run virtualization software?' This question gives rise to a number of sub-questions:

- What issues does one encounter when porting complex software to MINIX 3?
- Is it necessary to change MINIX 3 to be able to run QEMU?
- Is the microkernel design an obstacle for performance?
- Can bottlenecks be solved within this design?
- Is QEMU on MINIX 3 usable in practice?

The primary goal of this master's thesis is to provide answers to these questions based on porting QEMU to MINIX 3 and testing it. Besides providing just answers, this thesis is meant to serve as a manual for porting software to MINIX 3 and explain how QEMU can be used in MINIX 3. Moreover, it provides a list of areas where MINIX' performance could potentially be improved and makes virtualization available on the MINIX 3 operating system.

1.3 - MINIX

MINIX is an operating system created from scratch, originally serving as an educational tool to teach students of computer science about operating systems. It is suitable for this purpose because it is small and simple, containing no unnecessary detail and being easily changed and recompiled. With the latest major version, MINIX 3, this scope has been expanded. Besides the original educational goal, it now strives to combine a small footprint with high reliability, making it ideal for embedded applications [20].

Reliability is achieved in MINIX by adopting the microkernel model, in which operating system code runs with as few privileges as possible. Only those parts that need to be in full control of the hardware are placed in the kernel, while many operating system services and drivers are run as separate server processes. These processes have few privileges, so that they are unlikely to bring down the system if they crash. Moreover, it is possible to restart these servers whenever they cease to function properly. The microkernel model can be contrasted against the monolithic model, in which most or all of the operating system services run at the highest privilege level. In such a system overhead from communication between the various parts of the operating system is smaller, but if any of these parts malfunctions then the operating system irreversibly becomes unstable and must be rebooted. The occurrence of such bugs becomes more likely as the system becomes more complex; all software contains bugs, with the number of bugs generally proportional to the complexity. As complexity is typically measured as the number of lines of source code [3], reducing the size of an operating system kernel is highly desirable.

The structure of MINIX 3 is shown in Figure 1 and described at length in [21]. In this drawing, privileges increase from top to bottom. The kernel layer has full control over the hardware. It cannot formally be called a process as it is not scheduled, but is rather called by the hardware through interrupts whenever its services are needed. Whereas monolithic operating systems run process management, file systems and drivers at this level, MINIX only implements the bare minimum. This includes operations such as task switching and hardware access involving memory, input and output and interrupt handlers. The system task, which also runs in kernel mode, makes these facilities available to the higher levels. By making system calls, other processes can request that the kernel perform actions for them if they have sufficient privileges. Even device drivers cannot directly access the hardware, they need to be granted access and have to go through the kernel. The only exception is the clock driver, which is part of the kernel due to its time critical nature and the fact that this service is also needed for scheduling processes. It should be noted that this does not increase the size of the kernel by much as it is considerably smaller than most other drivers.

None of the layers outside the kernel has direct access to the hardware. They do differ, however, in the kinds of system calls that they are allowed to make. Device drivers are allowed many privileges, in particular regarding hardware access. Although they cannot access the hardware directly, they have system calls at their disposal to do tasks such as moving blocks of memory around, performing device IO and receiving interrupts. Through this mechanism, implementing drivers for all kinds of devices is possible while the chances of driver crashes causing serious damage is lower than on other systems.

Server processes can also call the kernel and use the drivers at the lower level, but they are allowed to use only those calls that they really need to do their job. In particular, servers in contrast with device drivers have little hardware access. The servers implement high-level abstractions that are needed by user processes to be able to use the system, for example though the POSIX API. The same services provided by the process manager, file system server and network server are typically performed by kernel-level components in other operating systems. One server that is specific to the MINIX approach to reliability is the reincarnation server. This server can restart drivers and services when they malfunction, so that ideally the user could continue working despite driver crashes. Although this is currently more theory than practice, the MINIX 3 team is working hard to make it reality.

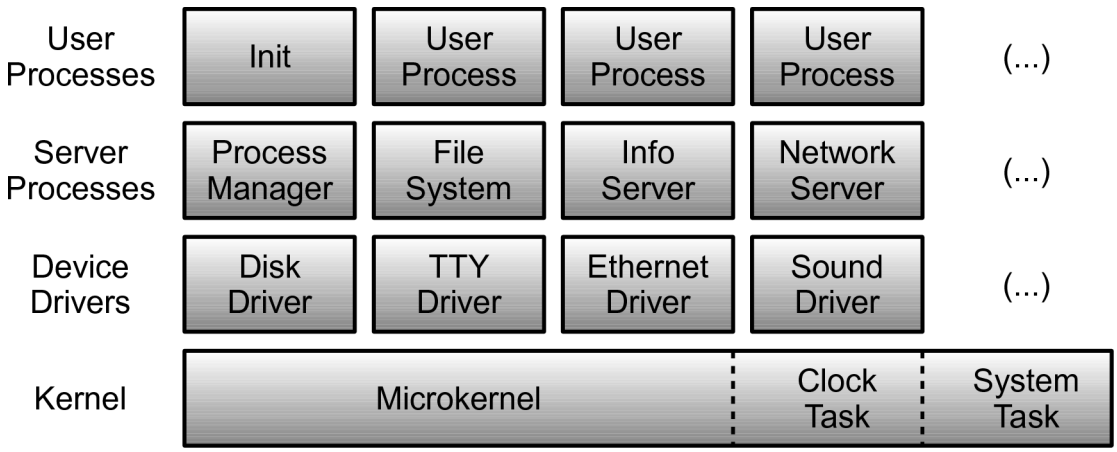

Figure 1: Structure of the MINIX 3 operating system, each box denoting a process isolated from the other processes; source: [21].

User processes, which form the top layer, do not have any privileges to call the kernel or drivers directly. Instead, they use the services provided by the servers, which in turn call the drivers or the kernel whenever needed. A typical request, such as reading a file using the `read` POSIX call, is sent by the kernel to the appropriate server, in this case the file system. This server validates whether the process is allowed to perform the operation. Unlike the user process from which the request originated, the file system server is allowed to communicate directly with device drivers. If the call is accepted, it requests the proper disk block from the disk driver. The disk driver, in turn, is allowed to ask the kernel to send IO requests to the disk drive and can receive notifications whenever the data has been read. By using this approach, each component gets only the privileges that are really needed and the low-level high-privilege parts are kept as simple as possible.

Because MINIX is a microkernel system in which operating system services run as separate processes, inter-process communication is an important aspect of its design. Each of the steps in the previous example requires communication between two processes. MINIX isolates processes from each other, only allowing communication by means of synchronous message passing. Whenever a message is sent, it is also checked whether the source process is allowed to sent to the specified destination process. This is an additional protection to make sure that user processes cannot access drivers and the kernel directly. By using only synchronous message passing, inter-process communication is kept simple. All communication is done through the same mechanism and there is no need for the operating system to handle concurrency or keep queues; a process simply blocks until its message is delivered.

Why is this simple microkernel operating system interesting from a scientific perspective and as a host platform for QEMU in particular? First, it demonstrates that microkernel operating systems can work in practice and it allows for experiments to reveal their capabilities and their performance. Since virtual machine emulation is demanding, having QEMU run on it provides an additional test for its capabilities as well as a new way to measure its performance. Moreover, it serves as an example of how modularity, reliability and self-healing can be achieved in operating systems. Virtual machines are used extensively in server farms to achieve high availability and a highly reliable operating system would provide a good basis to reach this goal.

1.4 - QEMU

QEMU is an open source virtualization program which aims to be portable, fast and which can be used for general purpose full system emulation [5]. Although some competitors, such as VMWare Workstation and Microsoft Virtual Server, can provide faster emulation, their source code is not available so that it is not possible to port them to MINIX. Xen is an open source alternative, but it cannot run within a host operating system and, unless hardware support is available, supports only guest operating systems which have been modified to run on it. For this reason, it is not general purpose like the other programs mentioned. Another well-known open source CPU emulation program is Bochs, but this program is considerably slower than QEMU because it does not use dynamic code generation. The difference between Bochs and QEMU is similar to the difference between an interpreter and a compiler; the former has more control over the program and is can therefore provide useful debugging features, but this comes at a considerable performance cost compared to the latter. It should be noted that some hardware emulation code from Bochs has been used in QEMU, so the

programs are not entirely unrelated. Given the goal of testing MINIX' capabilities and the aim to give MINIX users access to general purpose full system emulation with performance that is acceptable for practical use, QEMU is the most suitable choice.

1.5 - Structure of this thesis

Besides answering the research questions, this thesis is also intended to serve as a guide for those who intend to port software to MINIX as well as provide instructions to those who intend to use QEMU on MINIX. Hence, some reading audiences may find some chapters and sections more useful than others.

The next chapter introduces the topic of virtualization from both a theoretical perspective and from the more practical viewpoint of its implementation in QEMU. The former part is based on my bachelor's thesis and provides an overview of the issues encountered in virtualization as well as the various approaches that are available to address them. The latter part elaborates on the specific design choices made in QEMU with regard to these issues.

Chapter 3 focuses on the changes that I needed to make to both QEMU and MINIX and the ways in which I addressed the issues I encountered. It is practical in nature and I invite those who intend to port software to MINIX to read it to allow them to avoid the obstacles that I encountered. This chapter also considers the ways in which QEMU is different on MINIX than on other platforms. Reading the section on this topic is advisable for those who wish to use the more advanced features of QEMU in MINIX.

For those who merely want to try QEMU on MINIX, I would recommend chapter 4. This chapter provides instructions to install, compile and run QEMU. As such it has some overlap with the documentation provided for QEMU, but focusses in particular on those issues that are specific to MINIX.

In chapter 5, I discuss how I have tested QEMU's performance and present the outcomes from those measurements. Important differences with Linux are discussed, providing an overview of which bottlenecks MINIX has and how they could be addressed. Additionally, I present the results of tests I performed to justify some design choices I have made in porting QEMU.

The final chapter concludes the thesis by summarizing the main findings. In doing so, the research questions are answered and the main consequences of the answers are discussed.

2 - Virtualization

2.1 - Introduction

What is virtualization?

Virtualization is a technique which allows one to partition a computer system in multiple systems that are isolated from each other. Each of these provides a software environment which is very similar to that of a physical computer. Such an environment is called a virtual machine (VM).

One will typically want to install an operating system on a virtual machine to be able to run applications. This guest operating system assumes that it has complete control of the computer, and it will attempt to access it's hardware. This cannot be allowed, since the hardware is shared with guest operating systems running on other virtual machines. A program called virtual machine monitor (VMM) or hypervisor is needed to make sure all resources are shared properly.

The role of a VMM dividing resources between operating systems differs from that of a kernel dividing resources between applications. The main difference is that the latter typically provides an abstraction of physical devices, while the former does not change the abstraction level [19]. The VMM should present a faithful low-level interface to virtualized hardware.

The VMM itself may have full hardware access, but it can also can be a normal application running on an operating system. In this case the operating system on which the VMM runs is called the host operating system. This situation is shown in Figure 2.

Why are virtual machines useful?

Server farms

One can use virtual machines to run multiple isolated virtual servers on a single physical server. Virtualization is more efficient despite its overhead because servers have to be sufficiently powerful for peak loads, but experience much lower load levels most of the time. By consolidating many servers on one machine, variation in the overall load level is reduced. Additionally, some services cannot normally be combined on a single server for security reasons; it might be a bad idea to put one's critical services such as the file server or mail

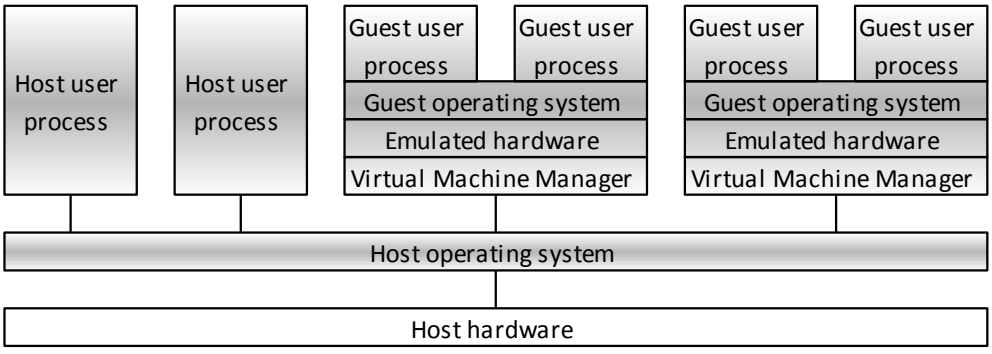

Figure 2: The virtual machine manager as a regular user process

server together on one machine with a web server that is more vulnerable. Because of the isolation provided by virtualization, they can be put on the same physical machine so hardware can be used more efficiently and costs can be decreased.

For an internet hosting company virtualization can be used to allow customers full access to a virtual server without endangering other servers on the same physical machine. This way, customers have a large degree of freedom at relatively low cost.

Another advantage of using virtual servers is the fact that they can easily be moved to other computers. Typically the interface to the virtualized hardware does not depend on the actual hardware, so the guest operating system does not even notice the move. This can be used to minimise downtime after restoring a backup to different hardware. This makes maintenance easier and increases availability.

Software development

Virtual machines have their uses in software development as well. They provide a way to switch between different (versions of) operating systems easily and quickly. This is very useful for testing and debugging software on multiple platforms.

VMMs may also allow the user to make snapshots of virtual machines. This means one can test something and revert the machine to it's original state afterwards. This is again very useful for testing and debugging.

Yet another application for development is to debug low-level software, such as kernels. A VMM could provide a kernel debugger with much information about what is going on. It also makes recovery from crashes easier, since one can simply use a previous snapshot.

Untrusted software

Another application which is especially useful nowadays is the ability to run untrusted software in an environment where it can do no damage. Examples are running potential malware downloaded from the Internet and opening suspicious e-mail attachments. If the VMM is secure enough, there is no way for malware and viruses to infect the physical machine or other virtual machines.

Security research

Finally virtual machines provide an easy way to build secure “honeypots.” These are unprotected machines which are connected to the Internet. The purpose is to get information about new methods to exploit flaws in operating systems and applications. By using virtual honeypots, these exploits are less likely to do damage to the system itself. This may make investigation and recovery easier. They also allow having multiple honeypots with different (versions of) operating systems running simultaneously.

Theoretical background

Possibility of efficient virtualization

Popek and Goldberg [17] investigated sufficient conditions which allow a computer architecture to support virtual machines. They defined a virtual machine as “an efficient,

isolated duplicate of the real machine." Although their research was motivated by the question why IBM 360/67 could support virtual machines while DEC PDP-10 could not, their criterion still applies to modern architectures.

Central to Popek and Goldberg's theorem is the distinction between supervisor and user modes. The supervisor mode allows complete access to the machine and is typically used by operating system kernels and VMMs, while the user mode is more limited and is typically used by applications. They further define a trap operation, which places the processor in a stored state (typically the supervisor mode) while saving the current state.

Popek and Goldberg consider some properties that individual instructions can have:

- A privileged instruction performs a trap operation in user mode, but does not trap in supervisor mode;
- A control sensitive instruction can change the operating mode or virtual memory mappings;
- A behaviour sensitive instruction executes in different ways depending on operating mode or virtual memory mappings;
- A sensitive instruction is control sensitive and/or behaviour sensitive.

Having defined these terms, one can state the following theorem: "A virtual machine monitor may be constructed if the set of sensitive instructions for the computer is a subset of the set of privileged instructions." The virtual machine is constructed by letting the VMM operate in supervisor mode and the virtual machine in user mode. This way the monitor can emulate sensitive instructions, since it is notified by a trap operation. nonsensitive instructions can safely be executed directly.

Virtualizability of IA-32

The IA-32 architecture is very widely used in personal computers and servers which run Windows, Linux or, more recently, Mac OS. It is also the only architecture currently supported by current versions of MINIX 3. It evolved from (and is still compatible with) the instruction set architectures used by the 8086 and it's successors and is therefore also commonly known as x86. The Pentium D and Core Duo chips from Intel and Opteron and Athlon 64 X2 from AMD are examples of modern implementations of IA-32.

IA-32 defines four security rings numbered 0 through 3. Ring 0 can be considered the supervisor mode, while the other rings correspond with the user mode. The architecture defines privileged instructions which, when executed in user mode, trap by calling an interrupt handler in ring 0. Virtual memory is implemented through segmentation and paging. Segments are identified by 16-bit segment selectors which contain the number of the least privileged ring allowed to access them. The operating system's perspective of IA-32 is described at length in [8].

Robin and Irvine [18] have investigated the IA-32 architecture and have found many instructions that are sensitive, but not privileged. An example is the `push cs` instruction, which pushes onto the stack the selector for the segment containing the currently executing code. Since this segment selector contains the number of the current security ring, this instruction is behaviour sensitive.

The lack of native virtual machine support has lead to the use of many different techniques on the IA-32 platform. The approaches I mention are applicable to other architectures, but I will focus on IA-32. I will also discuss two approaches which are highly similar to virtual machines and provide the same advantages, but which are not virtual machines according to the definition I presented before.

Recently Intel has added true virtualization support to their newest IA-32 chips by including an extension instruction set called "Virtual Machine Extensions." AMD has also introduced a similar (but incompatible) instruction set extension to achieve the same goal. I will discuss this technology as well.

2.2 - Possible approaches

Dynamic Binary Translation

Dynamic binary translation can be seen as an advanced way to do emulation. In case of pure emulation, the host software implements the instructions that are available on the CPU. This allows it to interpret the instructions supplied by the guest. As such, emulation is the most obvious approach to build virtual machines on hardware platforms that have no native support for them. Bochs is an example of a program which emulates IA-32 CPUs.

Unfortunately pure emulation provides very poor performance, since executing a single guest instruction typically takes many instructions on the host machine. This lack of efficiency means that an emulated machine does not satisfy the Popek and Goldberg definition for a virtual machine. As such, I will not consider emulation separately.

Dynamic binary translation overcomes the performance limitations of emulation by translating the instruction stream to host instructions which can be executed natively. In this step sensitive instructions are replaced with calls to the VMM. The translated instruction stream can be cached. Because of this, only a small part of the time is spent on translation. The main loop of a pure dynamic binary translation VMM is shown in Figure 3.

Ung and Cifuentes [24] provide details on how one can implement binary translation.

If the host has the same architecture as the guest and the VMM has access to kernel mode, dynamic binary translation can be sped up by directly executing code in some cases. When running in kernel mode, the VMM is able to reproduce the environment the virtual machine runs in on the physical machine. When running guest code in user mode, interrupts are raised whenever the guest attempts to access hardware, allowing the VMM to intervene and emulate virtual hardware instead.

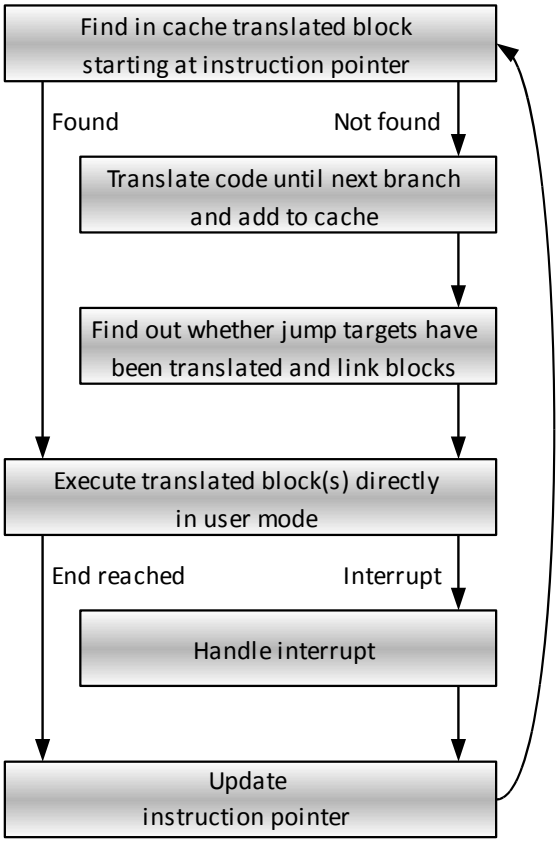

Figure 3: Main loop for binary dynamic translation

Unfortunately this form of emulation is not and cannot be perfect since Robin and Irvine [18] found that the IA-32 architecture is not virtualizable. Kernel mode guest code can easily find that it is really running in user mode. This is likely to cause it to fail and at least allows it to detect the VMM. For this reason kernel code generally still has to be emulated, causing direct execution to be alternated with binary dynamic translation. Even user level guest code can detect that it is not running on the bare hardware. On IA-32 the `sidt` instruction, which allows one to detect the location and size of the interrupt table in physical RAM, is completely unprotected. This allows applications to find out where the VMM stores the interrupt table, which is likely not the same location used by the guest operating system. This instruction is useless for user mode code and hence will not cause it to fail, but allows it to detect emulation. For some purposes of virtualization this is unacceptable, for example in the case of virtual honeypots.

Dynamic binary translation is widely used. Market leader VMWare created several VMMs which are based on this technique: VMWare Workstation runs as an application on Windows and Linux and VMWare ESX server runs without an operating system. The architecture of the virtualization software is described in [26]. For the server version, this software runs on a proprietary microkernel operating system [25].

Another example of a VMM which uses dynamic binary translation is QEMU. This program provides more insight in the implementation of this technique, since it is open source. Its author describes the internals in [4]. QEMU translates CPU instructions to C code, which is compiled using the GCC compiler. This results in very good portability, since GCC has been ported to many platforms. Translation happens in blocks ending at the next potential jump or

important change in CPU state (such as changing mode of operation). These blocks are stored in a cache. Pages containing translated code are marked read-only, and by handling the resulting protection faults QEMU can invalidate the cache when code changes.

Other examples of virtualization products which use dynamic binary translation include Microsoft VirtualPC and Virtual Server, Parallels Workstation and Serenity Virtual Station.

With dynamic binary translation much of the code is translated only once and can be executed natively. I therefore expect the chaining of translated blocks to have the most important impact on performance. Benchmarks show [1] that a highly optimised binary translation VMM such as VMWare Workstation can reach good speeds, but worse results should be expected in branch-intensive or self-modifying code.

The host kernel should ensure that the guest code is executed in a user mode ring, typically ring 3. All of this code is generated by the VMM. This provides the additional guarantee that the guest cannot attempt to call the host kernel directly. This results in double protection, making it unlikely that a single bug in either the kernel or the VMM compromises the isolation. As such, dynamic binary translation can be very robust.

Binary translation VMMs can be highly portable. In principle, the guest operating system does not matter at all, as long as the VMM faithfully implements all hardware interfaces it requires. As QEMU shows, even emulation of different CPU architectures is achieved reasonably easily, although direct execution is not possible in this case.

A binary translation VMM can run either as an unprivileged application within an operating system or with full control without operating system. This section has discussed examples of both.

Paravirtualization

The approach I described before tries to mimic the environment in which the operating system is running on a real machine. For this reason it is also called full system virtualization. More efficient virtualization may be possible if the VMM cooperates with the guest operating system. This approach is called paravirtualization.

Xen is an example of a virtual machine monitor which uses paravirtualization. It requires that guest operating systems be changed to make their kernel run in ring 1 instead of ring 0. The ported kernel uses “hypercalls” to replace sensitive instructions. This is done by placing parameters in registers and then calling an interrupt handler. Xen itself runs in ring 0. The approach is shown in Figure 4.

Xen's creators ported Windows XP, Linux and BSD to run on Xen [2], although Windows for Xen is not publicly available. MINIX for Xen is also available, and the document [13]

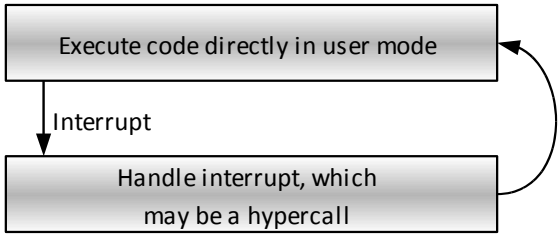

Figure 4: Main loop for paravirtualization

describing the porting process shows how a simple microkernel operating system can be ported to Xen.

Another example of paravirtualization is User Mode Linux. This is a modified version of the Linux kernel which is capable of running as a user-mode application inside Linux.

Paravirtualization can be expected to be very fast, since all code can be executed without runtime translation. The only overhead is having to use hypercalls instead of accessing hardware directly. This may affect the performance slightly, but this is unavoidable since direct hardware access is unacceptable in a virtual machine.

With paravirtualization, code is not translated. This means that one must be careful not to allow it any access to the physical machine. Since all of this code runs in user mode, the only way out should be calling the host kernel (at least, if the host kernel properly sets up permissions). This should be prevented, as it might allow virtual machines more access than they are entitled to.

If direct access to the host kernel is prevented, then the security of a paravirtualizing VMM relies on the ability of the host operating system to securely isolate applications. If no host operating system is installed, such protection should be provided by the kernel included in the VMM. In both cases the robustness can be expected to be at the same level as the protection between applications delivered by the host operating system. Paravirtualization lacks the double security provided by binary translation.

For the relationship with the host operating system, it has been shown that a paravirtualization VMM can run directly on the system (like Xen) or as an unprivileged application inside a host operating system (like User Mode Linux).

With paravirtualization, each guest operating system has to be ported specifically. This takes some effort and can only be done if the source code is available. A closed-source operating system can only be used if it's owner ports it. Hence portability is rather bad for paravirtualization.

Previrtualization

LeVasseur et al. [16] suggest a modified version of paravirtualization, which they call previrtualization. In this case sensitive instructions are replaced by the compiler, drastically reducing the effort needed to perform the port. This can only be done if the hypercall interface is sufficiently similar to the interface sensitive instructions provide to the CPU,

I am not aware of current real-world usage of previrtualization. The University of Karlsruhe is currently developing experimental VMMs which apply previrtualization. Marzipan runs without an operating system and is based on the L4Ka Pistachio microkernel developed at the same university. It is used for research. BurnNT runs as an application on Windows XP. This program is currently in an early stage of development, being able to start the Linux kernel, but not supporting user applications running on it yet.

Previrtualization is very similar to paravirtualization, but some differences in characteristics can be expected.

The restriction that the hypercall interface be similar to the interface provided by the CPU may result in lower performance than paravirtualization, where for example multiple actions may be merged in a single hypercall. Performance can therefore be expected to be slightly

worse than for paravirtualization. Because code is not translated at runtime, a significant performance benefit over binary translation can still be expected.

Portability is significantly better than for paravirtualization, since it should be possible to port open source guest operating systems with little effort. Closed-source operating systems still require that owner of the operating system cooperate with the porting effort.

Operating system level partitioning

I will discuss operating system level partitioning only briefly, as it is not really a virtualization technique as defined by Popek and Goldberg. It is still an interesting technique because it is much more lightweight than the solutions discussed before.

In case of partitioning, virtualization support is provided by the operating system. The applications running on the operating system are partitioned in several isolated groups. The implementation for system calls make sure that these groups cannot interact in ways different than interaction between separate machines. This means that they do not share the same file system and that one who has root access to a single partition may not have this privilege for other partitions or the machine itself. Kamp and Watson [12] describe partitioning as implemented by FreeBSD, where the feature is called "jails."

Solaris 10 implements this feature and names it "Zones." Partitioning on Solaris is highly configurable, allowing partitions to share some read-only directories to avoid duplication of data. OpenVZ and Linux V-Server are patches for the Linux kernel which allow one to use partitioning on Linux.

Performance for partitioning is very good. Both applications and the kernel are executed natively and hypercalls are not needed. The approach is very lightweight because there is no need (and indeed, no possibility) to run multiple operating systems. Overhead is limited to some extra security checks in the implementations for system calls.

The approach is as robust as the kernel itself, since no separate VMM is used. The relationship with the operating system is inherently that the VMM is part of the kernel.

Portability of the partitioning solution is bad, since different partitions are all running the same operating system as the host.

Application virtual machine

Yet another virtualization approach which is interesting despite not satisfying the Popek and Goldberg definition is the use of application virtual machines.

An application virtual machine does not duplicate a real machine, but instead exposes an instruction set architecture especially designed for virtualization. Such an instruction set is designed for running applications, not operating systems. A simple VMM is in between the guest application and the host operating system. This VMM typically compiles the guest instructions and the caches the result. This is similar to dynamic binary translation, but in this case the job is simplified by a good choice of virtual instruction set architecture. This allows compiling larger chunks of code at once and is typically called "just in time compilation" in this context.

Gough [6] discusses application virtual machines and describes and compares two important examples: the Java Virtual Machine and Microsoft .NET. Both use an instruction set which is stack based and inherently object oriented. The main difference between the two is that Java is more oriented towards emulation, while .NET was designed with just in time compilation in mind. In practice both use just in time compilation on common operating systems running on IA-32. On other platforms only emulation may be available.

Because for application virtual machines the instruction set architecture is designed with virtualization in mind, one can expect performance which is better than dynamic translation.

An application virtual machine can be very robust. The instruction set architecture is normally designed with security in mind. One also has the same kind of double security that there was in case of dynamic binary translation: all code passes through a translator, so no code goes through unchecked.

Application virtual machines do not typically offer the level of isolation that normal virtual machines do. Still the VMM is in complete control of all interaction between the application and the operating system and it can selectively block or alter interaction with the system, if so configured by the user. This may be even more useful than total isolation in some situations.

The relationship with the operating system is inherently that of an unprivileged application.

An application virtual machine can only run guests using the specially defined instruction set. This means that portability is very bad.

It deserves mention that application virtual machines such as Java have very good host portability. One can execute an application on different operating systems and architectures without recompilation as long as one has the proper version of the VMM.

Hardware supported

Recently Intel has introduced instruction set extensions which allow hardware supported virtual machines. This new technology is called Virtual Machine Extensions (VMX). An overview of this technology is presented by Uhlig et al [23] and complete documentation can be found in [8].

Intel introduces a distinction between VMX root and nonroot modes. When the CPU runs in root mode, it can set up an environment for the nonroot mode and switch. When it runs in nonroot mode, sensitive instructions which do not trap and interrupts will cause a VM exit. This saves the complete CPU state and restores the root mode situation. It is possible for the root to prevent certain instructions from causing a VM exit for performance reasons.

Both modes are just like the original situation, including four security rings. In effect the new chips therefore have 8 different security rings. This allows a guest operating system running in nonroot mode to use ring 0, while the VMM stays in control. Therefore these operating systems can run entirely without modification. Figure 5 shows this approach.

Now one can consider ring 0 of the VMX root mode to be supervisor mode, the VMX nonroot mode to be user mode, and a VM exit to be a trap. The VMX extensions eliminate exactly the problem that made IA-32 not satisfy the conditions for the Popek and Goldberg theorem. Ring 1 through 3 of the VMX root mode still do not satisfy these conditions, but they now can be avoided in the VMM.

The performance of hardware supported virtualization depends heavily on the implementation of the chip. It is likely that performance is better than for binary translation, but for paravirtualization it is hard to tell. This depends on which is more expensive: a hypercall or a VM exit. The state data which needs to be updated on a VM exit is stored in a 4096-byte block of data, so more memory needs to be updated than for a hypercall. It is not unthinkable, however, that the implementation highly optimises this specific case.

A VMM which uses the hardware solution is likely to be very simple, and as such likely to be robust. Security would only be compromised if a bug causes guest code to execute in root mode. Such a severe bug should be relatively unlikely in a simple, well-tested VMM. As such, hardware virtualization can be considered to be very robust.

VMX instructions trap outside of ring 0. This means that kernel-mode support is needed to be able to use them.

Portability between guest operating systems is very good. Like for dynamic binary translation, it does not really matter what code executes on the Virtual Machine. Unlike binary translation, porting in such a way that guests can run on a different architecture is not possible.

2.3 - QEMU implementation

QEMU uses a binary dynamic translation approach designed with a focus on portability. This allows for the support of a number of guest architectures on a number of host platforms. It has three features that particularly increase portability: availability of a user-level memory management unit, code generation from C code fragments and (optionally) absence of kernel-level components. Each of these features will be discussed in turn.

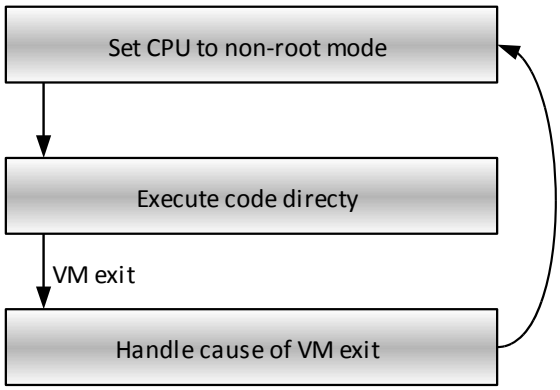

Figure 5: Main loop for hardware-supported virtualization

User-level memory management unit

QEMU can be configured to use either the hardware memory management unit (MMU) built into the CPU or to emulate one. To clarify the role of the MMU in dynamic binary translation, I first discuss the function of the MMU and then the difference between software and hardware solutions.

Programs running on modern CPUs in typical operating systems do not have access to all physical memory in the computer. Instead, each process has one or more virtual address spaces (segments) in which it can address memory more or less freely. These virtual address spaces provide a view of a part of the total virtual memory space, which may include physical RAM and memory-mapped IO ports as well as data swapped out to disk. This is typically achieved through two mechanisms: segmentation and paging. Segmentation allows division of the physical address space into a number of delimited virtual address spaces, while paging allows mappings where virtual memory is not backed by a contiguous area of physical RAM. Both additionally allow for access protection, restricting reads from and writes to specific segments and pages. The MMU is the part of the CPU responsible for performing these computations and checks.

To make matters concrete I discuss the x86 MMU, which supports both segmentation and paging. When an x86 instruction references memory it specifies a virtual address as well as a segment selector, the latter typically being implicit. Segmentation is handled before paging. The MMU checks whether the virtual address is valid for the segment—it must not be above the segment limit—and whether the reference is allowed for that segment—read-only segments, for example, may not be written to. Next it computes the linear address by adding the segment base to the virtual address. These linear addresses form a single address space which is affected only by paging. Pages are blocks of memory, each typically 4096 bytes large. To resolve a linear address to a physical address, the page it is in is looked up in the page table, which specifies the corresponding page in the physical address space. Pages may be missing, resulting in a page fault that allows the operating system to update the page table. The

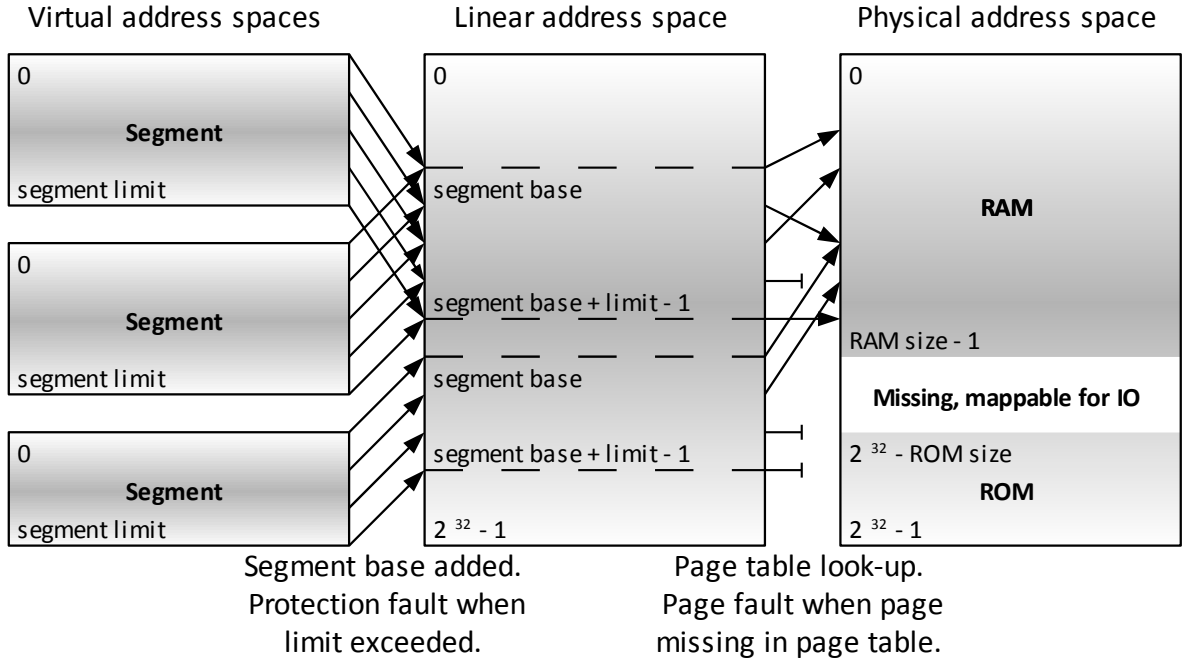

Figure 6: MMU operation and terminology on x86

physical address space mostly consists of addresses referring to RAM modules, but some addresses may be mapped to ROM chips or IO devices. An overview of the entire process and the terminology involved is provided in Figure 6.

If the guest CPU has an MMU, QEMU must emulate it. Segmentation is relatively easy to implement efficiently. If one assumes that segment mappings are not changed, segmentation is resolved by a simple addition with a constant known at translation time. This is the approach used by QEMU. This means that translated code must be flushed when segment mappings are changed, but as this is relatively rare and therefore has little performance impact. QEMU could check segment limits in the same manner, but does not do so (at least in version 0.8.2). This is a deviation from the x86 architecture and may cause problems with operating systems that rely on it. Fortunately, it is not much of a problem in practice and it keeps segmentation simple and fast.

Paging, in contrast, requires more processing. The page table is a large data structure, stored in the memory of the guest machine. While the correct segment could be determined during translation, most of the time addresses are computed at runtime so the correct page table entry can only be looked up at runtime. This requires several memory reads for each memory operation, which may slow down emulation significantly. The typical solution is to let the host MMU do some of the work. This can be done using `mmap`, `mprotect` and related POSIX calls, which allow user processes some control over the way pages in their data segment are mapped. Unfortunately these calls can only be implemented on host CPUs that have MMUs. Moreover some operating systems, including MINIX, do not implement these calls on any architecture. A fully hardware-based solution to paging would therefore be a major portability issue. Fortunately QEMU implements both a software and a hardware MMU, allowing all use of the memory mapping functions to be disabled, although at the cost of performance. If the software MMU is used, each memory reference is translated into a function call emulating the MMU. The performance hit is mitigated somewhat by caching page information in a translation look-aside buffer (TLB), which is similar to the mechanism used in hardware MMUs.

Code generation

Virtualization software based on dynamic binary translation speeds up emulation by generating host code that is functionally equivalent. Such a program has a collection of code fragments which emulate guest instructions, which are glued together to generate guest code. These code fragments must be very efficient because they are run all the time; since these fragments are typically very small, a single additional instruction may cause a substantial slow-down. This and the requirement that it must always be possible to copy these code fragments around and glue them together makes it attractive to write them in assembly language, so the programmer has complete control over what happens.

Unfortunately, using assembly greatly reduces portability as each processor architecture has its own assembly language. Suppose a hypervisor supports n host architectures and m guest architectures, all instructions for each guest architecture would need to be implemented separately for each host architecture. This means one would need a total of $n \cdot m$ code fragment databases to be able to support every guest on each host. Therefore, while use of hand-coded assembly provides a performance benefit, it makes supporting many host architectures as well as many guest architectures very hard.

QEMU takes a different approach. All code fragments needed for emulating guest instructions are coded in C and therefore the same code can be used on each host platform. To be able to copy code around and glue it together, QEMU uses a program that analyses the object file containing the compiled code fragments. By locating relocations and return instructions this program makes it possible to move the code around safely. To reduce the performance penalty incurred by using C rather than assembly, QEMU uses temporary variables and forces the compiler to store them in CPU registers. This reduces the number of memory operations, which would also be a major goal when using hand-coded assembly to implement the instructions. These aspects are discussed in turn.

Moving code around

Unfortunately, concatenating C functions at runtime is nontrivial. To determine the size of data structures at compile time, the C language offers the `sizeof` operator. No such operator exists for functions so that one cannot determine how many bytes to copy to get the entire function body. Even if it did, there would still be problems. Most forms of branching instructions—such as function calls as well as conditional and unconditional jumps—take arguments relative to the instruction pointer.

Figure 7 shows an example of a branching instruction with a relative address in x86 assembly. Assembly instructions and the addresses at which they are stored in memory are shown on the left-hand side. The corresponding C code has been included on the right-hand side to clarify what is happening here: the `get_answer` function calls the `get_double` function, the former starting at 0x10000 and the latter at 0x11000. When the compiler comes across the function call to `get_double`, it generates the `call` instruction. This instruction first pushes the current address on the stack—so that the called function knows where to return to - and then adds the specified value to the instruction pointer—causing a jump the specified function to occur. This means the specified value is an address relative to the instruction pointer. If one were to copy the `get_answer` function elsewhere to generate code, the destination will also change. When the `get_answer` function is copied to 0x12000 for example, it will expect to find `get_double` at address 0x13000 rather than 0x11000. When it jumps to this address, some random code is executed and the program will most likely malfunction and/or crash. This would not be much of an issue if one were to use assembly, as one could avoid instruction pointer-relative references. Even if no such instructions are available or if they are inefficient, one could at least know in advance where to find the references. This is unpredictable with C code, especially when compiling on different platforms or even different compiler versions.

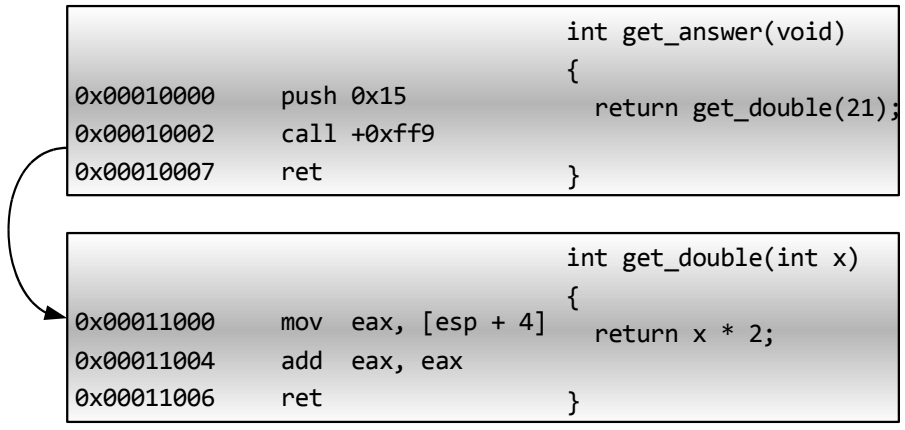

Figure 7: Function call with relative displacement on x86

Fortunately, compilers keep track of references to symbols. These are stored in a structure called the relocation table in the object files created by the compiler. This allows the linker to adjust these references once the addresses in the final executable are known. QEMU comes with a tool called Dyngen that can also read symbol tables and relocation tables from object files. It converts them into C header files, which are in turn used by the code that translates guest instructions into host executable code. How this affects the compilation process is shown in Figure 8.

First, the Dyngen tool is compiled and linked into an executable and the `op.c` file, which contains implementations of the guest instructions, is compiled into an object file `op.o`. Dyngen reads this object file. It recognizes code fragments because their names start with `op_`. For each of these code fragments it locates the relocations that apply and generates code to copy the code fragment and correct all relocations that apply to them. If the case shown in Figure 7 were such a code fragment, the code generated would look something like this:

```
memcpy(gen_code_ptr, (void *)((char *) &get_answer+0), 7);
*(uint32_t *)(gen_code_ptr + 3) = (long) (&get_double) -
  (long) gen_code_ptr + -7;
```

The first line copies the code of the function into the code generation buffer. Although the function is eight bytes long, it copies only seven. The last byte is the `ret` instruction, which returns to the code that called the function. This is detected by Dyngen and it is not copied to allow code fragments to be concatenated. The second line corrects the reference to `get_double`. Dyngen detected this reference, which consists of a four-byte address at the fourth byte of the function (`gen_code_ptr + 3`). The relative address is set to the address of the function being called (`&get_double`) which is corrected by subtracting the value of the instruction pointer of the instruction following the `call` instruction after copying (`gen_code_ptr + 7`).

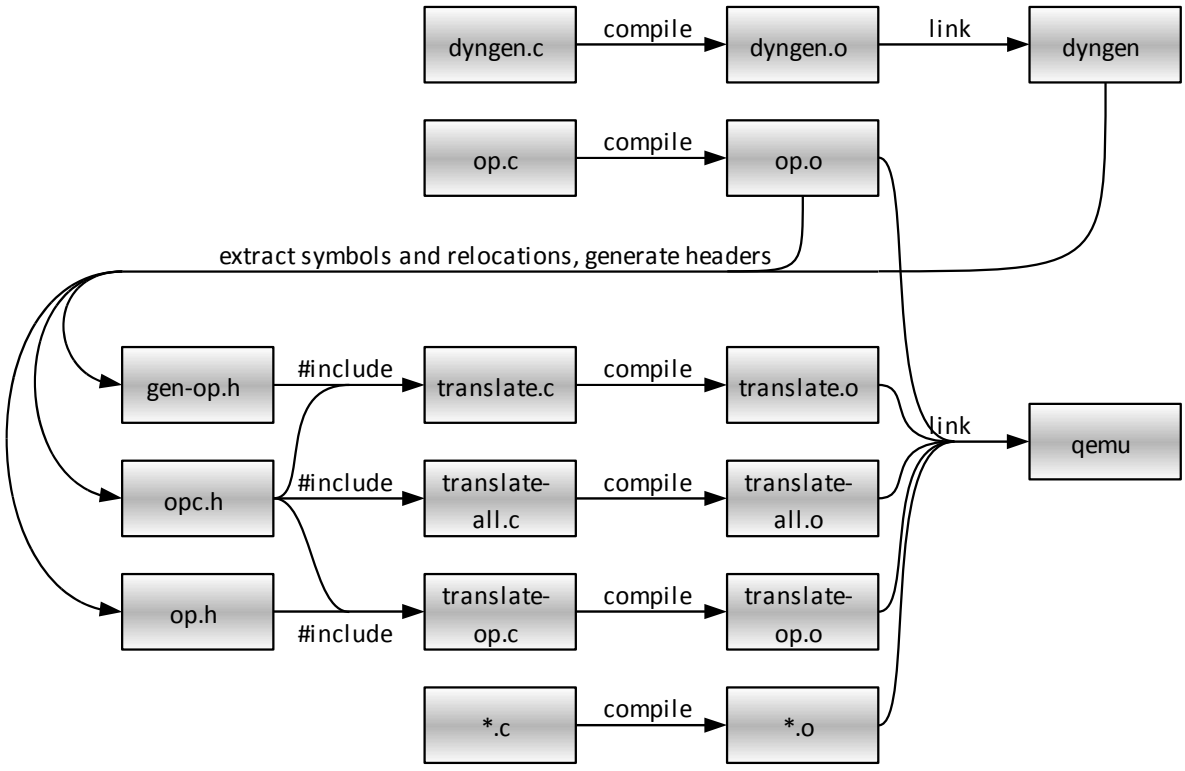

Figure 8: Steps in the compilation of QEMU using the Dyngen tool

The header files are included into the C source files involved in the translation. Code generation is a multi-step process, which will be explained in the next section; the C source files explicitly mentioned in Figure 8 roughly correspond with different steps in the translation process. Eventually, all C files are compiled and linked together so that the addresses of the code fragments in `op.o` used in the `translate*.c` files are filled in by the linker.

The approach to code generation used in QEMU has two implications for portability:

- Since no assembly code is used, porting between host processor architectures is made much easier;
- Since Dyngen has to be able to read the relocation and symbol tables from object files, porting to operating systems using different object file formats than already supported requires additions to the Dyngen program.

The former implication has no effect on porting to MINIX as current versions of MINIX only run on the x86 architecture, but the latter is relevant. In MINIX, object files are stored in a very simple format called "a.out." This is also the format of executables on the MINIX platform. This format was not supported originally, so I had to add support for reading it and applying its relocations to generated code.

Temporary registers and intermediate instructions

Another technique used by QEMU to generate code is worth mentioning: each guest instruction is split in a number of simpler intermediate instructions before being translated in host code. This process is demonstrated in Figure 9.

Like other dynamic binary translators, QEMU splits its input—the guest machine code—in basic blocks. Basic blocks do not contain any jumps so that they are normally executed in their entirety (that is, unless faults or interrupts occur). The basic block shown is a simple

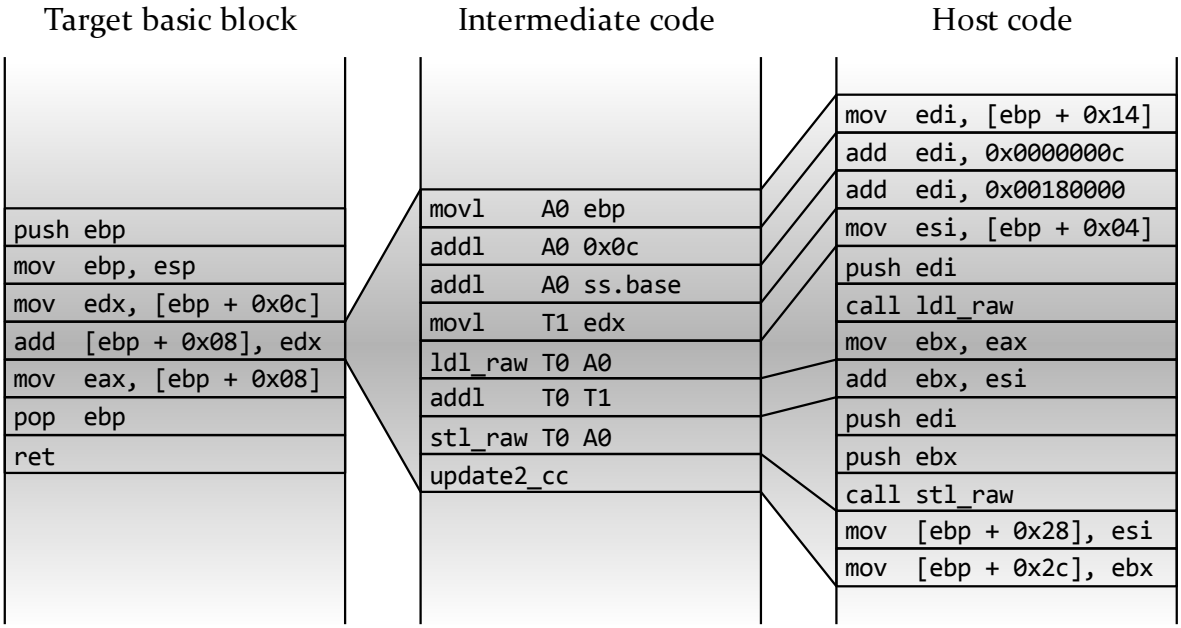

Figure 9: Translation of a x86 function using intermediate code

function adding its two arguments and returning the result, compiled by a nonoptimizing compiler for the x86 architecture. It can be represented in C as follows:

```
int addition(int x, int y)
{
    x += y;
    return x;
}
```

The most interesting part of this function is the addition, which is coded with only two x86 instructions. Besides adding, it involves more operations to load `x` and `y` from memory and to store the new value for `x`. Moreover the CPU does not know whether the result will be used in a conditional branch instruction, so it has check for several condition codes and store them in the flags register. I focus on the `add` instruction itself, which performs most of these actions. Before this instruction, `y` has been loaded into the `edx` register as x86 instructions can take at most one memory operand. For QEMU to emulate the `add` instruction, the following operations need to be performed:

- Compute the address of `x`, which is specified as `ebp + 0x08`, by first loading the value of the `ebp` register and then adding the constant, storing the result in a temporary register (named `A0` by QEMU);
- As `x` is stored on the stack (which is implied by the use of the `ebp` register), the base of the stack segment should be added to obtain the correct linear address. At this point the segment limit ought to also be checked to avoid illegal memory references, but QEMU does not do this currently;
- Load the second argument into a temporary register (named `T1` by QEMU) from the guest register `edx`;
- Load the first argument into a temporary register (named `T0` by QEMU) from memory;
- Perform the addition using the temporary registers;
- Store the result of the addition back into memory;
- Update the condition codes; this involves checking for each of the following: a zero result, a negative result, signed overflow, unsigned overflow, decimal overflow and parity of the low-order byte.

Most of these operations correspond with a single intermediate instruction, most of which are reused for many different guest instructions so that the total number of code fragments stays reasonably low. In particular, many guest instructions reference memory using load and store intermediate instruction such as `ldl_raw` and `stl_raw`. This not only keeps the code fragments simpler and fewer in number, but also makes translation easier as the addressing mode bytes can be translated separately from the guest instruction itself.

It should be noted that the use of intermediate instructions makes the emulation slightly slower than would be possible by directly implementing host instructions. This happens due to two reasons: two separate translation steps are needed and the compiler has less ability to optimize the implementation of instructions. The latter is especially important. A direct implementation of the entire `add` opcode would allow the compiler to allocate registers more efficiently than is possibly when small code fragments are glued together.

Fortunately, the GNU C compiler allows one to statically allocate registers to global variables. By allocating host registers for temporary registers and—if possible—for guest

registers, QEMU is able to prevent some of the overhead caused by the use of intermediate instructions. This works especially well on RISC architectures, which tend to have many registers. Since QEMU uses only three temporary registers—`T0` and `T1` for values and `A0` for addresses—they can be mapped to host registers even on architectures which have very few general purpose registers, such as the x86 architecture.

The right-hand side of Figure 9 shows the host code that is produced for the `add` instruction on an x86 host. Due to the use of host registers for temporary registers, most intermediate instructions are translated into a single host instruction. Memory loads and stores are more complex and are handled in separate functions.

As a further optimization, evaluation of condition codes is deferred to avoid unnecessary computations. The `update2_cc` instruction in the diagram only stores the arguments of the guest instruction for later use. If it can be determined in advance that none of the flags resulting from the `add` are used, even this step is eliminated. This is not the case here, as the caller of the function may use the flags after it returns. The ability to determine in advance whether the flags register may be used is yet another advantage of the use of intermediate instructions.

No kernel-level components

Modern CPUs are generally capable of running code at several privilege levels, allowing the execution of the more sensitive instructions only for the most privileged code. For example, the x86 architecture has four privilege levels called "rings." These rings are numbered zero to three. Ring zero allows complete control over the CPU, while ring three disallows the use of most system instructions. This is done to keep the operating system in control; if an application were able to execute system instructions it could bypass the protection mechanisms provided by the operating system. Other architectures have similar protection mechanisms, so I will use more generic terms in the remainder of this text. The most privileged code—called ring zero on x86—will be referred to as kernel-level code. This is where the operating system kernel typically runs. All other code—rings one, two and three on x86—will be referred to as user-level code. This typically includes applications started by the user but may also include some drivers that do not need to use system instructions. This is especially common in microkernel operating systems such as MINIX, where the kernel size is reduced by moving drivers out of the kernel into user space.

For the purpose of virtualization, the division between privileged code and less privileged code is relevant for both the guest machine and the host machine. In a certain situation this separation can be used to greatly speed up emulation: if the guest and host architectures are the same, the guest is executing user code and the hypervisor is capable of executing in kernel mode. In this case the hypervisor is capable of setting up an environment in which user code can be run directly, without any translation. This approach is widely used together with binary dynamic translation to speed up the execution of user code dramatically while still being able to run kernel code safely.

The major drawback of direct code execution is that it requires the hypervisor to run in kernel mode to allow it to set up the proper environment for guest user code to run in. The guest must have the entire address space at its disposal, as static code analysis cannot reveal in advance which memory locations will be referenced. Therefore a hypervisor running as a user process cannot share its virtual address space with a virtual machine. The need to run in

kernel mode reduces portability. Kernel mode features can be accessed only in assembly, which is hard to port between processor architectures. Moreover, the way in which kernel modules are installed is not standardized between operating systems. Loading kernel modules also threatens the stability of the host operating system if these modules may be buggy or untrustworthy.

QEMU supports direct code execution, but does so using a separate and optional kernel module called KQEMU. This kernel module creates a device `/dev/kqemu` (or `\\.\kqemu` in Windows). Using the `ioctl` function this device can be used to set up an environment in which guest code can run directly. Due to this approach, QEMU and KQEMU are very loosely coupled and KQEMU support can be disabled completely, both at compile time and at runtime. In this case QEMU performs dynamic translation for all code. This decreases performance, but increases portability and stability of the host operating system.

For the MINIX port KQEMU support is always disabled. QEMU allocates memory for the virtual machine in shared memory if KQEMU is used. As MINIX does not support shared memory, this approach cannot be used in MINIX. KQEMU runs in kernel mode, so it should be possible to circumvent this by directly addressing the memory in the address space of QEMU. Due to time constraints, I chose not to implement this for now.

It should be noted that the way KQEMU works does not fit well with the MINIX ideal of moving code out of kernel space. When running a virtual machine KQEMU takes control of the entire system by changing control registers and installing a new interrupt table. This effectively cuts the operating system out of the loop. The operating system is restored when the next interrupt occurs. Due to this approach KQEMU can ignore restrictions imposed by the operating system and a single bug in KQEMU might bring down the entire system. As an alternative, system calls could be implemented to allow a process to do the following:

- Create a new address space to contain the virtual machine;
- To control its memory mappings to be able to reproduce the environment the virtual machine runs in, including the ability to change both page and segment mappings;
- To access its memory efficiently;
- To receive signals sent to it to be able to detect error conditions.

In part, these features are already present. The `fork` system call allows one to create a new address space and `ptrace` allows one to receive signals for another process. Shared memory and the ability to change page mappings will probably become available when virtual memory is implemented in MINIX. To change segment mappings for another process, a privileged system call will be needed. When all of these features are available, it should be possible to create an efficient alternative for KQEMU that does keeps the operating system in control.

3 - Issues encountered and changes made

3.1 - General

Especially when starting to work a porting project, one generally has little knowledge of the way the program to be ported has been implemented. In the case of QEMU, there was some documentation on the website, but most of this is about how to use QEMU rather than about how QEMU has been implemented. Therefore, my knowledge about QEMU was mostly general knowledge about the dynamic binary translation approach to virtualization. To get started, I used a trial-and-error approach consisting of simply fixing problems whenever they occur. This causes the process of porting a program to consist of the following steps:

- Get the configuration script to run;
- Get the all source files to compile;
- Create missing functions and other declarations to allow the compiled source files to be linked together into executable files;
- Fix problems that prevent the program for running;
- Test whether all important functionality is still present in the port and repair functionality wherever needed and possible.

Before starting the first step, a number of choices has to be made concerning the tools that are to be used. In this case, the choice of compiler was particularly important as QEMU is exceptionally picky about compilers. The standard compiler on the MINIX platform is the Amsterdam Compiler Kit (ACK), which comes with the operating system and is used to compile it. All other things being equal, it would have been preferable to use this compiler as that would keep down the number of dependencies. Unfortunately however, the QEMU source code is specifically aimed at the GNU compiler collection (GCC). The syntax is not pure ANSI C, but relies on a number of GCC additions. Moreover, for code generation it depends on the exact compiler output to such a degree that only GCC versions 3.3 and 3.4 are supported; neither older nor newer versions can be used to compile QEMU version 0.8.2. The degree of dependence of QEMU on compiler optimization is shown even further by the fact that using an optimization level other than the one specified by the `-O2` compiler switch or disabling inlining to make debugging easier is also not possible. After I found these problems, I decided to use GCC as porting QEMU to ACK would require making major changes.

After having chosen the correct compiler, the next goal was to create a script that would configure QEMU for compilation on MINIX and then compile it. This script is called `build.minix`. The `configure` shell script turned out to be incompatible with Ash, the default shell used in MINIX. I have chosen to use the Korn Shell (`ksh`), which is capable of running the script and is available as a MINIX package. This choice is arbitrary as most other shells available on MINIX would also do.

Once the `build.minix` script was there to run the configure script and compile QEMU using GCC, I simply started fixing compilation errors one by one. The exact changes I made will be discussed further on in the chapter. One finding was that MINIX does not implement all functions needed by QEMU. Most importantly, the `setitimer` function used to generate

regular alarm signals, functions used for virtual memory management and functions used for floating point arithmetic were missing. This was fixed by providing headers so the compilation would succeed even though the functions were not implemented. Using this method it was possible to get all source files to compile reasonably quickly, although eventually some functions were found to be missing by the linker.

In the next step, I needed to fix the missing library functions, either by creating replacements for them or by avoiding their use. In general, I avoided using the virtual memory functions and implemented the other functionality. In this phase, I attempted to reproduce the functionality of the `setitimer` function in user space, which would avoid having to make changes to MINIX, but its performance turned out to be too bad to be useful in practice. This means that unfortunately, getting QEMU to run properly on MINIX requires changes to the operating system. It should be noted that the changes that were made are not QEMU-specific and that additions conform to the POSIX standard, which means that they will also be useful for porting other software. I also attempted to keep changes to MINIX down to the minimum possible by implementing missing library functions in user space wherever possible.

The most important step in getting the compiled program to run was changing the code generation program so that it can read MINIX object files and make the proper corrections when copying code, as has been explained in section 2.3. Before fixing this, QEMU would always crash with `SIGILL` or `SIGSEGV` signals or hang and was therefore useless even though it did compile.

By the time QEMU could run mostly without crashes, testing showed that some functionalities present on other operating systems had been lost in MINIX. Some had been disabled intentionally, such as graphics support, while others did not work even though they had not been disabled, such as networking, reading host CD-ROMs and reading large disks. Graphics had been disabled because QEMU uses the Simple DirectMedia Layer (SDL) library to render its graphics and this library had not yet been ported to MINIX. Although QEMU can be built and run without it, it is of little use when it produces only text output. Therefore, the SDL library had to be ported before QEMU would run well. The problems with networking and CD-ROMs just mentioned eventually turned out to be caused by using the `ioctl` function without error checks. Large disks could not be read because the `lseek` function is implemented with a 32-bit offset, which means that only the first 4 GB of each file can be read. Although this is not a problem for regular files—the MINIX 3 file system cannot be larger than 4 GB—it is an issue if one wants to read another partition that does exceed this size.

I will discuss the changes I needed to make to MINIX in more detail in the next section. After that, I elaborate on the changes made to QEMU itself. In this section, I will also provide some general recommendations for porting software to MINIX based on my experience with QEMU. Not all changes I made were needed for QEMU to run correctly, some also added new functionality that is convenient for running it on MINIX. These additions are discussed in a separate section. After this, I provide information about how testing and debugging was done. At the end of the chapter, I will discuss how suitable MINIX is for running programs such as QEMU and how porting software to MINIX can be made easier.

3.2 - Changes made to MINIX

In this section, I describe the ways in which MINIX 3.1.2a has been changed to allow QEMU to run on it. Each subsection describes why a change was needed, how the change has been implemented and why this particular approach was chosen.

Addition of the `setitimer` function

Usage of `setitimer`

Some hardware emulated by QEMU needs to perform actions at regular time intervals or after some delay. Most importantly, QEMU emulates an Intel 8254 programmable interrupt timer (PIT) chip, which generates timer interrupts at a frequency set by the operating system running on the virtual machine. MINIX, for example, uses timer interrupts to determine when to switch processes and to keep track of the time. It programs the PIT to send a timer interrupt 60 times per second; this frequency can be changed by altering the `HZ` constant defined in `/usr/include/minix/const.h` and then recompiling and rebooting. Other operating systems may set the timer to much higher frequencies and may be able to change the timer frequency without recompiling or rebooting.

To be able to interrupt the virtual machine at the right moment, QEMU uses the `setitimer` function to request that it receive a `SIGALRM` signal every millisecond. By default, the most similar function present in MINIX is the `alarm` system call. This call, however, waits for at least a second between `SIGALRM` signals. This shows that something needs to be done to be able to properly emulate the Intel 8254 chip and other time-dependent hardware.

Possible solutions

Basically there are three possible solutions:

1. Avoid the need to use `setitimer`;
2. Implement the `setitimer` function in user space;
3. Implement the `setitimer` function in MINIX itself.

The first solution can be implemented by regularly checking the current time and calling the `SIGALRM` signal handler directly whenever some amount of time has passed. This is reasonably easy to implement in QEMU, as it is possible to insert such checks in the code generated for the virtual machine. The generated code is generally executing most of the time, so one can expect the timers to be checked often enough. The main problem with this approach is that one has to insert many checks in the generated code, as one should avoid a situation in which basic blocks are chained together into a loop that does not contain any timer checks. This would cause the virtual machine as well as QEMU itself to hang. At the time code is generated it is not known whether it will be part of such a loop, so a timer check has to be inserted into most basic blocks. Summarizing, the first solution allows one to produce timer interrupts at a high rate without modifications to MINIX, but can be expected to substantially slow down emulation.

The second solution involves creating a new process to send `SIGALRM` signals to QEMU's main process. This second process can be expressed in pseudo code as follows:

```
while (parent process is running)
{
    wait for specified amount of time;
    send SIGALRM to parent process;
}
```

The `select` system call allows the second process to wait for a specified amount of time. It blocks the second process, so that it does not take much CPU time from QEMU. As the wait time is passed to `select` in a `struct timeval`, in theory very small waits should be possible Unfortunately, MINIX only checks whether the wait period has been completed when it receives a timer interrupt at the time the quantum for the current process has expired. As was mentioned before, MINIX only requests 60 timer interrupts per second by default, so emulation does not run as well as it would on host operating systems which set the timer to higher frequencies. Moreover, it may take even longer before the second process is scheduled if is first has to wait for the 8-tick quantum of QEMU to run out. Although all timer interrupts are still delivered to the virtual machine, they arrive in large batches rather than at regular intervals if the frequency requested by the guest operating system is too high. Moreover, switching between QEMU itself and the helper process can be expected to slow down emulation. Summarizing, the second solution does not require modifications to MINIX or to the generated code, but the rate at which timer interrupts can be generated is limited and performance can be expected to be lower than with a kernel space implementation of `setitimer`.

The third approach involves adding the `setitimer` system call to the process manager. Its implementation is highly similar to the `alarm` system call and the two can share code. Again, the maximum signal frequency is limited by the number of timer interrupts MINIX receives. I preferred not to make any changes to MINIX, since the need to use a modified MINIX makes installation more difficult and if MINIX needs to be changed I cannot really claim to have ported QEMU to the original MINIX. However, adding a `setitimer` system call does solve the problem without adding overhead for polling or extra context switches. Moreover, `setitimer` is a standardized function specified as part of the optional XSI group of functions in POSIX and as a mandatory function in the version 3 of the Single UNIX Specification [22]; this means that adding it to MINIX is likely to also benefit other ports.

Implementation of `setitimer`

I have attempted each of the three solutions listed here and decided to use the third one as the loss of performance caused by the other approaches is considered to be unacceptable. In particular, when using the second approach QEMU was so slow that it was barely useful. This may have to do with the large number of context switches caused by sending timer signals from another process. The first approach has the advantage that it can produce clock interrupts at any rate, which improves the quality of the emulation. Its implementation has been kept in the source code, but it is only used if the `CONFIG_DETERMINISTIC` precompiler symbol has been defined. It allows one to run a virtual machine in a deterministic way; this feature will be discussed in detail in section 3.4.

My implementation of the `setitimer` call was obtained from David van Moolenbroek, a PhD student working on MINIX. This essentially replaces the `alarm` system call with `setitimer` and then implements the former using the latter; this is possible because the functionality of `alarm` is a subset of that of `setitimer`. This allows for an efficient implementation that does not require many changes and that does not use any additional timers. Unfortunately, while

testing QEMU with David's `setitimer` implementation, I found that due to this approach some library functions using `alarm` system call and then restoring the original value caused the `setitimer` settings to be lost. This was fixed by making these library functions use `setitimer` directly rather than have them call the `alarm` function.

Timer resolution

As mentioned before, one issue with the `setitimer` function is that it cannot generate signals at a higher rate than specified by the `HZ` constant. This is not much of a problem when running guest operating systems that set the timer to a low frequency—such as MINIX itself—but it turned out to be a problem when running operating systems that request more ticks and are sensitive to timing. For example, while testing Windows 98 Second Edition I found that it can only boot in safe mode. Using debug output, I found that Windows 98 normally sets the clock to 200 Hz and that it is occasionally raised to 1000 Hz for short periods of time; this means that regular delivery of clock interrupts is clearly impossible at the default setting of 60 Hz. Unfortunately, on my test machine, increasing the clock frequency to match that on the guest does not help in this case. At 1000 Hz, QEMU spends all its time generating clock interrupts and barely any useful processing happens. Windows 98 SE can run if MINIX uses a 150 Hz timer. If one avoids alarm signals by configuring QEMU to run deterministically Windows 98 does also run. See section 3.4 for more information on running deterministically. I discuss my experiences with several operating systems in more detail in section 3.6.

The problem with the low resolution of the `setitimer` function on MINIX can be solved in three ways. These are the possible solutions ordered by increasing complexity:

- Increase the value of the `HZ` constant;
- Allow software to change the timer frequency while MINIX is running;
- Schedule clock ticks based on need rather than periodically.

The first solution is the easiest one and this is what I have done to test guest operating systems that need many ticks. Unfortunately, when using this solution one has to recompile and reboot MINIX every time the clock frequency needs to be changed. Simply picking a very high clock frequency is not desirable as it reduces performance, since servicing clock interrupts costs CPU time that would otherwise be spent executing user code. The extent to which high clock frequencies hamper performance has been tested and is described at length in section 5.2. My tests show that, when compared to the hypothetical situation that the clock is deactivated, each increase in frequency by one clock tick per second reduces performance by $-7.29 \cdot 10^{-4}$ %/Hz. This may seem to be a very small figure, but it means that general performance degradation is approximately 0.75% at the 1024 Hz frequency preferred by QEMU. When the system is idle, using a high clock frequency also increases power usage and, when running as a guest operating system on a virtual machine, slows down the host computer.

The second solution is used on standard Linux. Linux allows programs to set the clock frequency by using the `ioctl` function with the `RTC_IRQP_SET` request code on the `/dev/rtc` device. This way, extra timer interrupts are only generated when some program needs them and there is no need for recompilation or rebooting. This feature is used by QEMU on Linux to set the clock frequency to 1024 Hz. Implementing this in MINIX is reasonably simple: a device has to be implemented to allow processes to set the clock frequency, the frequency has to be changed from a constant into a variable and some of the timekeeping code will need to

be changed to deal with heterogeneous cycle lengths. To perform the performance test discussed earlier, I have implemented this in practice as a `/dev/clock` device; this implementation is discussed in more detail together with the test.

The third solution would involve larger changes to the MINIX kernel than the others, but would also provide the largest gain. By avoiding periodic clock cycles and instead scheduling clock ticks based on need, time-outs in functions such as `setitimer` are as accurate as possible, while idle power consumption is reduced. This solution, a tickless kernel, has been implemented as a patch to Linux and has been merged into the current release version on the x86 architecture [7].

I have chosen not to address the issue of the minimal `setitimer` interval being too large for now, as each of the solutions has some problems:

- Increasing the `HZ` constant negatively affects operating system performance and the new value would have to be chosen arbitrarily.
- My implementation of a device that allows processes to set the clock speed is too experimental for me to recommend its use. In particular, it uses a very high value for the HZ constant which reduces maximum system uptime because of the 32-bit signed `realtime` variable in `kernel/clock.c` overflowing. This happens after $\frac{2^{31}}{60\,\mathrm{Hz}} \approx 414\,\mathrm{days}$ for standard MINIX and $\frac{2^{31}}{2400\,\mathrm{Hz}} \approx 10.4\,\mathrm{days}$ for my implementation. It may also cause problems with some drivers that use ticks for timing while not using the `HZ` constant. Moreover, only divisors of the `HZ` constant are allowed as clock frequencies with my implementation. A more durable solution would have to be implemented for general usage and this is outside the scope of this master's project.
- Implementing a tickless kernel takes more effort than a settable frequency, so this is also outside the scope of this master's project, but might be a useful addition to MINIX in the future. It would provide more accurate timing and would use less power, which would be desirable considering MINIX' goal of being suitable for embedded computers.

As most operating systems will run as guests even with a 60 Hz clock cycle, this change is not considered essential for QEMU. Any user that still wants to be able to run such operating systems can change the `HZ` constant and recompile MINIX.

Implementation of the `pread64` and `pwrite64` functions

By default, MINIX cannot read from or write at file offsets above 4 GB. This is due to fact that the file system driver uses a 32-bit integer to store the current file position. Operations that cause the file position to overflow are not permitted. This is not a problem for regular files as the file system itself cannot grow larger than 4 GB, but it is a serious limitation when accessing large block devices such as the hard disk or its partitions. If this problem is not fixed, physical partitions over 4 GB cannot be used by virtual machines.

One way around the problem, which is used in some of the utilities included with MINIX, is to use the `DIOCSETP` `ioctl` code to change the partition table in the memory of the hard disk driver. Although this does work, it is not satisfactory for use with QEMU. The most important problem is that it consists of a number of steps and that in between these steps other processes' views of the disk are changed, leading to a race condition if multiple processes access the disk simultaneously.

As an illustration, the following code sample shows the workaround being used to read `size` bytes of data at byte `offset` from an open disk identified by the file descriptor `fd`:

```
/* step 1: read old partition table */
ioctl(fd, DIOCGETP, &partition_entry);

/* step 2: adjust partition table base and size */
partition_entry_temp =  partition_entry;
partition_entry_temp.base = add64(partition_entry_temp.base, offset);
partition_entry_temp.size = sub64(partition_entry_temp.size, offset);
ioctl(fd, DIOCSETP, &partition_entry_temp);

/* step 3: seek and read/write */
lseek(fd, 0, SEEK_SET);
read(fd, buffer, sizeof(buffer));

/* step 4: restore partition table */
ioctl(fd, DIOCSETP, &partition_entry);
```

Now suppose that two instances of QEMU are running, both using the physical hard disk `/dev/c0d0` as their primary hard disk. This is quite possible in practice, for example if MINIX 3 is running on a triple boot system and both of the other operating systems are run simultaneously with MINIX 3 using QEMU. Now suppose that both instances read the disk, the first instance at `offset1` and the second at `offset2`. Table 1 shows a possible schedule for the operations shown in the source code example, indicating what can go wrong. Instance 2 stores the wrong offset because it has just been modified by instance 1. This results in it restoring the partition base incorrectly at the end, so all disk reads will return incorrect results until the driver is reset. Moreover the processes interfere with each other by one process changing the partition base before the other gets an opportunity to perform the read. As a result, both reads also return incorrect data. The schedule shown in Table 1 is a worst case scenario which will not happen often in practice, but does show why having such a race condition in QEMU is unacceptable.

To solve the race condition without making large changes in MINIX, I added two `ioctl` functions to the disk driver. This allows it to bypass the file system driver, which unlike the disk driver uses only 32 bits to store file offsets but allows `ioctl` calls to be passed on to the driver. The new `ioctl` functions are called `DIOCREAD64` and `DIOCWRITE64` and, like `DIOCGETP` and `DIOCSETP`, they are implemented in `libdriver`, which is a basic driver framework used by the disk drivers. Like the original workaround, they are implemented by shifting the partition base, performing a read or write operation and restoring the partition base again. Unlike with the workaround, it is not possible to interrupt this sequence halfway because the driver can handle only one request at a time. Additional requests from other processes are blocked until the `ioctl` call returns. This is more secure and increases performance because fewer context switches are needed.

Instance 1	Instance 2	Action	Base afterwards	Error
			b	
Step 1		Store base: b	b	
Step 2		Set base: b + offset1	b + offset1	
	Step 1	Store base: b + offset 1	b + offset1	
	Step 2	Set base: b + offset1 + offset2	b + offset1 + offset2	
Step 3		Read data	b + offset1 + offset2	Read data at offset1 + offset2 rather than offset1
Step 4		Restore base: b	b	
	Step 3	Read data	b	Read data at 0 rather than offset2
	Step 4	Restore base: b + offset 1	b + offset1	Base restored to incorrect value

Table 1: Possible schedule when two instances of QEMU read the same disk

Besides adding the `ioctl` calls, I also added two library functions. These functions, `pread64` and `pwrite64`, are also provided by Linux. Both the signature and functionality of the functions are as similar as possible to their counterparts in Linux, but two changes needed to be made. First, the `offset` parameter uses the MINIX-specific `u64_t` type rather than the compiler built-in `long long` type because the latter is not available on ACK. The other difference is that the file position is not changed if it would not fit into a 32-bit integer. These functions use `lseek` and `read` or `write` when possible—when the both the offset and the file pointer after reading fit within a 32-bit integer—and use the new `ioctl` functions otherwise. This makes sure that the functions work for both regular files and for devices that support the new `ioctl` functions.

Signal handling bug

MINIX 3.1.2a has been the target platform for the port of QEMU. Although even at the time I started working on the port a newer release of MINIX was available, the newer release was not stable. I decided that it would be easier to test on a platform that would not itself introduce bugs, which is why I used the stable release instead.

Unfortunately, I noticed some odd behaviour in QEMU that I could not explain by looking at the source code or even at the assembly code. It turned out that a bug in MINIX caused the contents of the flags CPU register to be overwritten whenever a signal occurred. If this happens between a comparison and a conditional jump, it may cause the jump to be taken even though it should not be or vice versa. It was very hard to find the cause of the problem, as one generally assumes that this kind of low level operating system functionality has been tested very thoroughly and should be relatively bug-free. However, as signals are relatively rare in most programs and the flags register being overwritten does not have a noticeable effect in most cases, this bug has very little effect on most programs. Due to the large number of signals QEMU receives from the `setitimer` function, the problem did occasionally crash QEMU at unpredictable moments. Fortunately, the bug had been found before and had been fixed in the version of MINIX in the source control; I would like to thank Jens de Smit for pointing this out to me in a newsgroup post [14].

The fixed version of the `do_sigreturn` function from the MINIX source code on SubVersion has been included to allow QEMU to run stably on MINIX 3.1.2a. It should be noted that this fix can be omitted if QEMU is being used on future releases of MINIX.

Use of the `select` function with the `/dev/eth` device

MINIX, like other POSIX-compliant operating systems, provides the `select` system call to wait for multiple file descriptors to become available for reading or writing. This capability is used in QEMU to be able to pass incoming input to and process output from the virtual machine without blocking. It is also used by the `qemu-vswitch` program I created to increase QEMU network functionality on MINIX; this program is described in detail in section 3.4. Depending on the networking settings, these programs may pass handles for the `/dev/eth` device to `select` so as to wait for incoming raw network packets. Unfortunately, in MINIX the `select` call fails if such handles are specified, as this functionality is not implemented for them. To be able to wait for network input on MINIX, I added this functionality to the MINIX network stack.

Servers implementing MINIX devices are sent `DEV_SELECT` messages whenever `select` is called for such a device. The `/dev/eth` device is implemented by the `inet` driver which implements the network stack. Based on the minor device number, it forwards such requests to the `eth_select` function. By default this function does nothing but print a message saying that it is not implemented. Fortunately, `select` functionality is implemented by the very similar `/dev/udp` device that is also part of the `inet` server. All datagram-oriented services—ethernet, IP and UDP—are implemented in very similar ways, the difference being mostly the way headers are parsed. For this reason I decided to copy the implementation of `udp_select` to `eth_select`, changing not much more than just the names of the data structures used and their members. This fixed the problem and the patched MINIX now accepts requests to wait for incoming ethernet packets.

3.3 - Porting QEMU

In this section, I discuss all changes I made to QEMU: how the need for change was found, why change was needed and what was changed. Many issues I encountered are likely to occur when porting other software as well, so I aim for this section to serve as a tutorial for porting software to MINIX. In particular, the first sub-section discusses a number of general caveats that most people porting software to MINIX are likely encounter and knowing about which would have saved me many hours of debugging. The other sub-sections each discuss issues related to some specific aspects of the program, such as compilation, code generation and networking. I hope that they may be useful for people experiencing problems with these aspects when porting programs.

General remarks on porting software to MINIX

Different `ioctl` operations

The `ioctl` function allows software to perform driver-specific operations on file handles. Since drivers typically implement different functionality and use different names for `ioctl` control codes, even for the same devices on different operating systems, the `ioctl` function is a general portability problem. However, I think the problems it caused me with QEMU justifies warning not to overlook calls to `ioctl` call when porting software to MINIX.

To allow QEMU to compile even if some `ioctl` control codes are not supported by the operating system, code sections using nonstandard control codes are typically conditionally compiled using the `#ifdef` preprocessor directive. In these cases, a fall-back option is provided to deal with the case that no suitable control code is supported. For example, to find the size of a CD `DIOCGMEDIASIZE` is used on BSD and `DKIOCGMEDIAINFO` is used on Sun systems. The fall-back option is to seek to the end of the file and use the position of the end, but MINIX does not support this and returns zero. The result is that, by default, neither the compiler nor QEMU itself complains but the program cannot read CDs. The solution is to add another conditionally compiled section using the `DIOCGETP ioctl`, which is the MINIX equivalent of the other calls. To prevent spending a lot of time on finding the cause of the bug, it would be smart to search for all `ioctl` calls in advance to be able to know where to expect portability problems.

Another `ioctl`-related problem is that MINIX implements some calls for fewer devices than QEMU expects them to work for. I found this problem with the `FIONREAD` call, which determines the number of bytes still to be read on a socket. This function is implemented only for TCP sockets, but QEMU uses it for UDP sockets. As QEMU neither checked whether the `ioctl` function returned an error code nor provided a default value for the result, it was hard to find out that failure of this call was the reason I experienced network emulation failures.

Hardware floating point arithmetic is not supported

Most CPUs implementing the x86 architecture provide a floating point unit (FPU) which allows floating point operations to be performed quickly and accurately. Unfortunately, MINIX does not support use of the FPU. Although programs could in principle use it by issuing floating point instructions, the floating point registers are not saved by MINIX and multiple processes using the FPU would interfere with each other's computations.

The unavailability of the FPU is hidden from programmers as both the ACK and GCC compilers implement floating point computation by calling library functions rather than emitting x86 FPU instructions. These library functions implement floating point arithmetic using regular CPU instructions, but unfortunately the results are not identical to those provided by the real FPU. In particular, the FPU which comes with x86 chips uses an 80-bit representation of floating point numbers, called `long double` by C compilers. The floating point library only supports the `float` (32-bit) and `double` (64-bit) representations. Usage of the `long double` type is avoided by calling the `double` versions of the functions instead. This reduces accuracy as well as the range of numbers that can be represented. These issues can be worked around by using another floating point library that does support the `long double` type. As QEMU comes with such a library, intended for instruction set architectures that do not support an FPU, I decided enable this library. The inaccurate default library is replaced and virtual machines get the accuracy they expect. It should, however, be noted that due to this issue floating point arithmetic is truly slow on MINIX.

Besides causing reduced accuracy, it should also be noted that emulated floating point support also drastically reduces performance of programs that use floating point computations. This should be taken into consideration when deciding whether to port compute-intensive software to MINIX, or when considering running such software on top of the MINIX port of QEMU.

Another issue with the MINIX floating point implementation is that it does not include functions related to rounding, such as `nearbyint`, `rint` and `fesetround`, and also lacks the `remainder` function. Each of these functions is mandatory in POSIX and version 3 of the Single UNIX Specification [22], so their addition also makes other ports easier. Since these functions are needed by QEMU, I provided implementations for them. These implementations are found in the `minix-math.c` file, which is included in the `/qemu/qemu-0.8.2-minix/minix` directory of the CD-ROM that comes with this thesis. Further information on the contents of the CD-ROM can be found in Appendix A. I implemented the `nearbyint_rm` function by operating directly on the bits of the IEEE 754 double type as specified in [9]. First it truncates the number by setting all bits which have a significance smaller than one to zero and then it corrects for the rounding mode. All rounding modes are supported: towards zero (truncation), upward (ceiling), downward (floor) and to nearest

(bankers' rounding). This is needed as the emulated FPU should also be able to deal with every rounding mode. This function is used to implement the others.

Lack of a loopback device

On many operating systems, the `localhost` (IPv4 address 127.0.0.1) address allows one to use the networking functions without using an actual network adapter. This can be a convenient method of interprocess communication, with the additional advantage that it is easy to switch between interprocess communication on the local machine and over the network. Unfortunately, MINIX implements `localhost` in a different way than do most other operating systems, resulting in an unexpected reduction in functionality. Rather than delivering local packets through a special network loopback device, local packets are delivered to the same device handling the physical network adapter. This has a two implications for user programs:

- Packets to `localhost` are delivered only if the computer has an IP address. This can be tested by running `ping localhost`. This only works if the computer has an IP address, for example obtained from a DHCP server or set using `ifconfig`.
- The `bind` function, which is typically used by server programs to indicate which IP address and port number to listen on, fails if the IP address 127.0.0.1 is specified.

The former issue causes socket IPC to only be possible with either a working network connection or if `ifconfig` has been used to set an IP address. This is not needed on operating systems that use a separate device, such as `lo` on Linux, to deliver local packets. In this case the IP address of the physical network adapter (if any) is irrelevant for the delivery of local packets. One result of this is that X11, which uses sockets to communicate between the X server and X clients, can only be run if either the network is functional or a fake IP address is specified using `ifconfig`.

The latter issue is particularly relevant for porting programs. A server program may attempt to `bind` to `localhost` to make sure that only local requests are accepted, for example for security reasons or to avoid interfering with other servers. In this case `bind` will fail, specifying that it cannot bind to the address. Considering that `localhost` should always be a valid IP address, this had me puzzled. Originally, I assumed that I got the byte order of the IP address or port wrong or that the port was already in use. Only inspection of the MINIX source code made me understand the cause was really this issue. An easy workaround is to specify `INADDR_ANY` instead of 127.0.0.1, causing connections from any IP address to be accepted. If it is important that only local connections are accepted, one should check the remote addresses returned by the `accept` socket call.

Limited implementation of socket functions

On most operating systems, one can use the `read`, `recv` and `recvfrom` functions to receive incoming data on sockets and the `write`, `send` and `sendto` functions to send data. The `recv` and `send` functions are more flexible than `read` and `write` respectively as they allow one to specify flags. The `recvfrom` and `sendto` functions are even more flexible, as they allow one to determine the source host or to specify the destination host. This is useful with datagram-based services such as UDP, which operate without connections so that these cannot be determined from the handle.

MINIX implements the `recv` and `send` functions by calling `recvfrom` and `sendto` respectively and only implements the latter two functions for UDP sockets. Moreover, even though they are implemented, MINIX does not provide headers for the `recv` and `send` functions. QEMU, uses the `recv` and `send` functions specifying TCP sockets. When QEMU originally warned that these functions were being declared implicitly, I found that they were implemented and added their headers. This eventually caused them to fail, even though the calls would be perfectly valid on other operating systems. Fortunately, QEMU never specifies any flags when using these functions, so I was able to fix the issue by replacing `recv` and `send` calls with equivalent `read` and `write` calls.

Another minor issue with the MINIX network functions is that the `recvfrom` function contains a left-over debug message: "recvfrom: for fd %d\n". This is more a minor annoyance rather than a real problem, but I did fix it as part of my MINIX patches. It has also been fixed in the version of MINIX found on SubVersion.

Memory allocations are more likely to fail

Since MINIX does not support virtual memory, each process obtains a fixed amount of memory when it is started and when a new executable image is loaded. This means that assigning the right amount of memory in advance is crucial for the programs to function adequately. If too little memory is assigned, memory allocations will start to fail at some point. If too much memory is assigned, memory is wasted and fewer processes can be run simultaneously. The total amount of memory needed is hard to predict, causing the `malloc` function to return `NULL` occasionally if insufficient memory has been assigned. This is not as common on most other operating systems, where virtual memory allows the amount of memory used to be changed at runtime.

Because failure to allocate memory is so uncommon on other operating systems, programmers often do not check whether `malloc` succeeded. If `malloc` fails and this is not noticed, data is written to the `NULL` pointer and the memory locations following it. As elaborated on in the next sub-section, this need not result in a segmentation fault. Hence, failed allocation can cause data corruption, the effect of which may not be noticed until much later on. Hence it is important to make sure that programs ported to MINIX always check whether memory allocations succeeded.

`NULL` is a valid pointer

`NULL` is a valid pointer on MINIX. This means reads from and writes to the `NULL` pointer will not fail, but rather go undetected. Other operating system typically use paging to mark `NULL` as an invalid address, causing a `SIGSEGV` signal immediately on attempts to read, write or execute at or close to this address. This is currently not possible in MINIX as it does not use paging; this may change when virtual memory is introduced. In the case of QEMU, bugs in some cases caused jumps or calls to `NULL`, which I did not recognize as such initially. Such jumps and calls may fail eventually or they may restart the program, as the `NULL` pointer is a valid and common value for the program entry point in MINIX. In either case, a core dump is likely to be produced eventually but it is harder to analyse than it would have been if the program had failed immediately.

The simplest solution to be able to detect jumps to `NULL` is to create a binary with a shared instruction and data segment and then write an invalid CPU instruction to `NULL`. Combining

the instruction and data segments is needed because it is not possible to modify the instruction segment otherwise. When using the ACK compiler, the `-com` linker switch combines the segments. With GCC no switch is needed as it always combines segments. One effective way write an invalid CPU instruction is to add following code to the program in such a way that it is executed as soon as possible:

```
#ifdef __minix
    *(short *) NULL = 0x0b0f;
#endif
```

This code fragment sets the instruction at `NULL` to the `UD2` x86 opcode, which is guaranteed by Intel to always be considered an illegal instruction [10]. Executing it results in a `SIGILL` signal and typically a core dump. The instruction is represented by the bytes `0x0f` and `0x0b` (note that the bytes are reversed in the code fragment because the x86 architecture is little endian). It is important that the code is only included on MINIX, as it would cause a segmentation fault on most other operating systems.

The main advantage of the approach described in the previous paragraphs is that it is simple and does not require changes to the operating system. Unfortunately, it only detects jumps or calls to `NULL` and does not detect read or write operations. Catching bugs related to reading from or writing to `NULL` requires cooperation from the CPU, which in turn means MINIX must be modified.

For my testing, I have implemented the `mprotect` system call in MINIX. This call allows processes to mark which memory pages can be read from and written to. This call is typically implemented on operating systems supporting virtual memory. It was reasonably easy to implement on MINIX because, although it does support virtual memory and does not really use paging, paging is enabled and a page table corresponding with the identity map is used. My implementation of `mprotect` can be found on the CD-ROM supplied with this thesis in the `minix/minix-3.1.2a-mprotect` directory. The contents of this CD-ROM are described in Appendix A.

If one makes sure that the first memory page—4096 bytes on the x86 architecture—of the binary image contains nothing important, this function allows one to completely block access to `NULL`. This is the default behaviour on many operating systems and it makes bugs involving the `NULL` pointer very easy to detect. Experience shows that both ACK and GCC place the entry point—the code calling the `main` function—at the beginning of the executable image. After the program has started, this code should never be used again and access to it can safely be blocked. To avoid other useful functions from being blocked, garbage should be place directly after the entry point code to fill up the first page. This garbage should be code rather than data, as data is always placed after the code section.

It should be noted that my `mprotect` implementation is intended for debugging purposes only; it has not been tested extensively, but it does show that solving the `NULL` pointer problem is possible in MINIX with few changes. I expect a version of this function suitable for serious use to become available in MINIX when virtual memory is implemented.

Random branching when signals arrive

Due to a bug in the `sigreturn` function, the CPU flags are not properly restored after a signal. This goes unnoticed most of the time, but will cause random crashes and weird behaviour eventually. This is the same issue described in section 3.2. I would recommend anyone

noticing unexpected behaviour in the program they are porting to attempt whether applying this fix helps. This is particularly likely to help if the program receives many signals and if the issue involves flow control statements, such as `if`, `for` and `while` in C, behaving erratically.

Changes related to compilation

As mentioned before, my first aim was to get MINIX to compile. As I was not very familiar with QEMU yet, having the compiler pinpoint issues was the most convenient way to find what should be changed. Fortunately, as I was using the same compiler used on other platforms, I did not need to fix any syntax errors. If I had used ACK, there would have been many such errors; for example, the `//` single-line comment style is often used, while ACK only accepts `/* ... */` multi-line comments. Although the former style is not valid in C code, it is accepted by GCC. This means all remaining errors and most warnings pointed to real issues.

As I believe that a program should compile without warnings to really be correct, I have fixed most warnings as if they had been errors. Only one type warning could not be eliminated easily, so I disabled it. This is the warning about type compatibility in `printf` calls. To avoid these warnings, the types specified in the format string should match exactly the types of the arguments, but this is hard to accomplish if types are set to `int` on one platform and `long` on another. This distinction is irrelevant on MINIX, as both types are identical at 32 bits, although it might become a problem if MINIX were to be ported to a 64-bit architecture.

Build script and configuration

As I discussed in section 3.1, I created a `build.minix` shell script that allows QEMU to be built. This script is required by the package manager to have a fixed way of building packages, but it is also useful for people making changes to QEMU to be able to fully recompile easily. The build script takes care of the following things:

- Specify where the GCC compiler and its tools are to be found. This is needed because, in MINIX, these programs are in the `/usr/gnu/bin` directory rather than in one of the default binaries directories.
- Run the `configure` script to set compilation parameters. This script cannot be run using the default shell that comes with MINIX, so the Korn shell is used instead.
- Perform a clean build and installation using the `Makefile`.
- Set the amount of memory that QEMU is allowed to use. MINIX requires this amount of memory to be set in advance using the `chmem` command. I chose to assign 144 MB of memory by default, which is enough to run a virtual machine with the default setting of 128 MB of RAM.

I made the following changes in the configuration script:

- Allow MINIX to be detected and specify `CONFIG_MINIX` to the Makefile if the program is being compiled on MINIX. The `uname -s` command is used to detect the operating system; it returns `Minix` on MINIX.
- Add the `–enable-deterministic`, `--enable-histogram`, `--enable-host-time` and `--enable-qemu-profile` options. This options enable new features I added, which are disabled by default. These features are discussed in section 3.4.

- Allow the archiver to be used to create libraries to be specified on the command line. Normally the default archiver `ar` is used, but I use `/usr/gnu/bin/gar` from the GNU toolchain instead of the MINIX default implementation.
- The configure script creates a file `config-host.h`, which is included in many of the source files. On MINIX, this file includes `minix/minix.h`, which includes header definitions that are missing in MINIX. I will discuss this issue in more detail later.
- Additions to allow the Curses library to be used instead of X, based on a patch found in the Debian Linux package management system.

Removing references to missing headers, functions and fields

Some functionality could simply not remain in the MINIX port of QEMU, as MINIX does not support these features at all and writing an implementation of them—either in user mode or as part of the operating system—is not feasible. I found three such issues:

- Virtual memory is not supported, including the ability to protect memory regions against change and the ability to map files to memory. This means that the `sys/mman.h` header file as well as all functions it contains are missing. The result for QEMU is that:
  - QEMU can only be built using the soft-MMU configuration. This means that it emulates the guest MMU entirely in software rather than use the host MMU whenever possible. It is to be expected that this reduces performance somewhat.
  - I had to disable the COW virtual disk format, which is implemented using memory mapped files. The reasons for and implications of disabling it are discussed in more detail in the section on missing features.
- The `IP_MULTICAST_LOOP` socket option and the use of out of band data with TCP sockets are not supported, as these features are missing in the MINIX network driver. Again, the implications are discussed in more detail in the section on missing features.
- There is no `st_blocks` in `struct stat`. On some other operating systems, this member is provided to indicate how much space a file takes up on disk. This may be less than the actual file size if the file has holes or is compressed. It may be slightly more if the file allocates a disk block that is not completely used. In QEMU, this member is used to show information about virtual disk files. Therefore, getting the exact number is not all that important. I simply use the file size rounded up to the maximum disk block size, which is an overestimation of the actual allocated size.

The code sections related to the issues mentioned here were conditionally disabled using the preprocessor. In most some cases I added `#ifdef __minix ... #endif`, while in others I added `&& !defined(__minix)` to an existing `#if` construct.

Adding missing declarations

Those references to missing header files and functions that were too important to be removed had to be added in additional header files. For this purpose, I created a `minix/include` subdirectory for the header files that were missing or incomplete. These headers are always included by including them in the `minix/minix.h` file described before.

There is also a number of header files that need to be included in MINIX but not on other operating systems. For example, Linux automatically includes `sys/select.h` whenever `stdlib.h` is used. This does not happen in MINIX, resulting in compilation errors if its declarations are used. The same goes for some other header files, which are either included from the `minix/minix.h` file if they are used often or in the source files themselves otherwise. It should be noted that this has additional consequences. For example, the file `dyngen-exec.h` intentionally does not include `stdint.h` to be able to redefine the `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` types. On MINIX, this file is indirectly included through `minix/minix.h`. To avoid compilation problems due to redefinition of these types, their declarations are disabled on MINIX. This does not cause problems because the sizes of these types are always correct on 32-bit systems.

To be more concrete, I found the following declarations used by QEMU to be missing in the MINIX header files:

- The `ENOTSUP` error code in `errno.h`;
- Rounding control functions and constants in `fenv.h`;
- Many floating point functions in `math.h`;
- 64-bit integer support in `stdint.h`;
- The `realpath` function in `stdlib.h`;
- The `localtime_r` function in `time.h`;
- The `inet_aton` function and some socket-related constants in `sys/socket.h`.

Adding these missing definitions to get QEMU to compile was my first concern, while implementing the missing functions was done later based on the missing symbols reported by the linker.

Another header file causing problems is `zlib.h`. In MINIX, this file is not installed in `/usr/include` but can only be found in `/usr/src/lib/zlib-1.2.3`. As a solution I added this directory to the include path in the makefile.

Removing compiler warnings

Some of the changes I made just to remove compiler warnings are not discussed in detail, but I mention them in this section to be complete:

- There were some warnings about unassigned variables being used. In all cases the variable would always be initialized before being used, but the compiler could not detect this. I fixed these warnings by initializing them in their declarations.
- The `isalpha` macro in the `ctype.h` header file determines whether a character is alphabetical. Both MINIX and Linux use an array lookup to do this, but Linux casts the argument to an `int` first while MINIX does not. This causes an array lookup with a `char` index on MINIX. As `char` may be signed on some compilers and unsigned on other compilers, the array lookup is ambiguous and the compiler emits a warning. This was fixed by explicitly casting the argument to the `int` type.
- Socket functions that return a socket address, such as `accept`, `getsockname` and `recvfrom`, use the `socklen_t` type defined in `sys/socket.h` to return the size of the

socket address. This type is defined as `int` on Linux and `long` on MINIX. QEMU used the `int` type directly, which goes unnoticed on Linux but causes warnings on MINIX because, even though the types are identical in the current 32-bit world, they may become different in the future. I fixed this by using `socklen_t` instead of `int` wherever appropriate.

- Some variables and labels were unused because the only place they were used was disabled by `#ifdef` compiler directives on MINIX. I fixed this by conditionally compiling their declarations using the same condition.

Although these changes may seem trivial and irrelevant corrections—they probably do not make any difference to the compiled files—I still consider these changes to be useful when porting. Compiler warnings are a way in which the compiler helps pointing out potential issues. By eradicating all warnings, either by fixing them or by thinking through their potential consequences well and then disabling them, one can be sure to have used this help to the largest extent possible.

Changes related to code generation

As has been discussed in the section of the implementation of code generation in QEMU, information about code fragments is extracted from object files by the Dyngen tool. Using this information, it generates C code that can copy the code fragments and make the proper adjustments. In MINIX, object files use the "a.out" format. This is a very simple binary format which has fixed sections to be loaded—code, initialized and uninitialized data—directly followed by relocation and symbol tables. Dyngen uses the symbol table to find out what code fragments exist and how large they are; implementations of code fragments are marked by the prefix `op_` in their function name. The relocation table is used to find out which pointers must be adjusted when the code is moved around. The code itself is also used, both to find the exact end of the function by looking for the return opcode and to find the offsets of the pointers to be relocated.

For Dyngen to be able to read this information from an a.out file, a number of functions and parts of functions had to be implemented. Most of these were rather simple as the records in a.out files have a reasonable correspondence with those in other object file formats, but I found that I needed to make some adjustments. For example, the a.out file includes leading underscores in symbol names defined in C code, which Dyngen does not expect so they have to be removed. As symbol sizes are not stored in the symbol table, these are computed as the distance to the next symbol. This is not entirely accurate, as functions are padded so as to be aligned on four-byte boundaries, but this fixed by searching for the return opcode at the end of the function and adjusting the size afterwards. The size of functions is of particular importance, because the function bodies are glued together, which means that the return opcode must be chopped off in order for code generation to work.

The most difficult part of getting code generation to work is not to read the data in a.out but to properly apply it. Unfortunately, the way in which relocations work in the GCC variety of the MINIX a.out format is not well-documented. I found that in the object file containing the code fragments, there are several kinds of relocation that must be processed in different ways. This depends on (1) whether the pointer is relative to the program counter or is an absolute address and (2) whether the symbol being relocated is defined in the same object file or is external. Relocations relative to the program counter need to be adjusted for the address

pointed to as well as the location after copying, while absolute pointers only need correction for the former. For the relocations which refer to a symbol declared in the same object file, the value of the symbol is added by the compiler. In these cases, a pointer to the section it is in should be added rather than the value of the symbol itself.

Changes related to networking

Support for virtual networks in QEMU has been expanded in the MINIX port to make sure that at least the same level of functionality is available on typical MINIX installations as on Linux systems. Changes to ensure this include an implementation of the platform dependent TAP networking mode for MINIX as well as a new program called `qemu-vswitch`. As these changes are really additions rather than changes of existing functionality, they are discussed at length in section 3.4.

As was discussed in the section on general issues, `ioctl` control codes and functions related to networking do not always behave as they do on other operating systems. This causes the SLIRP library, which is used to provide virtual network support without needing root privileges, to function badly even though it does compile. Without adjustments, UDP works most of the time but client TCP connections are often closed due to errors occurring in the SLIRP library. The errors which have been corrected, of which some have been discussed before in the section on general issues, are the following:

- The SLIRP library uses the `recv` call on TCP connections and this functionality has not been implemented in MINIX; as flags were never specified, the `recv` call was equivalent with the `read` call in all cases and replacing it improved the situation;
- The `FIONREAD` `ioctl` control code is used to determine the size of incoming packets; MINIX defines this control code but does not implement it for UDP sockets. QEMU uses the control code without checking for errors, resulting in the variable receiving the number of bytes having an undefined value afterwards. This causes random behaviour which is correct most of the time but not all of the time; I finally found this out because negative buffer sizes were being passed to the `recvfrom` function. I fixed this by using a default value of 65 535 on MINIX. This does not introduce a limitation on packet sizes as network packets cannot be this large. Error checking was also added, even for the non-MINIX case.
- MINIX does not support the `SO_TYPE` socket option, which is used to determine whether the socket associated with a file descriptor uses UDP or TCP. To be able to find this out on MINIX, the `NWIOGTCPOPT` and `NWIOGUDPOPT` control codes are used instead to test the socket. If the former is sent using `ioctl`, it returns information for a TCP socket and fails with the code `EBADIOCTL` for a UDP socket; if the latter is sent, the results are the other way around.
- The `SO_REUSEADDR` and `SO_OOBLINE` socket options are also not implemented in the MINIX TCP driver even though they are specified in the header files. The `setsockopt` function fails if these options are specified, which causes several QEMU features not to work. Fortunately, these options are not essential and the corresponding calls are simply removed on MINIX.

Miscellaneous changes

Dealing with failed memory allocations

When testing QEMU, it turned out that saving a screen dump often caused it to crash with a segmentation fault. After such crashes, `mdb` showed the code segment to be completely overwritten with apparent garbage up to the instruction pointer. Screen dumps are produced by allocating an additional buffer, writing the current screen image to that buffer using the default screen update functionality and then saving the contents of that buffer to a file. Since in MINIX the amount of memory available for process is fixed in advance using `chmem`, it is quite normal for the memory allocation step to fail at high resolutions. In typical operating systems, where memory is dynamically assigned to processes, this rarely happens. QEMU did not consider this possibility and wrote the image to the pointer returned by `malloc`, which was `NULL` as the memory allocation had failed. Since in MINIX `NULL` is a valid pointer referring to the start of the code segment, the problem would only show after the image rendering code had overwritten itself.

After discovering this issue, I have searched the QEMU source code to find out whether there were more unchecked memory allocations and found many. Since most of them only allocate small chunks of memory, their failure would most likely go unnoticed; the functions at the start of the address space are used for initialization and not typically called during normal execution. This makes them even more of a problem than larger memory allocations; the simultaneous use of the start address space could lead to data corruption without the user noticing it. For this reason I defined a macro `MALLOC_CHECK` which checks whether the last memory allocation succeeded and exits the program with an error message if it failed. This should prevent further issues from going unnoticed.

Determining the size of removable media

QEMU allows the user to let a virtual machine access a physical storage device directly by specifying its device file as an argument to the `-hda`, `-hdb` and `-cdrom` options. To properly emulate the device, QEMU needs to find out its size. This is a platform-dependent operation: on BSD and Solaris the `DIOCGMEDIASIZE` and `DKIOCGMEDIAINFO` `ioctl` codes are used, respectively. On other systems, the `lseek` function is used with the argument `SEEK_END` to return the address of the last byte. On MINIX, none of these approaches works for CD-ROM drives. This originally caused QEMU to think the disk was zero bytes in size, causing emulation of the CD-ROM drive to fail. I use the `DIOCGETP` control code instead to get size of CD-ROMs on MINIX. This currently always returns 800 MB for ATAPI devices, as specified in the `atapi_open` function in `drivers/at_wini/at_wini.c`. This is not really a problem, as QEMU is satisfied with an upper bound. Using the control code seems to be a better solution than simply using some constant; this would fail if, for example, MINIX starts supporting DVD drives in the future.

Implementation of missing library functions

A number of library functions that are missing in MINIX had to be implemented. As mentioned before, compilation was possible without warnings because I provided headers for them. During the linking phase one can see which functions are really being used, and I implemented these functions whenever their use could not easily be eliminated. These

functions are `localtime_r`, `realpath`, `strtoull` and a number of floating point arithmetic support functions. I discuss each in turn.

The `localtime_r` function returns the current time in the configured time zone. In this sense it is identical to the `localtime` function, which MINIX does implement. However, this function stores the result in a buffer passed to it by the caller, while the `localtime` function returns a pointer to a static buffer. This means that the `localtime` function is not re-entrant. This is a problem when signal handlers use the function or when multiple threads use it. I implemented it by calling the `localtime` function and copying its result to the caller-supplied buffer. To avoid re-entry, signals are blocked using the `sigprocmask` function so signal handlers cannot call the function before the buffer is copied. As MINIX does not support multi-threading, this is sufficient to safely implement `localtime_r`.

The `realpath` function builds an absolute and canonical path by resolving symbolic links and references to the current or parent directory. MINIX does not implement this function, so it has been implemented from scratch on top of available operating system functions. First, the current working directory, obtained using `getcwd`, is added if the path is relative. Next, `lstat` is used for each component to determine whether it is a symbolic link. Link targets are resolved using `readlink`.

MINIX does not support provide support for 64-bit integer arithmetic. The GCC compiler, on the other hand, does support it. This means that 64-bit types can be used in computations, but that support functions are missing. One such support function is `strtoull`, which parses a string representing an integer and returns the resulting unsigned 64-bit integer. Again, I implemented this function from scratch.

MINIX does not support floating point arithmetic. Like 64-bit integer support, GCC generates code to emulate floating point operations using integer instructions but some support functions are missing. In particular, I implemented a number of rounding functions, reduced accuracy functions, comparison macros and a classification function.

Rounding functions include `fegetround`, `fesetround`, `nearbyint`, `remainder` and `rint`. The first two functions get and set the rounding mode in the floating point unit (FPU) control word. This influences subsequent rounding operations, including those involved in representing answers as floating point numbers. I did not find a way to modify rounding behaviour in the FPU emulation code, so my implementation only affects the other three rounding functions I implemented. This is sufficient in practice, as the rounding direction is much more important when rounding to integers than when rounding to representable floating point numbers. The other rounding functions have been implemented by directly manipulating the fields of the IEEE 754 floating point type, as specified by Intel [9].

Reduced accuracy functions, such as `sqrtf`, use the `float` type rather than the `double` type to represent arguments and results. Some of these functions are missing in MINIX even though their counterparts with `double` accuracy are present. I implemented these functions simply by using the `double` accuracy functions, which results in GCC implicitly converting the arguments and the result. Emulating 32-bit floating point arithmetic directly would be more efficient than this solution, but I expect that these functions are rarely used so that it does not matter much in practice.

Comparison macros and the `fpclass` classification function are mostly used to deal with floating point values that are not ordinary numbers. Examples are infinities and NaN (not a

number) values used to signal error conditions. The comparison macros are used to make sure that NaN values are not compared directly, but rather that any comparison involving them is false. This is based on the `fpclass` function, which parses the fields of the floating point value specified to determine what kind of value it is.

Support for 64-bit integers in format functions

QEMU occasionally uses 64-bit integers, which are at times also passed to `printf` and related functions to be displayed. Unfortunately, the MINIX implementations of these functions cannot deal with 64-bit integers, which means they cannot parse the format strings referring to them and consequently interpret their arguments incorrectly. Rather than completely re-implement these functions and change all references to them, I opted to replace the `_doprnt` function which is the core of their MINIX implementations. This function takes the format string, a variable argument list and a stream object as parameters. It parses the format string, inserts the arguments with the appropriate formatting and prints the result to the specified stream. I copied the source code for this function from `lib/stdio/doprnt.c` and modified it to be able to handle 64-bit arguments. The `main` function of QEMU is modified to call `doprnt64_activate`, which overwrites the `_doprnt` function with a jump to my implementation `_doprnt64`. As a result, the entire family of `printf` functions can deal with the `ll` format prefix that specifies a 64-bit argument. The functions described here can be found in `qemu/qemu-0.8.2-minix/minix/doprnt64.c` on the CD-ROM that comes with this thesis.

Missing functionality

Unfortunately, not all functionality could easily be ported to MINIX and some had to be disabled. This section summarizes the missing features in QEMU when running on MINIX.

COW disk image file format

QEMU supports a number of different file formats for storing disk images. These file formats allow for efficient storage of virtual disk images by only storing those disk blocks that are in use. Some also provide additional features, such as compression and encryption. Multiple formats are supported to allow for easy interchange of virtual machines between different emulators. Besides its native QCOW format, QEMU also supports disk images used with Bochs, Microsoft Virtual Server and VMWare. It also deals some special cases, such as the RAW file format in which the disk contents are stored in the file verbatim. This allows for sharing real disks, disk partitions and CD-ROMs with the virtual machine. Each format is implemented as a block driver within QEMU. These drivers have a common interface and individual drivers can easily be disabled.

Unfortunately, one of the block drivers depends on memory mapped files. Since MINIX does not support virtual memory, it does not supply the `mmap` system call needed for this. The affected driver is the COW driver, the predecessor of the QCOW format and QEMU's former default block driver. This driver has been disabled on MINIX. This should not be much of a problem as old COW images can easily be converted to QCOW using the `qemu-img` utility. Support for this driver is also missing on Windows, which confirms the fact that this driver is not essential. It would be possible to implement a driver for this file format that does not use memory mapping, but this does not seem to be worth the effort for a legacy feature like this.

Linux-specific features

Some QEMU features are available only on Linux and are therefore not present in the MINIX version. These features include use of a high-resolution timer device and redirection of USB devices so that the virtual machine can interface with them directly. Since neither a high-resolution timer not USB support are present in MINIX, it would not be meaningful to include these features unless substantial changes are made to the MINIX operating system.

Networking-related features

The `IP_MULTICAST_LOOP` socket option causes multicast messages to be sent to the loopback network interface, which allows multiple instances of QEMU on the same machine to communicate with each other using UDP. This socket option is not supported on MINIX. The result is that UDP-based virtual networking may not work well if multiple instances are running on the same host. Fortunately, this can easily be worked around by the user by connecting them using TCP instead. The `qemu-vswitch` program can be used in this case as an easy way to connect many virtual machines together.

Out of band data on TCP sockets is not supported by the MINIX network driver either. The result is that, if user mode networking is used, out of band data cannot be sent or received. This should not be a major issue, as use of out of band data seems to be very uncommon in practice. Moreover, the issue can be worked around by using ethernet tunneling or `qemu-vswitch` to connect to the network. In either case, the MINIX TCP/IP stack is bypassed and the virtual machine is free to implement out of band data.

Sound support

One feature that is missing entirely is sound support. To play sound, QEMU relies on SDL. When initializing the sound subsystem, SDL fails due to lack of support for threading primitives. These are required as SDL uses a separate thread to send sound output, which avoids the need to use a timer or to regularly call a function to fill the sound output buffer. Since MINIX does not implement threads and moving the sound thread into a separate process would be outside the scope of this project, there was no simple way to get the sound to work.

Because SDL originally failed due to the lack of a MINIX sound driver rather than missing thread support, I have worked on some sound issues in the hope to resolve these issues. In particular, I have created a MINIX sound driver for SDL and debugged timers in QEMU due to this issue. The former may be useful for those who wish to further pursue enabling SDL sound in MINIX, although it should be noted that I was not able to test the SDL sound driver due to the lack of threads. The latter revealed the issue that QEMU runs extremely slowly when sound is enabled in deterministic mode. I found this to be caused by the fact that the QEMU sound system creates a timer which is set to expire immediately. When using the `setitimer` function, this results in the minimum possible delay as determined by the clock frequency, but in deterministic mode the alarm expired after executing a single basic block. As a solution I set the minimal time-out to $\frac{1}{1024}$ of a second, QEMU's preferred clock resolution when running on Linux. This issue is relevant not just for the deterministic mode, but would also increase performance if the MINIX kernel were to be made tickless. Further

profiling of the timer callbacks revealed that no other functions use exceptional numbers of time-outs.

3.4 - Features added in QEMU for MINIX

Curses support

Running QEMU under the X environment in MINIX is somewhat harder than on similar operating systems due to memory limitations. Since MINIX does not support virtual memory, a fixed amount of memory is allocated each time a process is created or an executable is loaded. The amount allocated is set using the `chmem` utility and should be sufficient for the maximum amount of memory the process ever uses. Memory is not swapped out to disk, so the entire allocation is backed by physical memory. Moreover, memory allocations need to be contiguous because paging is not used. Therefore memory fragmentation may make it impossible to allocate large blocks even if the sufficient memory is available. In sum, MINIX typically requires more physical memory than do other operating systems.

Both X and QEMU need to allocate large buffers, so running them together can be hard on systems with little memory. Some X video drivers require buffers to back video memory, while QEMU uses a large amount of memory to store the contents of the virtual machine memory. Therefore QEMU's `chmem` value determines the amount of memory that can be assigned to a virtual machine. This leads to the conclusion that X reduces the memory size available for virtual machines, so it is desirable that QEMU can be run without X.

To be able to run QEMU without X, I added support for the Curses library. This library allows QEMU to present the display of the virtual machine on a console, as long as it uses a text rather than graphical display mode. This support has been merged in from the QEMU package for Debian Linux. Besides allowing virtual machines to have larger memories, this patch also make it possible to use QEMU on a MINIX machine over a TELNET connection. To use text mode QEMU, the `-curses` command line switch can be specified.

Memory allocation recommendation

As explained before, memory availability is a limitation for the memory size of virtual machines. If the value assigned with `chmem` is too low, QEMU fails when allocating memory for the virtual machine. If it is too high, QEMU may not run and less memory remains for other programs. Therefore it is important that the `chmem` allocation be properly adjusted to the largest virtual machine size used in practice. Therefore I added code in the `qemu_vmalloc` function, which is a `malloc` wrapper used for several large allocations. If the `malloc` call fails, it prints a message recommending the use of `chmem` and suggesting an estimate of the amount of memory to be assigned. Normally this amount is the memory size of the virtual machine plus eight megabytes. This is an estimate and, although it works for the virtual machine sizes I have tested, one cannot be sure that it is always correct. Therefore, the total amount of memory required after the allocation plus one megabyte is suggested instead whenever it is higher. This provides a fall-back if the user uses the other suggestion and it turns out to be too low.

Networking

An important QEMU feature is the ability to emulate network adapters as well as a virtual network. This is typically the easiest way to exchange data between virtual machines and the host machine and it is obviously required for the virtual machine to get access to the Internet. Efficient network emulation is especially important if the virtual machine is used to run server programs, which is one common use of virtualization. QEMU supports several different ways to create virtual networks, each of which has their own advantages and disadvantages. I have also created an additional server program called `qemu-vswitch` that allows for more efficient and secure network emulation on MINIX.

I will discuss the various approaches that can be used to connect QEMU running on MINIX to the network in turn. A summary of the advantages and disadvantages discussed here is provided in Table 2.

User mode networking

QEMU can emulate a network even if the user does not have root privileges. It does this by translating network packets sent from and to the virtual machine into UDP and TCP socket operations on the host machine. It emulates a DHCP server to provide the virtual machine with an IP address and a gateway to give it access to the local area network. This effectively places the virtual machines behind a network address translation (NAT) router.

	User mode networking	**Ethernet tap**	**qemu-vswitch run from QEMU**	**qemu-vswitch run at boot time**
Requires root permissions	No	Yes	Yes	For configuration but not for use
Topology	Behind NAT router	Connected directly to LAN	Behind switch	Behind switch
Limitations	Cannot act as server due to NAT translation; can only use UDP and TCP (no ICMP); subject to limitations of MINIX network stack, which means no out of band TCP data	None	Fork requires twice the memory assigned to QEMU to be available temporarily	None
Ease of use	Default, no configuration needed	Command-line switch	Command-line switch	Needs to be set up in advance
Security issues	None	VM can eavesdrop on host and spoof physical machines	Optional protection against eavesdropping and spoofing by VM	Optional protection against eavesdropping and spoofing by VM (decided on by root)
Performance	VM processes only directly addressed packets	VM has to process all packets, slowing it down with heavy network traffic	VM processes only directly addressed and broadcast packets	VM processes only directly addressed and broadcast packets

Table 2: Advantages and disadvantages of several virtual networking approaches supported by QEMU on MINIX

Unfortunately, there are certain restrictions that come with emulating the virtual network using host TCP and UDP sockets. For example, it is not possible for the virtual machine to accept incoming connections and it is not possible to send and receive ICMP packets. The former limitation implies that the virtual machine cannot function as a server, which is one important use of virtualization. It also makes FTP, a useful tool to exchange files between virtual and physical machines, harder to use. The latter limitation implies that network diagnosis tools such as `ping` and `traceroute` generally do not work properly. To mitigate this problem, QEMU converts ICMP ping packets into UDP echo packets; this allows `ping` to be used if the target host runs an echo server. An additional disadvantage is that the virtual machine uses the host networking stack and is therefore subject to its limitations and bugs. When using MINIX as a host operating system, this means that it is not possible to use TCP out of band data as this is not supported by the MINIX TCP implementation.

User mode networking is the default approach if no networking command line switches have been specified. It can also be enabled explicitly by using the `-net user` command line switch.

Ethernet tap

Another, more flexible way to link the virtual machine to the local area network (LAN) is to forward ethernet packets directly. This means that packets sent by the virtual machine are sent out using the physical network interface and that incoming ethernet packets are forwarded to the virtual network interface. As QEMU and the host operating system do not change any headers, there are few restrictions on the protocols used in those packets. This allows more freedom for the virtual machine. For example, it allows the use of ICMP by the `ping` and `traceroute` utilities. Moreover, the virtual machine can obtain its own IP address in the LAN, typically using the same DHCP server used by its host. This means that address translation is not needed and all packets can be forwarded to the virtual machine correctly. This allows it to accept incoming TCP connections and act as a server.

On Linux, the ethernet tap is implemented using the TUN/TAP device named `/dev/net/tun`. This device allows one to create a virtual network adapter which is either an IP tunnel (TUN) or an ethernet tap (TAP) [15]. The former function forwards IP packets between the physical and virtual network adapters according to IP routing rules, while the latter forwards all ethernet packets. Both functions are supported in MINIX but using different devices, respectively `/dev/ip` and `/dev/eth`. QEMU uses the TAP function, so I made it open the `/dev/eth` device. After proper configuration QEMU can use this device the same way it uses the TUN/TAP device in Linux. This configuration consists of telling the `/dev/eth` device to accept any incoming packet regardless of MAC address and make a copy of it available to QEMU. As packets from all MAC addresses are accepted, packets sent to the MAC address of the virtual machine can be received. As each packet is copied, other processes including the normal host network stack still receive every packet and are not affected.

When used on Linux, a setup script is needed to set up the TUN/TAP device correctly. On MINIX, QEMU will perform this configuration itself and specifying the `-net tap` command line option is enough to use ethernet tap networking. Unfortunately, this does require root permissions as by default only the root user is allowed to open `/dev/eth`. This approach also has the disadvantage that every incoming packet is sent to the virtual machine and packets sent out by the virtual machine are accepted without restrictions. This has two implications:

- The virtual machine has to process every incoming packet, even if it is not relevant to the virtual machine;
- The virtual machine can eavesdrop on all host network traffic and can send out packets that seem to originate from it or from other computers in the network.

The former issue can slow down emulation, especially if the host machine generates much network traffic. The latter issue implies that, if the programs running on the virtual machine are not trusted, network security may be compromised unless one uses cryptography to protect local network traffic.

Virtual switch

As can be seen from Table 2 and from the previous sections, both user mode networking and ethernet taps have problems. The former mostly suffers from limited functionality on the virtual machine and the latter requires root permissions and may provide bad performance and security. To make the situation better on MINIX, I have implemented a third way to provide the virtual machine with access to the network. This is a separate program called `qemu-vswitch`, which opens the `/dev/eth` device as QEMU instances normally would if the ethernet tap mode were used. This program allows instances of QEMU to connect to it using local TCP connections. This functionality is built into QEMU by default, as it is also used for multiple QEMU instances to communicate with one-another.

I will first discuss the way in which the virtual switch is implemented and then consider the implications of using the virtual switch rather than the network emulation features originally present in QEMU.

Implementation

Figure 10 shows a situation in which two instances of QEMU are using `qemu-vswitch` and two other host user processes are also using the network. Clusters of adjacent rectangles represent host processes, with each rectangle representing some functionality in the process it belongs to. Lines show interprocess communication and data is exchanged within processes wherever rectangles are adjacent. The illustration shows that `qemu-vswitch` taps packets from the `inet` server at the ethernet level, while the host network stack still operates normally. Even though multiple instances of QEMU are running, only one instance of `qemu-vswitch` and one ethernet tap are needed. When a packet comes in, the virtual switch decides which virtual machines need it and forwards it over the appropriate local TCP connections. Packets are routed using the MAC address, so there is no need to use specific higher-level protocols. Each guest operating system uses its own network stack to handle these packets when they arrive at its emulated network interface card (NIC).

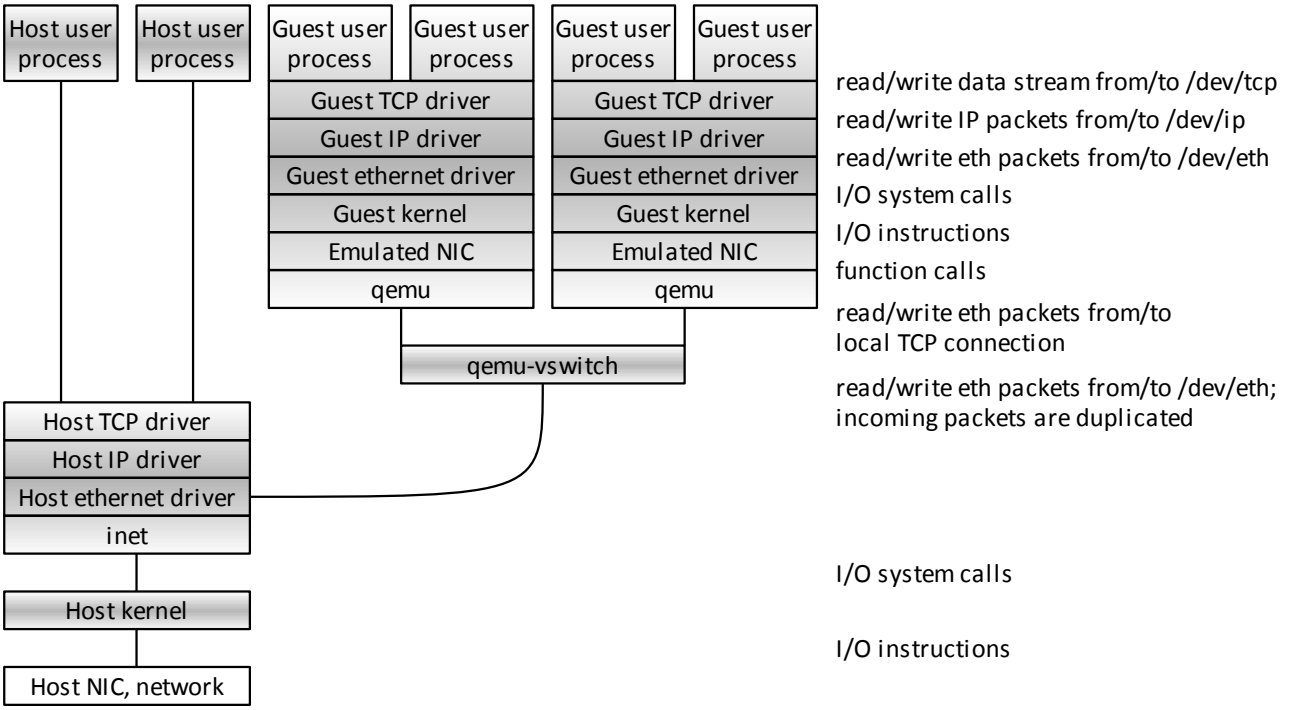

Figure 10: Processes involved in network emulation using `qemu-vswitch`

Figures 11 and 12 show how the virtual switch routes ethernet packets. Two instances of QEMU, labelled "qemu 1" and "qemu 2," are running and have connected to the virtual switch. The first virtual machine has been running for some time, while the second one has just been started and is initializing its network stack. For each instance of QEMU connected to it, `qemu-vswitch` builds a list of MAC addresses by storing the source addresses of outgoing ethernet packets. At this point the MAC address of the first virtual machine is already known (52:54:00:12:34:56), but the second virtual machine has not sent any packets yet and therefore its MAC address (52:54:00:12:34:57) is not known yet.

To obtain an IP address, the second virtual machine will eventually broadcast a DHCP discover packet. This packet has the source MAC address of the virtual machine (52:54:00:12:34:57) and its destination is set to the broadcast MAC address (ff:ff:ff:ff:ff:ff).

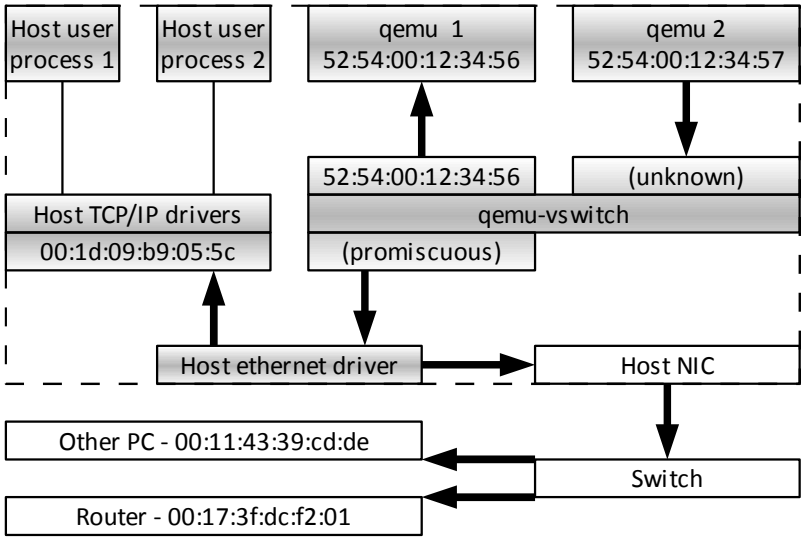

Figure 11: qemu-vswitch routing an outgoing DHCP discover packet

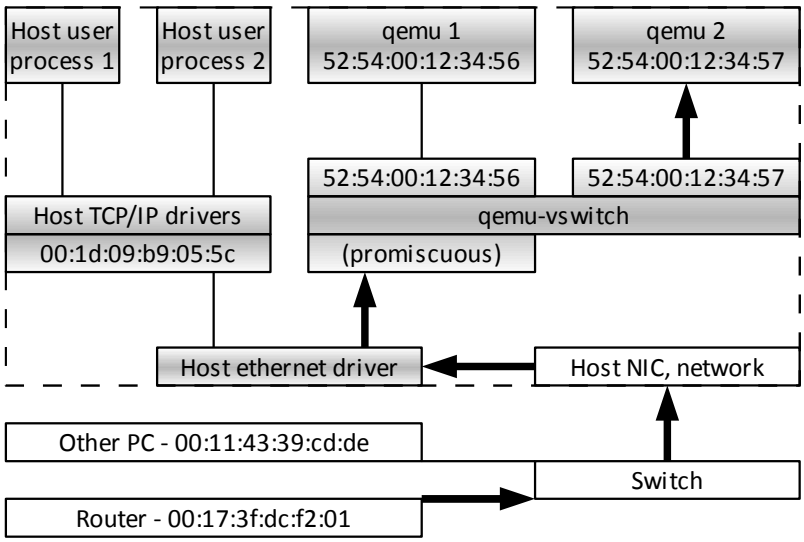

Figure 12: qemu-vswitch routing an incoming DHCP offer packet

`qemu-vswitch` stores the source MAC address and will be able to deliver any future ethernet packets correctly. The arrows in Figure 11 shows how the DHCP packet is routed; as the packet is broadcast, the packet is forwarded in all directions. This causes the packet to be delivered to the first instance of QEMU, the host TCP/IP stack, any other computers on the local area network and the router. In a typical home configuration, only the router would be running a DHCP server and all other receivers would discard the DHCP packet because they do not listen at the port number for DHCP.

When the router receives the DHCP discover packet from the virtual machine, it sends back a DCHCP offer packet indicating the IP address it allocated for the virtual machine. Figure 12 shows how this packet is routed. This time, the source MAC address is that of the router (00:17:3f:dc:f2:01) and it is sent directly to the virtual machine (52:54:00:12:34:57). By now, the physical network switch knows where it should send a packet with this MAC address, so only the host machine receives it. The host network stack forwards the packet to `qemu-vswitch`, but the host IP driver does not process it as the destination MAC address does not match the MAC address of the network adapter. `qemu-vswitch` uses its list of MAC addresses to determine where to deliver this packet, so only the correct virtual machine receives it.

Usage

The virtual switch program can be run in two ways:

- If QEMU is started with the `-net vswitch` command line argument, it starts `qemu-vswitch` and connects to it automatically. A file is created in `/var/run` to prevent the program from running twice, so that all instances of QEMU connect to the same virtual switch. This directory should be (but is not currently) cleaned up at boot time.
- The system can be set up to run `qemu-vswitch` at boot time. Users then start QEMU with the `-net socket,connect=localhost:5768` command line option to connect to it.

The former approach is easy to use, but it requires root privileges for the user starting QEMU. Memory usage is another issue. QEMU typically has much memory assigned to it as it needs space for the memory of the virtual machines. As on other POSIX systems, executing a

program on MINIX involves first calling `fork` and then `exec`. In between the memory claimed by QEMU is doubled, which may be a problem on many MINIX systems. This will be resolved if a future version of MINIX would use copy-on-write pages to implement `fork`.

The latter approach requires some configuration by the root user, but the user starting QEMU does not need root privileges. When no virtual machines are connected to it, `qemu-vswitch` only listens for incoming TCP connections and does not keep the `/dev/eth` device open. This means it uses very few resources when running in the background.

As `qemu-vswitch` filters packets and only delivers them where they are needed, virtual machines process fewer packets than they do when ethernet tap networking is used. Therefore better performance can be expected, especially if the host machine experiences heavy network traffic. If the `-f` command line switch is specified when running `qemu-vswitch`, it blocks outgoing packets specifying a MAC address that is assigned to a physical device. This means virtual machines cannot spoof packets from other physical machines. This also means that they cannot get packets from these MAC addresses delivered to them, so they cannot eavesdrop on physical machines either. Therefore, security can be better than with ethernet tap networking.

Opcode histograms

To be able to test QEMU's performance, I ran a number of benchmarks to find out which CPU instructions have most performance impact. To make this easier, I have created a feature to measure the number of times each type of instruction is executed. By inserting counting code into the generated code, it is possible to figure out how many times each instruction was executed after a benchmark. To make the C compiler generate the counting code in the same way as the regular code, an operation called `op_inc_opcode_histogram` is created and inserted before each other instruction when profiling is enabled.

As keeping track of the histogram causes the performance of the generated code to decrease, it is only done whenever enabled by specifying the `--enable-opcode-histogram` parameter when configuring the build and by the guest program doing the benchmark.

To allow the virtual machine to communicate with the host while making minimal changes to QEMU, I decided to modify the `CPUID` x86 instruction. This instruction is normally used to check for the presence of CPU features. Normally, the `EAX` register indicates what information is to be queried and information is passed back in the `EAX`, `EBX`, `ECX` and `EDX` registers. I added a function with a completely different value for `EAX`, the ASCII string “QEMU,” to avoid interfering with the normal use of this instruction. To allow the benchmark program running on the guest to control when measurement is enabled and process the results of the measurements, three functions are added: one to start measurement, one to stop measurement and one to fetch the instruction histogram. A more detailed description of these operations is included in Table 3.

Operation	Start measuring	Stop measuring	Fetch histogram data
Instruction	CPUID	CPUID	CPUID
EAX in	"QEMU"	"QEMU"	"QEMU"
EDX in	"hist"	"hist"	"hist"
ECX in	"star"	"stop"	"fetc"
EBX in	-1	-1	Index of first byte to fetch
EAX out			Data: first dword
EBX out	0	Number of opcodes in histogram	Data: second dword
ECX out	0	Histogram buffer size	Data: third dword
EDX out			Data: fourth dword
CF out	0	0	0
Side effects	Measurement enabled Code generation buffer flushed Histogram data cleared	Measurement disabled Code generation buffer flushed	

Table 3: Operations used by the guest to control the opcode histogram feature

If a program which expects to be running on QEMU with histogram support is running on the bare hardware or a version of QEMU compiled without support, it is important that it is able to detect whether the call succeeded. Running on the bare hardware, the effect of `CPUID` on the `EAX`, `EBX`, `ECX` and `EDX` registers is undefined if the value in `EAX` does not correspond with a known call, so these standard registers cannot be relied on for this purpose. Instead, QEMU clears the carry flag (`CF`) to indicate that the `CPUID` call was intercepted. As CPUID normally does not change the flags, it is possible to detect whether the call succeeded or not if one sets the carry flag in advance.

A typical benchmark program is expected to first call the "start measurement" function by setting the registers as described in the table and then issuing a `CPUID` instruction. If the carry flag is cleared, this indicates that the program is running inside QEMU and that the opcode histogram feature was enabled before building it. After the benchmark has been run, the "stop measuring" operation is called and the buffer size returned in `ECX` allows the program to allocate an adequate buffer. For each 16-byte block, the "fetch histogram data" operation is used and the data is copied from the registers into the buffer. This buffer contains an array of 64-bit unsigned integers, each counting the number of times an instruction was executed since measurement was started, followed by a list of null-terminated strings indicating the names of the instructions. This data allows the program running in the guest machine to present the results to the user or store them in a file.

Running deterministically

At times, it is convenient if a virtual machine behaves in exactly the same way when started from the same state multiple times. To be able to better debug QEMU, I modified the program in such a way that this is possible. If the `--enable-deterministic` option is passed to the configuration script before building QEMU, this results in the following changes:

- The guest system timer is set to a fixed value—midnight January 1st 2000—when the virtual machine is booted;
- The guest system timer increases as a linear function of the number of virtual machine instructions executed rather than based on real time;
- The moment at which timer events are fired occurs after a fixed number of virtual machine instructions, causing the system timer to also fire interrupts at fixed positions in the guest code.

It should be noted that user input and incoming network packets may still result in nondeterministic behaviour, but in cases where this matters these can be avoided by disabling network support (using the `-net -none` parameter) and not providing input.

The deterministic running mode is implemented by counting and checking the number of virtual instructions executed after each basic block. Although slows down the generated code somewhat, an advantage is that it avoids the need to use `setitimer`. This is beneficial if the guest operating system sets the interrupt timer at a high rate, since it greatly reduces the number of context switches because fewer signals are received. It also makes the timing of guest timer interrupts much more accurate, allowing clock resolutions significantly higher than that of the host without clustering interrupts together. One disadvantage of this approach is that the rate of change of the system time of the guest may depend on the speed of the host. This is compensated to some extent when the guest is idle, by adjusting wait times depending on the degree to which the virtual machine time runs ahead of or lags behind real time.

When QEMU runs in deterministic mode, it is no longer possible for benchmark programs running on it to obtain a meaningful estimate of their runtime because they do not have access to a source of real time. To still allow benchmarks to be run in this situation, I created a configuration option `--enable-host-time`. This option, like the opcode histogram discussed previously, creates a new function for the CPUID instruction. This function reads the time as reported by the `RDTSC` instruction and the `gettimeofday` system call on the host. Both are included because the former has a much higher resolution, while the latter is better corresponds with real time. It should be noted that this feature can also be used when QEMU does not run deterministically. Even though timekeeping by the guest is possible in this case, it is more accurate on the host as timer interrupts may not be delivered at exactly the right time. Moreover, it increases comparability of benchmark results to use this feature both when running deterministically and otherwise.

Operation	Read host time
Instruction	CPUID
EAX in	"QEMU"
EDX in	"time"
ECX in	0
EBX in	0
EAX out	Host time stamp counter (low-order 32 bits)
EDX out	Host time stamp counter (high-order 32 bits)
ECX out	Host real time (tv_sec)
EBX out	Host real time (tv_usec)
CF out	0
Diagnostics	In the (unlikely) case `gettimeofday` fails, both EBX and ECX are set to zero
Side effects	Measurement enabled Code generation buffer flushed Histogram cleared

Table 4: Operations used by the guest to read the host time

Simple profiling of QEMU

Besides the fact that generated code is likely to be considerably less efficient than the original code, QEMU itself also adds some overhead. In particular, it has to generate code, handle alarm signals to emulate timer hardware, check for input and produce output. To find out how much time is spent on these activities allows one to get an impression on what part of the time is spent productively and, in particular, whether there is a difference in this regard between MINIX and other host operating systems. In this way, it can be useful to be able to find performance bottlenecks in QEMU. The advantage of this approach over ordinary profiling is that it causes only very little overhead, which means that the observed run times are better estimations of real-world performance. The disadvantage is that it is much less precise in pinpointing performance bottlenecks, but this is not needed if one only wants to investigate how much of the time is spent productively.

To determine how much time was spent on each activity, I added calls to a new function `qemu_profile_set_mode` whenever the activity changes. These calls are conditionally compiled to allow profiling to be enabled when configuring the build. This can be done using the `--enable-qemu-profile` configure script switch. The function keeps track of the number of CPU ticks since the last call, which allows for very accurate time measurement. It should be noted that the number of CPU ticks per second need not be constant on modern power-saving CPUs, but it is in MINIX as this operating system currently does not implement power management. When QEMU exits, an overview of the percentage of time spent on each activity is printed to the standard output.

3.5 - libSDL

To be able to run QEMU with graphics support, the libSDL library is required. This library is a platform-independent intermediary between an application and a graphical shell. This library

was not yet available on MINIX 3. Fortunately, libSDL uses the GNU Autoconf library, which makes it highly portable. Moreover, other software using Autoconf can re-use the changes I made to the configuration files supplied with libSDL. Besides QEMU, the libSDL library is also useful for numerous other programs that use it.

In this section, I discuss changes made to libSDL to allow it to be used on MINIX 3. The source code and a patch that shows the differences are provided on the CD-ROM that comes with this thesis, stored in the `/SDL` directory. Changes are discussed file by file.

The `configure` and `configure.in` files

The configuration files are generated by GNU Autoconf. They prevent the libSDL source code from using features not present on the target platform. This is done by testing for features rather than based on operating system and compiler versions. This is very useful in this case, as this means that adding a new operating system—MINIX 3—does not require much effort. There are two configuration files—`configure` and `configure.in`—both of which have been changed in the same way. These changes were only small and few, which bodes well for porting other projects that use GNU Autoconf. The version of libSDL which I ported (1.2.13) uses GNU Autoconf 2.61 and changes may be slightly different for other versions.

The first change that needed to be made was adding a definition of the `_POSIX_SOURCE` precompiler symbol to the `BASE_CFLAGS` variable. This is needed because some symbols in the MINIX 3 header files only get declared if this symbol is declared. In particular, this is the case for nearly all functions related to signal handling. The `_POSIX_SOURCE` symbol is defined only when compiling on MINIX 3, instead of the `_GNU_SOURCE` symbol which used on other platforms. On these other platforms, `_GNU_SOURCE` implies `_POSIX_SOURCE`, while on MINIX 3 it is not used.

The configuration scripts perform a test to determine the maximum length of the command line. In MINIX—as on many other operating systems mentioned in the file—this test fails. The reason of this failure is the process running out of memory before the maximum command line length is reached. This is due to the restrictions posed by each process having a fixed amount of memory. Like on the other operating systems for which the test is known to fail, I made the script skip the test and use the default value of 8 kb.

Code is added to detect the MINIX operating system name as provided by the `uname` command. This is needed to avoid an error message about the operating system being unknown. Due to its POSIX compliance, MINIX is added to the list of Unix-like operating systems, performing the same feature check tests as for example Linux, BSD and Solaris.

Finally, `SDL_STATIC_LIBS` variable is set to supply the location of the X11 static library when compiling libSDL. This is needed because MINIX 3 does not support dynamic loading, which is common on other operating systems. By including this compiler flag, the X11 libraries are included directly into the libSDL library.

Changes to SDL files

GNU Autoconf takes care of most portability issues, resulting in very little need to make change to the SDL files themselves. Only two more files required changes and those changes were very minor. The `include/SDL_platform.h` file is changed to define the `__MINIX__` symbol when compiling on MINIX for consistency with other platforms. This symbol is used in

`video/x11/SDL_x11events.c`, where the `sys/select.h` header is included when running on MINIX. As mentioned with the general portability issues, other platforms automatically include this file whenever `stdlib.h` is used, so it is often needed to explicitly add it when porting software to MINIX.

Build file

To be able to build libSDL with a single command, as is required for making a `packman` package out of it, I have included a `build.minix` shell script. The configure script and the makefile do most of the work, so this script is rather simple. It specifies the proper directories, disables unsupported features and calls the configure script and the makefile. GCC is used to compile libSDL since it is also used for QEMU. The object file formats for GCC and ACK—although both forms of the a.out format—are incompatible so that the library can only be linked with QEMU if it is compiled using GCC.

The build script disables three features: assembly routines, direct graphics access (DGA) and Xi Graphics Miscellaneous Extensions (XME). The former two are performance enhancements. The latter provides support for a proprietary X server by Xi Graphics. The problem with the assembly routines is that they use the MMX and SSE instruction sets. Since MINIX does not save the MMX and SSE registers at context switches it is not safe to use these functions. Although they could be rewritten to avoid these instruction sets, little would probably remain of their improved performance. DGA is another way to increase performance, which allows the X client (QEMU in this case) direct access to the graphics memory allocated by the X server. Hence the sockets interface that is normally used for communication between the two can by bypassed. However, due to the lack of virtual memory, it is not possible to make the shared memory available to the client. XME is not supported because it uses threads, which MINIX does not support. This extension would not be useful even if threads were available, because the X server that it interfaces with is not available on MINIX. Hence, the changes may affect performance but do not reduce functionality.

3.6 - Debugging QEMU

Debugging QEMU turned out to be hard because in many cases, errors only crash the program a long time after they occur. In particular, errors in generated code often resulted in jumps to random locations or in code being overwritten. If such problems happen, much code may be executed before something goes sufficiently wrong for the program to crash. At this point, the only possibility is to open the core dump file created by the crash and figure out manually from the contents of the stack where the original problem was. Another problem was a situation in which QEMU would hang so badly that it could only be ended by sending it the `SIGKILL` signal, which does not produce a core dump. However, I have found some strategies that helped me find bugs. These may also be useful in other porting projects and are discussed in this section.

In general, one difficulty is the fact that the GNU debugger is not available, which means that only the MINIX debugger `mdb` can be used. This utility is useful for analysing core dumps, but has fewer features for debugging running processes and cannot read symbol tables written by the GCC compiler. As a result, lines of the source code are not displayed with the

disassembled core file and addresses have to be looked up from the binary manually, using the `gnm` utility.

The approaches described in this chapter eventually lead me to find several errors in the emulation, including incorrect pointers being put in the generated code and a bug in the floating point rounding functions I wrote. The problems that were hardest to find and solve were the ones caused by MINIX itself. In particular, as has been noted in the section on the MINIX patches, the flags register is sometimes set incorrectly after MINIX 3.1.2 returns from a signal handler. Such errors require extensive analysis of core dumps resulting from them and hence some of the approaches mentioned here are aimed mostly at making those core dumps more readable. Another example is the `setitimer` implementation, which was disabled by a number of library functions that used the old `alarm` function. The methods described here are likely to also be useful for debugging other software if similar problems are encountered.

Causing crashes to occur early

One of the main difficulties with QEMU crashes is the fact that they often occur a some time after the original problem. If, for example, a relocation in the generated code is processed incorrectly, this often leads to a jump to or write at an incorrect address.

If QEMU incorrectly jumps to a address that contains valid code, then the result is quite unpredictable. Commonly, incorrect jumps are caused by uninitialized function pointers, causing jumps to the `NULL` pointer. While this address is invalid and its use causes a segmentation fault on many operating system, this is not the case in MINIX. On the x86 architecture, this is solved by placing a `UD2` instruction at the `NULL` pointer. This instruction is guaranteed to cause an invalid instruction exception, causing QEMU to be terminated and a core dump to be produced. A similar approach is possible for unused sections of the generated code buffer. Here I use the breakpoint instruction, which has the advantage that it is only a single byte long. This way, each address in the unused part of the buffer reliably causes a breakpoint exception, which is fatal and produces a core dump if QEMU is not being debugged.

The problem with incorrectly writing to some address is harder to detect. Although by default MINIX separates executable code from data by placing them in different segments, reducing the risk of overwriting code, this feature cannot be used by QEMU. The source code is not compatible with the ACK compiler and the GCC compiler is not able to create a MINIX binary which separates code and data. Moreover, this approach would not be suitable even if it were available because it makes the execution of newly generated code impossible. As described in the section on `NULL` pointers in MINIX, I created the `mprotect` system call for MINIX. This system call is normally available on operating systems which support virtual memory and allows one to restrict access to memory on a page-by-page basis. By protecting the executable code against writing, the likelihood of an incorrect write being detected is greatly increased. Both the static executable code and the dynamically generated code buffer can be write-protected this way, the latter being temporarily removed while generating new code. It should be noted that, although this feature was useful in the debugging phase, it has been removed in the final version of QEMU because the calls to `mprotect` caused a considerable slow-down and because `mprotect` is by default not available on MINIX. The changes that need to be made to MINIX to add the `mprotect` system call have been described

previously and the source code is found on the CD-ROM in the `minix/minix-3.1.2a-mprotect` directory..

Lack of double and triple fault

On the x86 architecture, it is possible for the processing of an exception to cause another exception. An example is the case in which a division by zero occurs and the interrupt handler for this exception has been swapped out to disk. In this case the processor generates a page fault to indicate that the handler must be loaded. If, however, there is some bug in the operating system that causes the page fault handler to also be unavailable, there is a serious problem. It is not possible to execute the handler that was supposed to deal with the original exception. If this issue is not addressed, an endless cycle of exceptions would be raised, each of which cannot be handled. This would freeze the processor.

To solve this issue, the x86 architecture implements double and triple faults. The double fault is an exception, handled just like the others, that indicates that an exception occurred while handling an exception. This is a last opportunity for the operating system to deal with the issue. If even this exception handler cannot be called, for example because the entire interrupt table is unavailable, then the x86 CPU causes a triple fault. The triple fault causes the computer to be reset.

QEMU 0.8.2 does not implement the x86 double and triple faults. As a consequence, serious operating system errors cause QEMU to enter into an infinite loop of raising exceptions. Although such errors are generally uncommon, it is quite possible that their real origin is an emulation error or an incompatibility between the operating system and QEMU. This is the case, for example, with Debian Lenny 5.0.1 Linux distribution which freezes QEMU while booting. This does not only occur on the MINIX port of QEMU, but even if unmodified QEMU 0.8.2 is run on Linux. Interrupts are handled using the `setjmp`/`longjmp` combination, allowing generated code to return to QEMU whenever an interrupt occurs. In this case, there is an endless loop in which an interrupt triggers a `longjmp` to return to the main loop, which restores the same situation that caused the interrupt to be raised. One difference between Linux and MINIX is that, on the former operating system, timer signals continue to be delivered while on the latter they are not. This means that, while on Linux only the virtual machine freezes and QEMU keeps accepting user input, on MINIX the QEMU process freezes entirely.

Since this is an issue with QEMU in general rather than specific to the MINIX port, I have not made much of an attempt to solve it; it is to be expected that this has already been done in newer versions of QEMU. It does, however, make debugging harder as it is not clear at first why the process freezes.

Logging system calls

One problem encountered when debugging QEMU was the fact that sometimes the delivery of `SIGALRM` signals would stop. Because the `setitimer` function had been used with an interval specified, signals should be delivered until explicitly stopped. To find out why the signal delivery was stopped, I created code to intercept system calls. This code is found in `ipchook.c`, included on the CD-ROM that comes with this thesis in the `qemu\qemu-0.8.2-minix\minix` directory.

To be able to intercept system calls, I replaced the `sendrec` function. This function is used by all system calls to pass the call number and parameters to the server process that handles it. The start of the function is replaced by a jump instruction pointing to a hook function. This way, each `sendrec` call is replaced by a call to the hook function. This function logs call details, derived from the message passed to `sendrec`, to a file. The name of this file is specified in the `IPCHOOK_FILE` environment variable or, if this variable is not specified, it is written to the standard error output. The message is then passed on to the original `sendrec` function, a copy of which has been preserved before patching it. This approach allows one to log system calls without changing any operating system component. To use it, one only needs to include the `ipchook.c` file in the project and call `ipchook_init` as soon as possible in the `main` function to install the hook.

Using this approach, it turned out that library code using the `alarm` function caused `setitimer` to be reset. These two functions are implemented in such a way that they share a single timer. If `alarm` is used in a library function, the timer settings are restored later on by calling `alarm` again. Since `alarm` cannot be used to restore the interval, only the one-shot component is restored. Moreover, the restored value is only accurate up to the second because the `alarm` function has a lower resolution than `setitimer`. This causes only a single more `SIGALRM` signal to be delivered. The solution was to adapt the library functions to use the new `setitimer` function rather than `alarm`, allowing the timer settings to be restored properly.

Making the core file more readable

QEMU is highly optimized for performance. For a large part, this is achieved by inlining functions and using register global variables. Inlined functions are not compiled as separate functions, but included at the code locations where they are called. This saves a function call operation and allows for more aggressive optimization by the compiler. However, it also greatly reduces the usefulness of stack traces and the readability of the assembly code. When analysing core dumps, function calls are generally good reference points for finding the correspondence between the C code and the disassembled code. If functions are inlined, this correspondence is lost.

Register global variables allow often-used variables to be permanently stored in registers rather than in memory. This can save many memory load and store operations if these variables are used often. QEMU saves as many registers of the guest CPU in host registers as possible, because these variables are often used. Although beneficial for performance, this makes debugging harder. On the x86 architecture, the `EBP` register is used for global variables while it would normally be used to store the location of the stack frame. Because of this, it is harder to reconstruct a stack trace if register global variables are enabled. Moreover, functions that do not know about the register global variables, such as library functions, store them on the stack and use the registers for other purposes. This means that their values are hard to find if the code using them was not running at the time the core dump was produced.

Because function inlining and register global variables make debugging harder, I disabled these features while debugging QEMU. Although this should reduce performance, the difference is not noticeable in normal use.

In addition to disabling optimization features, which is generally recommendable when debugging, I also added code to store relevant information on the execution state in global variables. By, for example, causing each code fragment used for code generation to store a

pointer to the original fragment, it is easily determined from the core file which instruction may have caused the error resulting in the core dump.

Parallel testing

One approach to detect problems that is especially suitable for ports is parallel testing. By testing the same program on the new platform and on one where it already works, one may be able to find out at which moment differences start to occur. As a reference platform, I have used Linux as this is the platform on which QEMU ran originally. This approach requires the programs to be as similar as possible between the two platforms, which I have achieved by causing most of the sections that are conditionally compiled only on MINIX to also compile on Linux and disabling Linux-specific sections. After getting QEMU to compile this way and enabling the deterministic mode, virtual machines running on MINIX and Linux should in principle behave identically. By logging relevant events, such as interrupts, and comparing the log files afterwards to find the first deviation between the two, one can find approximately where the error occurs. This lead me to find that a division by zero error occurred when Linux running on QEMU was starting Xorg, which happened when MINIX was the host platform but not when Linux was the host platform.

Further attempts to find the source of the division by zero on the virtual machine revealed that several other programs, including the GCC compiler, also exhibited the same problem. Fortunately, the latter also printed the code location on which the exception had occurred. Looking up the location in the GCC source code, this was revealed to be code which converts floating point numbers. After a few more experiments, it turned out that I had implemented a rounding function for floating point number incorrectly, causing the function in GCC to get impossible results and eventually divide by zero. Re-implementing the function in QEMU fixed the problem.

Profiling supported by GCC

The GCC compiler has features to make it possible to profile programs. If one enables this option, two changes are made to the compiled code: each function calls the `mcount` function at entry and a profiling timer is used. The `mcount` function counts how often each function is called from each location, allowing a call graph to be constructed. When the profiling timer causes a signal, the current code location is stored to find out how much of the time is spent in each part of the program code. Combined, these approaches allow one to estimate how much time each function uses when counting both the function itself and the functions it calls, which makes it possible to find performance bottlenecks.

On MINIX, however, the `mcount` function is not implemented in the library and profiling timers do not exist. As a result, profiling does not work out of the box. To be able to debug severe performance issues that occurred when sound was enabled, I implemented GCC profiling in MINIX. The main constraint was that the `setitimer` function I used only supports a single timer, which is already used by QEMU. Rather than further change the operating system to create another timer, which would be overly involved for this kind of problem, I found another way to estimate time spent in each function.

Besides the profiling option, GCC also supports the `-finstrument-functions` switch that causes the `__cyg_profile_func_enter` function to be called at the start of each function and

`__cyg_profile_func_exit` function just before returning. I implemented both functions in such a way that they keep track of the amount of time spent in each function. Although this causes more overhead than the default profiling option and provides data at the function level rather than the line level, it does not require changing the operating system and proved very useful for debugging the performance issue (which, as it turned out, was caused by polling for sound playback completion occurring much too often).

Using MINIX' information server

Yet another useful source of debug information that is MINIX' information server. The function keys can be used to ask this process to dump system information to the console. For debugging QEMU, the shift-F2 combination which prints a list about the status with regard to signal handling for each process turned out to be valuable.

The problem in this case was the fact that QEMU would hang while performing benchmarks with Linux as a guest operating system. This happened consistently some 30-40 seconds after starting the benchmarks. This was quite surprising, as it happened during a benchmark that would only check the time in a loop and would terminate after a specific amount of time had passed. The information server revealed that, at the time QEMU stopped doing useful work, the `SIGALRM` signal had been blocked. As a result, no timer interrupts were sent to the guest, causing its time to stand still and the benchmark to continue forever. Moreover, the user interface is also updated from the timer signal handler, creating the appearance that the QEMU process was hanging.

Unfortunately, this problem proved to be hard to address. If debugging output was added to QEMU, for example by writing to the standard output or files or by sending signals, QEMU would not hang but rather terminate. Moreover the X server, the networking server and the ethernet card driver all came down with it and no core dump was saved. This suggests that there is a problem in MINIX, as it should not normally be possible for a process to bring all of these processes down with it.

I have been able to perform tests to determine whether the `SIGALRM` signal handler returns correctly and whether it restores the correct signal mask; since a signal is blocked while its signal handler is running, these factors could explain that a signal becomes blocked. It turned out, however, that the signal handler always returns and that the correct signal mask is stored on the stack and passed to the `sigreturn` function. This is the case even at the end of the signal handler, ruling out the possibility that an incorrect pointer operation somewhere corrupts the signal mask.

As a workaround, I have added the following function when QEMU is compiled on MINIX:

```
void testsigmask(void)
{
	static int reported;
	sigset_t set, oset;

	/* in MINIX, signals sometimes get blocked incorrectly; check
	 * regularly to remedy this
	 */
	if (sigemptyset(&set) < 0 ||
		sigaddset(&set, SIGALRM) < 0 ||
		sigprocmask(SIG_UNBLOCK, &set, &oset) < 0)
	{
```

```
                perror("sigprocmask");
                exit(-1);
        }

        /* warn about blocked signal (only first time) */
        if (!reported && sigismember(&oset, SIGALRM))
        {
                fprintf(stderr, "warning: re-enabled SIGALRM\n");
                fflush(stderr);
                reported = 1;
        }
}
```

This function unblocks the `SIGALRM` signal and warns the first time it has become blocked. It is called regularly to ensure that the signal is unblocked in time. This fixes the problem, but only for a short period of time. While a `SIGALRM` signal is delivered 60 times per second, it becomes re-blocked approximately 10 to 15 times per second after the first occurrence.

As I was incapable of finding the source of this problem in the time available for this thesis, this is something that will need to be looked into further. For now, the workaround appears sufficient but the underlying problem is not fixed. It appears that the problem is to be found in MINIX as providing output from a signal handler in a user process should not normally affect other processes and drivers, but it is quite possible that it has to do with the `setitimer` function that was added for this project.

3.7 - Testing QEMU

To determine whether QEMU actually works, I tested it with a number of guest images. The following operating systems and bootable CD-ROMs that do not use an operating system have been tested:

- GpartEd (partition editor live CD running Linux and X)
- Linux 2.6.18 installed on hard disk (Debian 4.0r3 Etch with Xfce and Firefox);
- Linux 2.6.20 installed on hard disk (small test image provided by on the QEMU website);
- Linux 2.6.27 installed on hard disk (minimalist Slackware 12.2 with GCC);
- Linux 2.6.27 live CD (Damn Small Linux 4.4.10);
- Memtest (memory test utility which boots directly from CD);
- MINIX 3.1.2a;
- MINIX 3.1.4;
- Windows 95;
- Windows 98 SE.

I have eventually managed to make each of these images boot. Networking works on each of the images except for Windows 95, for which I could not obtain an appropriate driver. It is possible to use Internet Explorer on Windows 98 SE or Mozilla Firefox on Linux to graphically browse the internet, something which at the time of writing is not yet possible on MINIX directly. When using Internet Explorer on Windows 98 SE on the test system, speed is acceptable for normal use, while Firefox on Linux is on the slow side but might be useful on

faster hardware. These tests were performed on a Pentium M 1.6 GHz laptop computer with 512 MB of memory, of which 128 MB was assigned to the virtual machine.

It should be noted that the clock frequency is an issue for Windows 98 SE. The `esdi_506.pdr` driver that provides support for large disks hangs on a normal boot. Booting is still possible in safe mode, in which the driver can be deleted from the `%windir %\system\iosubsys` directory to allow normal booting. Using debug output from QEMU, I noticed that Windows 98 SE generally uses a clock frequency of 200 Hz and occasionally changes it to much higher values. This suggests that the fact that MINIX can deliver interrupts at only 60 Hz may be a problem and leads to an alternative solution: the problem can be resolved by setting the MINIX `HZ` constant to a higher value. Windows 98 works when `HZ` is set to at least 150. I also tried 120 Hz, but this is insufficient. Compiling QEMU in deterministic mode also solves the problem, as `setitimer` is no longer used in this case and the MINIX clock frequency no longer influences QEMU. This is a clear indication that the problem is caused by the unreliability of delivery of clock events on MINIX due to its low default clock frequency.

Besides the images listed before, I have also tried a more recent version of the Debian distribution, Lenny, which comes with a Linux 2.6.26 kernel. This version does not work. It causes an invalid memory access while booting and hangs because the resulting interrupt cannot be handled yet at that time. It should be noted, however, that the same behaviour is found when running it on QEMU 0.8.2 on Linux, even when using the binary version from the Debian package management system. I have concluded that the problem is an incompatibility between QEMU 0.8.2 and this specific version of Linux, in particular since the newer kernel that comes with Slackware does work. Hence the problem is not caused by the QEMU port and I have not looked into it any deeper.

3.8 - Discussion

All in all, I found that QEMU can be ported to MINIX, but this does require making some changes to the operating system. Moreover, due to missing features and bugs in MINIX as well as some flaws in QEMU, porting is harder than it needs to be. These flaws include in particular non-standards compliant and compiler-dependent code and in some cases lack of error checking. Programs which have already been ported to more platforms, such as the SDL library, avoid such mistakes and pose much less of a problem. Being aware of possible issues in advance might save considerable effort when porting software to MINIX. Those who consider undertaking such a project would do well to check other people's experiences in advance. This chapter has in part been aimed at providing people with such information.

There are some ways in which changes to MINIX could contribute to easier porting. In particular, virtual memory could prevent problems due to lack of memory. Once virtual memory is implemented, it could also provide an implementation of `mprotect` and protect the `NULL` pointer in order to make debugging easier. For QEMU, the lack of the `setitimer` function has been a major issue. As this is a standardized function, it is to be expected that its inclusion in MINIX would also aid other porting projects. A full implementation could also provide a profiling timer, which would also make debugging easier. A variable clock frequency, for example by scheduling clock ticks based on need rather than with fixed periods (a tickless kernel), could further increase the resolution and hence the usefulness of this function.

Making the MINIX libraries more complete would further aid porting. Important functions I found to be missing are the floating point rounding functions, `realpath`, 64-bit support for the `printf` family and the `send` and `recv` socket functions. Some of these functions have been implemented to be able to port QEMU and could be adapted for inclusion in the MINIX libraries. Another major improvement to the sockets library would be to treat the loopback address (127.0.0.1) as a separate networking device rather than a special case. This would make porting easier, make the X server easier to use without a network and could make the system safer if this feature is properly used by local servers.

Finally, porting the GNU Debugger (GDB) would also make porting easier. It is a debugging tool that those porting software to MINIX are likely to already be familiar with and it provides many advanced debugging features. Moreover, unlike the MINIX Debugger (MDB), it would be capable of reading the symbol tables in the binary format produced by GCC. This makes analysing core dumps considerably easier, especially for those with poor assembly skills.

4 - How to use QEMU on MINIX

4.1 - Installing QEMU on MINIX

What has to be done

Unfortunately, compiling and installing QEMU on MINIX is not as simple as just downloading and unpacking a package. To be able to compile QEMU, a number of packages it depends on must be installed first. This includes a number of packages that can be installed with the `packman` utility as well as the newly ported libSDL package. To be able to install QEMU on current MINIX versions one has to patch and recompile MINIX, as the current release version of MINIX (version 3.1.2a) lacks some POSIX features needed by QEMU and has a bug that strongly affects it.

To allow for easy installation, I created a shell script that performs all actions needed to install required packages, patch MINIX and compile and install libSDL and QEMU. Additionally, this script is capable of creating and installing binary packages for libSDL and QEMU for use with `packman`. This will be particularly useful once the MINIX patches have been incorporated into the next release version of MINIX, removing the need to patch MINIX. The install script is found in the `/qemu/install` directory on the CD-ROM included with this thesis. More information on the contents of this CD-ROM can be found in Appendix A.

I will first discuss what has to be done to install QEMU using the installation script. Next, I provide instructions to install QEMU manually. This may be needed for safe installation on modified MINIX systems and it provides more insight in the inner workings of the installation script.

Installing using the installation script

The simplest way to patch MINIX and install the binary distribution of QEMU is to run the install script without any parameters. As the script involves a MINIX patch, I strongly recommend that the script be run only on unmodified MINIX 3.1.2a. Please back up any valuable data before running it. I do not take responsibility for any damage the script may cause, use it at your own risk. One can download and execute the install script in the following way:

```
urlget http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2-install.sh.bz2 \
  | bunzip2 > qemu-0.8.2-install.sh
sh qemu-0.8.2-install.sh
```

After confirming that the user wants MINIX to be patched and recompiled, the install script performs all actions needed to install QEMU. It also creates a file in the current working directory named `qemu-0.8.2-install.log`. This file contains an overview of everything that was done and the results. It can be used to debug in case something went wrong or simply to find out what changes were made by the script.

To control which actions are performed, a number of arguments can be specified. Each argument specifies a specific action to be included and may imply other actions which are

needed for the specified action to be executed successfully. As such, the way the install script is invoked is similar to the way makefiles work. The following actions can be specified:

`clean`	Uninstalls QEMU and libSDL and deletes all source and temporary files;
`packages`	Installs packages required to compile QEMU and libSDL;
`patchmnx`	Patches MINIX to include the changes required for QEMU;
`buildmnx`	Compiles and installs MINIX;
`buildsdl`	Downloads source code for libSDL, compiles it and installs it;
`installsdl`	Downloads libSDL binary package and installs it;
`makepacksdl`	Creates a binary package for libSDL from downloaded source code;
`build`	Downloads source code for QEMU, compiles it and installs it;
`install`	Downloads QEMU binary package and installs it;
`makepack`	Creates a binary package for QEMU from downloaded source code.

The following example downloads and patches a fresh copy of the QEMU source code, compiles it and installs it from source:

```
sh qemu-0.8.2-install.sh clean build
```

The actions are executed in the sequence in which they were listed. If nothing is specified, the default is to assume that only `install` was specified. The `buildsdl`, `makepacksdl`, `build` and `makepack` commands cause source files to be downloaded. Source code for QEMU is stored in `/usr/src/qemu-0.8.2` and source code for libSDL in `/usr/src/SDL-1.2.13`. The files in these directories are always overwritten, so the install script causes the distributed versions to be installed. One can use the `build.minix` scripts provided with these programs to compile these programs after making modifications.

The script needs to download a number of files, and it is therefore easiest to use on a MINIX machine with a working Internet connection. If there is no way in which MINIX can connect to the Internet, one may instead download the files elsewhere and copy them into the `/usr/tmp/packages`. The list of files to download is found below:

- Always required
  - http://www.few.vu.nl/~dcvmoole/minix/packages/patch-2.5.4.tar.bz2
  - http://www.few.vu.nl/~vdkouwe/qemu/downloads/minix-3.1.2-qemu.patch.bz2
  - http://www.minix3.org/packages/gcc-3.4.3.tar.bz2
- Required when installing QEMU binaries (`install` specified, `build` and `makepack` missing)
  - http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2.tar.bz2
- Required when installing libSDL binaries (`installsdl`, `build` or `makepack` specified, `buildsdl` and `makepacksdl` missing)
  - http://www.few.vu.nl/~vdkouwe/qemu/downloads/SDL-1.2.13.tar.bz2
- Required when compiling QEMU from source (`build` or `makepack` specified)
  - http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2.tar.gz

    - http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2-minix.patch.bz2
    - http://www.minix3.org/packages/grep-2.5.1a.tar.bz2
    - http://www.minix3.org/packages/gzip-1.2.4.tar.bz2
    - http://www.minix3.org/packages/ncurses-5.5.tar.bz2
    - http://www.minix3.org/packages/pdksh-5.2.14.tar.bz2
    - http://www.minix3.org/packages/X11R6.8.2.tar.bz2
- Required when compiling libSDL from source (`buildsdl` or `makepacksdl` specified)
    - http://www.few.vu.nl/~vdkouwe/qemu/downloads/SDL-1.2.13-minix.patch.bz2
    - http://www.libsdl.org/release/SDL-1.2.13.tar.gz

The packages that are downloaded from http://www.few.vu.nl/~vdkouwe/qemu/downloads/ can also be copied from the `/downloads` directory of the CD-ROM that comes with this thesis, while `packman` can be used to install some of the other packages from the MINIX CD-ROM. Please note that the former CD-ROM uses ISO format and therefore cannot be mounted in MINIX, but its files can be read using the `isoread` tool.

Installing manually

Prerequisites

The install script installs a number of packages to be able to compile MINIX and libSDL. This section about manual installation assumes that the MINIX system fulfils all prerequisites in advance. To be able to compile and run QEMU, one will need to have:

- Unmodified MINIX 3.1.2a. When running a different or modified version, merging in the operating system patches may have to be done manually. MINIX is available from http://www.minix3.org/download/.
- A functional Internet connection on the MINIX machine to download the source code and patches. If this is not possible, some alternative approach to transfer the files will be needed.
- The `gcc-3.4.3` package, needed to compile QEMU, is available through the `packman` command.
- The `grep-2.5.1a` package, needed by the libSDL configure script, is available through the `packman` command.
- The `gzip-1.2.4` package, needed to extract source code, is available through the `packman` command.
- The `ncurses-5.5` package, needed to for curses support, is available through the `packman` command.
- The `pdksh-5.2.14` package, needed to run the QEMU configure script, is available through the `packman` command.
- The `X11R6.8.2` package, needed to compile SDL, is available through the `packman` command.

Installing GNU Patch

Besides these default packages, a program to apply source code patches is needed. Unfortunately, the `diff` and `patch` programs included with MINIX cannot create and apply reliable multi-directory patches. For this reason I used GNU Diff to create all patches available here. GNU Patch is needed to apply them. GNU patch 2.5.4 has been ported by David van Moolenbroek and is available from http://www.few.vu.nl/~dcvmoole/minix/. It can be installed using the following commands on the console:

```
urlget http://www.few.vu.nl/~dcvmoole/minix/packages/patch-2.5.4.tar.bz2 | \
  bunzip2 | \
  tar xf - (ignore "File exists" errors)
chmem =1048576 /usr/gnu/bin/patch
```

GNU Patch uses `/tmp` for temporary storage. Since this directory is on the root partition, which is only 16MB, it will run out of disk space when performing patches. To fix this, one can run the following commands:

```
rm -r /tmp
ln -s /usr/tmp /tmp
```

This links `/tmp` with the temporary folder on the `/usr` partition, which tends to be much larger than the root partition. This solves the problem, but it should be noted that this merges the two folders for temporary files. If this is undesirable, the old situation can be restored afterwards by deleting and re-creating `/tmp`. Another possibility is to create a new folder on the `/usr` sub-partition and permanently link `/tmp` to that folder.

Patching MINIX

As has been explained, QEMU cannot run on bare MINIX 3.1.2a, which lacks features that QEMU requires and contains a bug that strongly affects it. The following issues are addressed by the MINIX patches included with the ported QEMU:

- A bug causes the CPU flags register to not be restored after a signal handler executed, resulting in unpredictable crashes. These crashes are uncommon in most programs, but common in QEMU since it processes `SIGALRM` signals at a high rate. This patch is taken from the Subversion repository. I would like to thank Jens de Smit for pointing this out in a post to comp.os.minix.
- Differences between the ACK and GNU definitions in the include files. This causes GNU library compilation to fail. I copied definitions from the ACK headers to the GCC ones, removing `const` from the `putenv` declaration in `stdlib.h` and adding definitions for `_SC_PAGE_SIZE` and `_SC_PAGESIZE` to `unistd.h`.
- QEMU requires the `setitimer` function, which is not available in MINIX. This function is added by the patch. It was implemented by David van Moolenbroek and I made some fixes to avoid the MINIX libraries from resetting it.
- MINIX 3.1.2a does not allow the `select` function to be used on an ethernet device (`/dev/eth`). This is needed to wait for incoming packets in TUN networking and in the `qemu-vswitch` program. I added this functionality, and it has since been added to the Subversion repository.
- MINIX does not implement the `pread64` and `pwrite64` functions which allow reading from and writing to files beyond the 2^{32} byte boundary. This is never needed for regular

files, which cannot be that large on MINIX, but is required to be able to access large disk partitions in a safe way.

The modified code can be found in the `/minix` directory of the CD-ROM that comes with this thesis. More information on the contents of the CD-ROM can be found in Appendix A. Instructions are provided below to apply the patches to MINIX. The patch file should work without errors on unmodified MINIX 3.1.2a, but the patch program is likely to need manual assistance if a different or modified version of MINIX is used.

First, one needs to download the patch and apply it using GNU Patch. This is done using the commands listed below:

```
urlget http://www.few.vu.nl/~vdkouwe/qemu/downloads/minix-3.1.2-qemu.patch.bz2 | \
  bunzip2 | \
  /usr/gnu/bin/patch -p0
```

After applying the patch, MINIX needs to be rebuilt. The modifications are only effective after a reboot. Unfortunately, the `reboot` program will cease to work after recompilation, as its libraries have changed but the MINIX kernel and servers running are still the same. For this reason, one needs to make a temporary copy of `/usr/bin/reboot` to be able to reboot without errors.

```
cd /usr/src/tools
make install
cd /usr/src/lib
make install-ack
PATH=$PATH:/usr/gnu/bin make install-gnu
cd /usr/src/commands
cp /usr/bin/reboot /tmp
make clean install
/tmp/reboot
```

Installing libSDL

QEMU uses libSDL as a portable way to output graphics. To be able to get graphics output from QEMU, I ported it to MINIX as well. This library is not strictly necessary as I included support for text output using the Curses library, but the ability to output graphics makes QEMU much more useful. One can choose between both output methods using command line switches, so there should be no need to disable graphics output at compile time.

To be able to compile and install libSDL, one first needs to download the source code. The source code for the MINIX port is provided as a patch to the original SDL code. The patched code is obtained using the following commands:

```
urlget http://www.libsdl.org/release/SDL-1.2.13.tar.gz | \
  gunzip | \
  tar xf -
urlget http://www.few.vu.nl/~vdkouwe/qemu/downloads/SDL-1.2.13-
minix.patch.bz2 | \
  bunzip2 | \
  /usr/gnu/bin/patch -p0
```

The next step is to build and install libSDL. To run its configure script, the shell needs to have sufficient memory available. As MINIX does not implement virtual memory, this memory needs to be assigned in advance using the `chmem` command. It should be noted that the `chmem` commands below should only be run if a higher value was not assigned previously; if one reduces the amount of memory allocated, applications that require the higher value may fail

afterwards. An easy way to check the amount of memory is running `chmem +0 <filename>`, replacing `<filename>` with the name of the file to be checked. The commands I use to compile libSDL are as follows:

```
chmem =1048576 /bin/sh
chmem =2097152 /usr/local/bin/ksh
cd SDL-1.2.13
sh build.minix
```

Installing QEMU itself

Now that all prerequisites have been installed and the operating system has been patched, QEMU can be compiled and installed. The first step is download and patch it, just like the way this was done for libSDL:

```
urlget http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2.tar.gz | \
  gunzip | \
  tar xf -
urlget http://www.few.vu.nl/~vdkouwe/qemu/downloads/qemu-0.8.2-
minix.patch.bz2 | \
  bunzip2 | \
  /usr/gnu/bin/patch -p0
```

Finally, the patched QEMU can be built and installed using the `build.minix` shell script. Concerning the use of `chmem` the same remarks apply as in the previous sub-section.

```
chmem =33554432 /usr/gnu/bin/gld
cd qemu-0.8.2
sh build.minix
```

If all of this succeeded, QEMU has been installed and can be run. Before continuing, one may want to restore the `/tmp` directory to its original state:

```
rm /tmp
mkdir /tmp
chmod 777 /tmp
```

4.2 - Running QEMU on MINIX

Running the pre-made disk images

For the reader to be able to test QEMU for him- or herself, this section describes how to run QEMU on the MINIX operating system. There is little difference between the way this is done on MINIX and Linux, so one who is already familiar with QEMU may want to skip this section. Most differences relate to networking, but the default user mode networking does not differ between MINIX and other host operating systems. The considerations to be made when setting up the network have been discussed previously (in section 3.4) and are therefore not repeated here.

The QEMU website provides a small and simple Linux image to allow one to test QEMU. It contains the Linux 2.6.20 kernel and a number of utilities to test it. This disk image can be run as follows:

```
urlget http://www.qemu.org/linux-0.2.img.bz2 | bunzip2 > linux-0.2.img
qemu -hda linux-0.2.img
```

Only the location of the disk image is specified, so QEMU uses the default for all other settings. Other disk images, such as the ones on the CD-ROM, can be specified in the same

way. This requires that X is available and that QEMU has been assigned sufficient memory to accommodate a 128 MB buffer of guest machine memory. This may be a problem when using a machine with little RAM because MINIX 3.1.2 does not implement virtual memory.

To run QEMU without X, add `-curses` to the command line. This causes the Curses library rather than X to be used to display text output from the virtual machine. To avoid an "Error opening terminal" error message, be sure to install the library first. This can be done by selecting the `ncurses-5.5` package in `packman`. An alternative is to run X on another machine and specify where it is running using the `DISPLAY` environment variable. The line below, for example, causes QEMU to use the X server which is running on the computer with IP address 192.168.0.10 at X port 0:

```
DISPLAY=192.168.0.10:0 qemu -hda linux-0.2.img
```

QEMU displays a message "Could not initialize SDL – exiting" if the `DISPLAY` setting is missing or incorrect. It should be noted that, while this approach if effective in saving memory, it seriously hampers performance, especially if a slow or high-latency network is used.

In some cases, depending on the video hardware used, it may be possible to substantially reduce the memory used by X simply by changing its `chmem` value. Its default value is 512 MB and such a high value should not be needed on most hardware. If QEMU does not start due to lack of memory, try changing the `chmem` value for `/usr/X11R6/bin/X` until a low value which still allows X to start is found.

To reduce the amount of memory needed by QEMU itself, use the `-m` switch to specify how much RAM is provided to the guest machine in MB. If, for example, one specifies `-m 64` then the guest machine gets to use only 64 MB of memory and QEMU's memory footprint can be reduced. The `chmem` command should then be used to avoid the memory from being allocated to QEMU. The default set by the compile script is slightly over 144 MB, which is appropriate only if the guest machine has 128 MB. If 64 MB is assigned to the guest instead, then I would recommend using the line below to reduce QEMU's memory footprint to 80 MB.

```
chmem =83886080 /usr/local/bin/qemu
```

If the `chmem` value is too large, memory is wasted and QEMU may not start due to a lack of free memory. If it is too small, QEMU exits with an error message and provides a recommendation for the amount of memory to be assigned to QEMU. When in doubt, assign a small value to trigger this error message and obtain an estimation. The estimate is computed based on the total RAM, video and BIOS memory available to the virtual machine plus 8 MB for other buffers allocated by QEMU as well as its stack. In my experience this has always been sufficient, although it is hard to estimate the exact amount of memory used. There may be configurations in which it is not enough and a higher value should be chosen.

Setting up a new virtual machine

Setting up a new virtual machine is not really different between MINIX and Linux; a short introduction is provided in this section for those unfamiliar with QEMU or with MINIX disk naming. With QEMU, all configuration is done on the command line when starting a virtual machine, so setting up the virtual machine involves only preparing the disk image. This means first creating the file in which it is stored and then installing an operating system on it.

To create a disk image, one uses the `qemu-img` tool which comes with QEMU. The size of a virtual disk is fixed after it has been created, so this needs to be decided on in advance. One

also needs to choose a format to store the disk image in, as QEMU provides several possibilities. These formats differ in the features offered and several formats are mainly offered for compatibility with other tools. The simplest type is "image," which means that the specified file literally stores the contents of the disk. This type takes up all space available to the virtual machine from the moment it is created, while other formats generally only store disk blocks that have been written to. This means the file storing the disk image is much smaller initially, but grows as the guest operating system uses it. The native QEMU formats doing this are called COW (copy-on-write) and QCOW (quick copy-on-write). The former type is not supported on MINIX. It is implemented by mapping the image file to memory, which is not possible in MINIX. This means that, unless compatibility with some other program is desired, QCOW is the most suitable format. Hence, the following command would be a typical way to create a 2 GB disk image named `my-disk-image.qcow` on MINIX:

```
qemu-img create -f qcow my-disk-image.qcow 2G
```

COW format files created on other operating systems can be converted to QCOW using `qemu-img` to allow them to be used on MINIX. This has to be done on an operating system on which they are supported, which excludes MINIX and Windows. The COW format file `my-disk-image.cow` can be converted to QCOW using this command:

```
qemu-img convert -f cow -O qcow my-disk-image.cow my-disk-image.qcow
```

Like on other Unix-like operating systems, it is possible to references disks and disk partitions as files and this way they too can be used as image files and can be made available directly to virtual machines. This is particularly important when installing operating systems, as these are often provided on CD-ROM. Moreover, using a physical partition allows one to share it with other operating systems on the host machine and to bypass the 4 GB limit of MINIX partitions. Unfortunately, MINIX does not support DVDs so those cannot be used in QEMU.

Disks are identified in MINIX as `/dev/c`*n*`d`*m*, where *n* indicates the number of the disk controller (either 0 or 1) and *m* the position of the disk on that controller. My experience is that, in a system with a single disk and a single CD-ROM drive, `/dev/c0d0` generally refers to the disk and `/dev/c0d2` refers to the CD-ROM drive. To boot from the CD-ROM to allow the operating system on it to be installed, one would then use the following command line:

```
qemu -hda my-disk-image.qcow -cdrom /dev/c0d2 -boot d
```

5 - Performance measurements

5.1 - Methodology

Measuring QEMU performance

To be able to determine how well QEMU performs on MINIX and why, I created a benchmark program which conducts a variety of tests. Each of these tests considers a different aspect of performance. The advantage of this approach is that it allows me to determine where in MINIX the bottlenecks are, so that suggestions can be made to improve the situation. It is most convenient to perform the timing measurements at the guest machine, as this is also where the benchmark code is run. When performing many measurements with several guest images as well as multiple configurations for QEMU, however, it is more suitable to store the results on the host as measurements from the virtual machines have to be combined. This means that there must be some way for the host to tell the guest to start benchmarking and some way for the guest to be able to send back the results. Moreover, it is convenient if changes to the benchmark program need not be applied to each guest image separately to keep the system flexible.

To measure at the guest and store results at the host in a flexible way, I have created three separate programs and a shell script which together perform the benchmarking. These programs have been included in the `/utils/benchdriver` and `/utils/benchmark` directories on the CD-ROM. The situation is shown in Figure 13. The `benchdriver` program coordinates the various tests. For each configuration and each guest image, it compiles QEMU and extracts the guest image. It then launches QEMU. The guest images have been set up in such a way that another program, `benchdriverserver`, is run at boot time. This is the only program that has to be inside the guest disk images. It has been kept as simple as possible so that the need for changes is unlikely. Although it would have been possible to use the Telnet daemon instead and this program is available by default for both Linux and MINIX, these operating systems have different implementations. This would add an unnecessary element of difference between them; overhead in the Telnet server might affect benchmark results.

While QEMU is booting the guest image, `benchdriver` regularly attempts to connect to `benchdriverserver`, which acts as a TCP server. Connecting in the other direction (from guest to host) would not require this kind of polling behaviour, but would require that the address of the host is known when the guest image is made. As this depends on the computer the benchmark is run on as well as on QEMU's networking settings, this is inconvenient.

Once `benchdriver` on the host has connected to `benchdriverserver`, it sends the `benchdriver.sh` shell script over the TCP connection. The guest runs this shell script. It downloads the source code of the `benchmark` program using FTP, compiles it and finally runs it. Using the script rather than performing these operations directly in `benchdriverserver` allows for much flexibility. Even though MINIX uses the ACK compiler by default, I have decided to compile the benchmark program using the GCC 4 compiler on both MINIX and Linux. Previous attempts where ACK was used showed that MINIX was put at a serious disadvantage, since GCC performs more extensive optimizations that had considerable impact on several of the benchmarks.

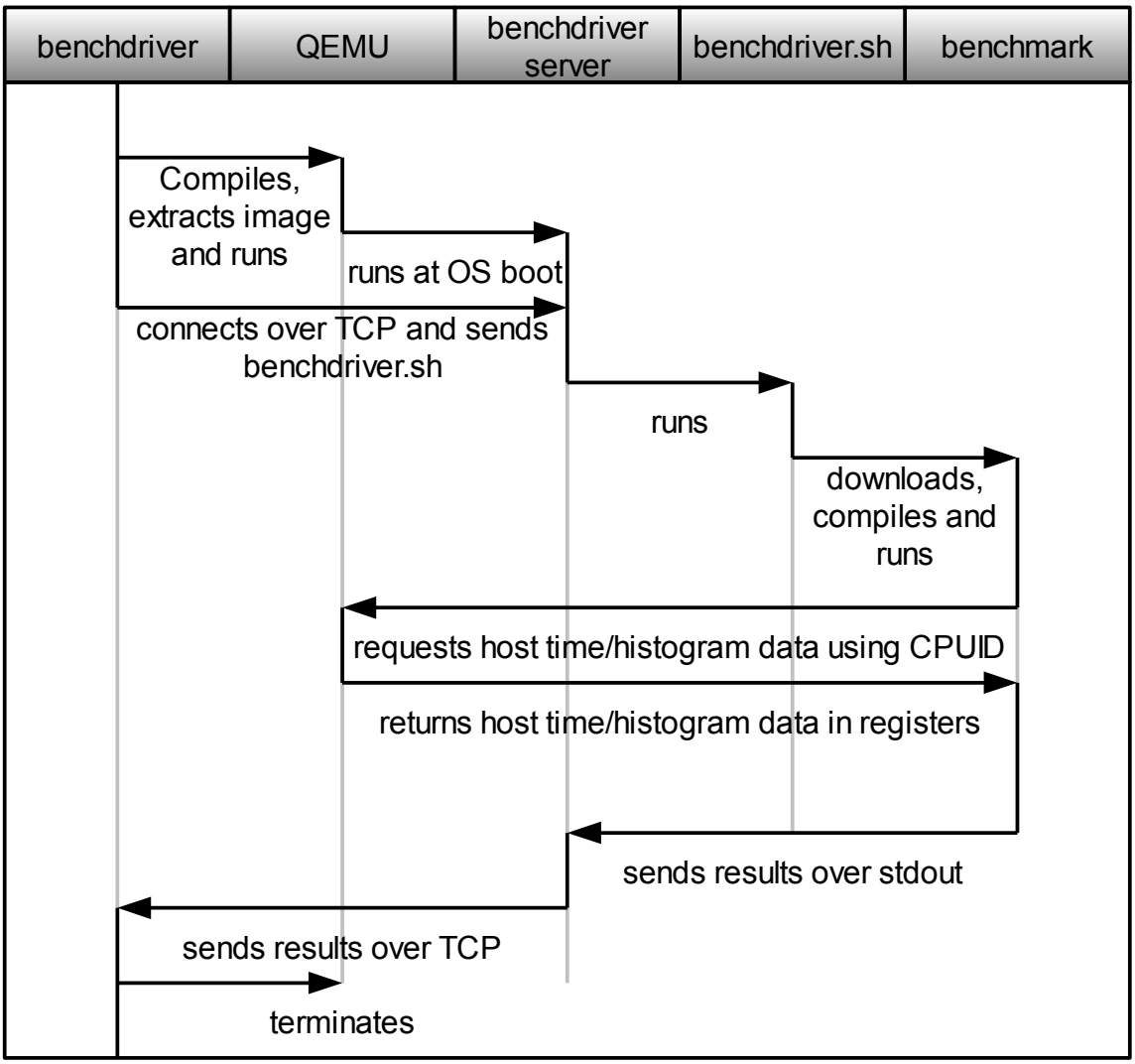

Figure 13: Interaction between processes involved in measuring performance.

Because the platforms tested have slightly different versions of GCC 4, I have also used the `objdump` utility to get the assembly code for the benchmark program and compared them between the different versions. I have considered in particular the innermost loops of the various benchmarks, which determine nearly all of the runtime. I have found only a few important differences. The most important difference is that the Linux version of GCC reduces memory access by loading a memory location in a register, manipulating it there and only writes back the result to memory when all operations have been performed. This provides a substantial performance benefit for some benchmarks. To avoid this optimization, I marked variables with the `volatile` keyword, forcing the compiler to generate the appropriate loads and stores on every access. Another difference was in the `flow_conditional` benchmark, where a special instruction sequence is used to avoid conditional jumps. I have made this optimization impossible by changing the action taken if the condition tested for is true; the function originally incremented a global variable and now adds a larger number.

When the `benchmark` program itself is run, it performs various tests which are listed in Table 5. Multiple tests are performed to find out which aspects of the emulation are particularly slow. Most benchmarks focus on a specific aspect of the CPU, the operating system or some hardware device. The exceptions to this are `calibrate_busy`, `calibrate_idle` and `blended_compileminix`. The former two run for a fixed amount of time and are therefore not

Benchmark	Description
calibrate_busy	Busy wait for a fixed time, determines processor frequency
calibrate_idle	Idle wait for a fixed time, determines whether processor frequency is fixed
arithmetic_int	Integer arithmetic
arithmetic_float	Floating point arithmetic
flow_call	Recursive calls
flow_conditional	Conditional jumps
flow_exception	Handle signals caused by division by zero
flow_jumptable	Computed jumps
flow_syscall	System calls
flow_taskswitch	Waiting for child process
io_disk_read_random	Read from disk, scattered small blocks
io_disk_read_sequential	Read from disk, sequential large block
io_disk_write_random	Write to disk, scattered small blocks
io_disk_write_sequential	Write to disk, sequential large block
io_display	Display text on the primary console
io_network_latency	Connect to an external computer over TCP
io_network_throughput	Send data to and receive data from an external computer over TCP
memory_load_random	Read from memory, scattered small blocks
memory_load_sequential	Read from memory, sequential large block
memory_store_random	Write to memory, scattered small blocks
memory_store_sequential	Write to memory, sequential large block
blended_compileminix	Recompile MINIX

Table 5: Benchmarks performed by the Benchmark program

truly benchmarks. Instead, they are used to determine whether time is most appropriately measured in CPU cycles or in milliseconds and provide a conversion factor between the two. Why this is needed is described in more detail in the next paragraph. The `blended_compileminix` benchmark compiles MINIX and serves to give an overall overview of performance on a real-life task.

To obtain an accurate measurement it is desirable that the benchmarks take a sufficient amount of time; this causes overhead events, such as clock ticks, to be averaged out over the time period. I have attempted to choose loop bounds in such a way that, whenever possible, benchmarks should take at least 100 milliseconds on the fastest configuration and at most 15 seconds on the slowest one. The host time is measured before and after each test. The host time rather than the guest time is used because it reflect real time more accurately, especially in deterministic mode. I added operations to the `CPUID` instruction to allow the guest machine to request the host time, as was described in more detail in section 3.4. The host time is measured in both milliseconds (using the `gettimeofday` function) and CPU cycles (using the `RDTSC` instruction). The latter allows for measuring at a much higher resolution than the former, reducing measurement error, but does not always reflect real time. In particular, the `calibrate_busy` and `calibrate_idle` benchmarks show that, when Linux is the host operating system, the clock frequency is strongly reduced when the CPU is idle. This is most likely done to save energy. This means that, when Linux is used as the host operating system, CPU cycles are not a reliable way to measure the time taken by tests that include idle time. The results show that these tests are `calibrate_idle`, `io_network_latency`, `io_network_throughput` and `blended_compileminix`. For this reason, I measure both and use the milliseconds only in those cases where CPU cycles are not reliable. The `calibrate_busy` benchmark keeps the CPU busy

for a fixed time, which allows the clock frequency to be computed and used as a conversion factor.

When benchmarking has been completed, the results are sent back to `benchdriverserver` using the standard output, which in turn forwards them to `benchdriver` over the TCP connection that was established in the beginning. Any text written to the error output is also sent over this connection to make it easier to diagnose problems. These two output streams are multiplexed over the same connection by prefixing each chunk with a channel number. `benchdriver` stores channel 1, the standard output of Benchmark, in a text file and sends incoming text on channel 2, the error output, to its own error output. After `benchmark` has terminated, `benchdriverserver` notifies `benchdriver`. `benchdriver` then terminates QEMU, cleans up the guest image and continues with the next configuration.

To obtain reliable measurements, it is desirable to perform multiple measurements and compute the average. This evens out disturbances caused by variation in such factors as response time in hardware and the network, the number of timer interrupts occurring during the benchmark (on both the host and the guest), differences in the way the benchmark process is scheduled on the guest and the way QEMU is scheduled on the host. An additional advantage is that, with multiple separate measurements, it is possible to estimate the variation in benchmark results. Using this information it is possible to distinguish between meaningful differences and differences potentially caused by measurement errors.

To obtain multiple measurements, the entire benchmark suite is run five times on each platform. The benchmark suite may run individual benchmarks multiple times if they are expected to be inaccurate due to a short runtime. Each benchmark is repeated until at least one minute has been spent on it. Hence, each benchmark is run at least five times but may be run more often for those configurations on which it is fast.

Benchmarks are run several times in succession, which may have a impact on performance due to caching. I found this effect in particular with the disk benchmarks when running on Linux, since Linux uses a large part of free memory for disk caching. As cached blocks are retrieved in a small fraction of the time needed for uncached blocks, performance was substantially higher in the later runs. I addressed this by clearing the disk cache before running these benchmarks and by comparing only measurements from the same run. For example, MINIX can be compiled multiple times within a minute when running natively but this generally takes more than a minute when running on QEMU. If the later runs are faster then it would be unfair to compare the multiple native runs with a single emulated run. Hence, only the first run would be compared in this case.

Measuring impact of the `HZ` constant

On Linux, QEMU sets the operating system clock frequency to 1024 Hz if possible. This allows for more accurate emulation, as guest machine timer interrupts can be delivered at most at the host timer frequency. MINIX uses a clock frequency of 60 Hz by default. This frequency can be changed only by modifying the `HZ` constant defined in `/usr/include/minix/const.h` and then recompiling and rebooting. As the default clock frequency for Linux is 250 Hz, guest interrupts are delivered irregularly; four or five interrupts are delivered in direct succession, followed by a interrupt-less pause that lasts much longer than Linux expects. Hence it would be desirable for QEMU if the MINIX clock frequency were to be higher than it currently is or, even better, if it would be flexible. As I have argued before,

it is not desirable to do this as a part of the installation of QEMU, as the current MINIX has the limitation that it crashes after 2^{31} clock ticks. Moreover, the value of the `HZ` constant has performance implications. Interrupt handling by itself is relatively slow and each clock tick requires some time to be processed by the kernel. The more clock ticks there are per second, the less time remains for user processes. As an aid for making decisions on the clock frequency, I have performed tests to measure its performance impact. How these tests were performed is described in this section.

To measure the impact of the `HZ` constant, one can run a simple benchmark that determines how much of the time is available for user processes. This is done by running a small loop, the duration of which in CPU cycles is known, and counting how many iterations of the loop are possible within a fixed number of CPU cycles. The maximum number of iterations possible if the operating system does not interfere can then be computed exactly, which allows one to determine how much performance is lost due to clock interrupts being processed.

The source code for this benchmark is found on the CD-ROM in the `/utils/bench-hzctl` directory. The main loop is found in the `bench-hzctl-asm-ack.s` and `bench-hzctl-asm-gcc.s` files. Different files are used for ACK and GCC because they use different assembly syntaxes; other than the difference in syntax, the contents are the same. The makefile selects the correct version. The inner loop is written in assembly to ensure that it is as small as possible, yielding maximum accuracy. It can be summarized in C as follows:

```
u32_t perform_test_asm(u64_t cycles)
{
	u32_t count = 0;
	u64_t cycles_end;

	cycles_end = rdtsc() + cycles;
	while (rdtsc() < cycles_end)
		count++;

	return count;
}
```

Here `rdtsc` is a function that reads the number of CPU cycles since system boot as a 64-bit unsigned integer. This functionality is available as a CPU instruction on the x86 architecture. Each loop iteration takes a fixed number of CPU cycles, depending only by the CPU model used. As the benchmark runs for 2^{32} cycles, the expected number of iterations without operating system interference can be computed. This allows one to determine the percentage of time lost to operating system overhead from the actual count that is measured.

The benchmark is run in several configurations. As reference points, it is run on an unmodified version of MINIX with `HZ` set to 60 and a version with `HZ` set to 250. To be able to measure the impact of the clock frequency, a modified version of MINIX is used in which the clock frequency can be set dynamically. To this end I have implemented a device named `/dev/clock` and implemented `ioctl` operations `CLIOCGETHZ` and `CLIOCSETHZ` for it, respectively reading and changing the clock frequency. This allows many different speeds to be measured without the need for recompiling and rebooting MINIX in between. The modified MINIX has `HZ` set to a high number, but requests clock interrupts at a lower frequency depending on the value set using `CLIOCSETHZ`. The clock interrupt handler does not increment the current time by one, as is normally the case, but rather by the ratio between `HZ` and the actual clock frequency. In this way, `HZ` is still the unit used for timekeeping but fewer clock interrupts are used. This approach is not suitable for inclusion in MINIX because it requires a high value for

`HZ`, which causes MINIX to crash sooner due to overflow of the `realtime` variable, and it only allows frequencies that evenly divide the `HZ` constant. For this test I set `HZ` to 2400, which has many divisors.

5.2 - Results

Based on the measurements I performed as described earlier, I compare the performance of MINIX with that of Linux. I have chosen to use Linux as a reference operating system because it is the original platform for which QEMU was written. Moreover, it is mature, widely used, well known and open source, which makes it a suitable reference operating system and makes it easier for any differences to be explained. The MINIX version is 3.1.2a for both the host and the guest, as this was the most recent stable version when I created the test images. As a host Linux operating system, the Debian distribution of the 2.6.18 kernel is used. Since this distribution was found to be too large to create a suitable image that could fit comfortably on a MINIX partition, I used Slackware based on the 2.6.27 Linux kernel for the guest image. This distribution is known for it flexibility and easily allows one to create a reasonably small image that includes all the essentials; in particular the GCC compiler and the tools and libraries that it needs are very large and leave little room for other luxuries. In total, the MINIX image takes up 219 MB and Slackware Linux uses 377 MB, while some other Linux installations I tried exceeded 2 GB. All tests have been conducted on a laptop with a Pentium M 1.6 GHz processor and 512 MB of RAM using these operating systems. The results are presented in three graphs, each comparing different situations. The averaged data have been included as Table 7 in appendix B.1, while the raw measurements can be found in the `measurements/benchmark/benchmark-raw-data.txt` file on the CD-ROM.

It is important to note that I have not used KQEMU when measuring QEMU's performance. This kernel module speeds up emulation by executing some code directly. As explained before, this does not fit well with the MINIX philosophy of having a small and reliable kernel. For this reason, amongst others, I have not ported it to MINIX. For the benchmarks, KQEMU is not used on Linux either as this would make the results incomparable and would obscure the ways in which MINIX could be improved.

In this section, I first discuss how well MINIX performs as a guest operating system. I highlight the bottlenecks and attempt to explain them. Next, the impact of QEMU emulation and the performance of MINIX as a host operating system are considered. The performance impact of using the deterministic mode and of running QEMU recursively are also briefly discussed. Finally, the influence of the `HZ` constant is tested.

Performance of MINIX as a guest operating system

Figure 14 compares MINIX' and Linux' performance as guest operating systems. They have been compared both when running natively (as a reference point) and when running on QEMU, which in turn runs on MINIX. This allows one to determine the strong and weak points of both operating systems, providing a basis for further analysis.

The largest differences between MINIX and Linux are found to be disk input and output, floating point arithmetic and graphics. In particular, Linux is shown to be 4596 times faster than MINIX when it comes to scattered disk reads. This difference shows that Linux is not, in fact, performing much disk IO at all. This is caused by different approaches to disk caching,

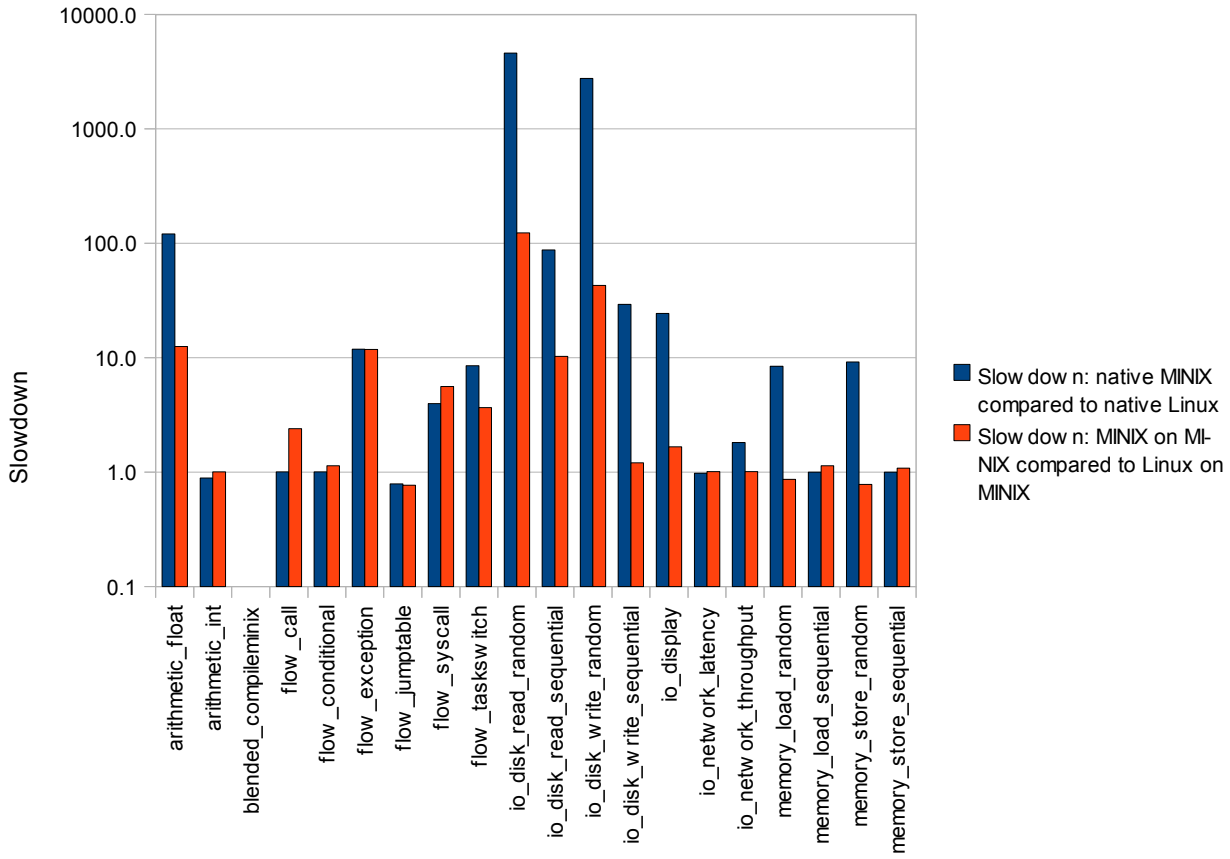

Figure 14: MINIX vs. Linux as a guest platform

which can be seen by adjusting the size of the file used in the benchmark. While MINIX has a fixed disk cache buffer in the file system server, Linux uses free memory to store disk blocks and may swap out uncommonly memory pages to accommodate even more disk blocks. The result is a vast disk cache that can accommodate all blocks that the benchmark previously read, while MINIX replaces the blocks before the time they are needed again due to lack of cache space. MINIX is fast if the file is reduced to be smaller than its disk cache, while Linux slows down substantially when the benchmark file is made larger than the amount of available memory.

The disk cache also holds dirty blocks to be written to disk at a later time, which explains that the write benchmark is also much faster. With the sequential disk read and write benchmarks, MINIX also profits from its cache which reduces Linux' lead substantially. The differences are also smaller for the emulated operating systems, which can be explained by the fact that a additional layer of disk cache is added. This effectively increases available cache for both operating systems as caching is done by both the host and the guest.

The fact that Linux is much faster at floating point arithmetic is due to the fact that MINIX does not support the floating point x86 instructions. Instead, the compiler generates calls to library functions that perform the computations using integer arithmetic. This is much slower than using hardware-supported floating point operations. The difference is smaller when emulated, but it still exists. Because QEMU is running on MINIX, floating point

arithmetic is done by a library in both cases but with Linux it runs natively using QEMU's floating point library while with MINIX the library is on the virtual machine and is therefore emulated.

Regarding graphics, Linux is considerably faster than MINIX when running natively but this difference largely disappears when the operating systems are emulated by QEMU. There are two differences that may explain this. First, Linux supports kernel graphics drivers that can access the hardware directly, while in MINIX the graphics drivers are part of X11 and are completely in user space. Second, it appears that MINIX does not have a driver to support the graphics chip on the test machine well and falls back to a default driver. Linux, on the other hand, has a specialized driver. The source code reveals that it uses hardware acceleration provided by the video hardware to speed up drawing the screen. If one considers all measurements (including those not compared in the graph here), it turns out that native Linux is much faster (at least a factor 24) than all other configurations, which do not differ nearly as much among each other. This suggests that hardware acceleration is the main reason for the difference. This is only beneficial when the operating system supports it and the machine drawing the graphics is able to directly access the hardware. The former is not the case on MINIX as there is not specialized driver, the latter is not the case with QEMU as the guest draws the screen leaving the host with only a buffer filled with pixel data.

The most important differences have now been discussed but some smaller ones remain to be explained. In particular, `flow_exception`, `flow_syscall` and `flow_taskswitch` are considerably slower on MINIX in both configurations. These benchmarks have in common that they include many task switches on the guest side. `flow_exception` switches to the kernel when an invalid operation is performed and then goes through the process management server to deliver a signal. Eventually the process manager is called again (through the kernel) to restore the situation prior to the exception. On Linux, which has a monolithic design, these tasks are performed directly by the kernel and switches between the kernel and process manager are not needed. Similar arguments go for `flow_syscall` and `flow_taskswitch`, both of which spend most of their time performing system calls. The results show that the additional context switches incur a substantial performance overhead in both configurations.

The last remaining benchmarks which perform substantially worse on MINIX are the memory benchmarks, but only when performing scattered memory accesses and only when running natively. The most important factor influencing memory access are the CPU caches, which are substantially faster than the main memory. Since there is no difference for sequential access, it appears that the main difference is that in Linux cache entries are retained for a longer time and there are therefore more cache hits; memory caching clearly is functional in MINIX when memory access is sequential. The factor ten difference found by the benchmark suggests that substantial amounts of cache are lost. This has to happen in the clock interrupt handler as this is the only way in which the operating system comes into the loop while performing the test. When looking at opcode histogram data collected by QEMU, it is notable that the `LLDT` instruction is used by MINIX but not by Linux. This instruction is executed on each interrupt and replaces the segment table for a user process (the local descriptor table). It seems plausible that this would affect caching. To verify whether the `LLDT` instruction is indeed the cause of the difference, I have reduced the number of times it is used and considered the impact. This procedure is described in appendix B.2. My conclusion is that the `LLDT` instruction is not the cause of the difference, which leaves the performance

gap unexplained. More research will be needed to determine why scattered memory operations are slow on MINIX.

Most of the remaining benchmarks have a slow-down factor close to one, meaning that there is little difference between the operating systems. It is notable that MINIX is slightly faster in a few cases. These are all benchmarks that do not use any operating system operating system services. This suggests that the overhead of the handling of timer interrupts explains the difference, as this is the only way in which the operating system interferes if it is not actively called and there is no other input. By default, MINIX programs the system clock to receive 60 ticks per second while Linux requests 250 ticks per second. As is shown in a separate sub-section, handling clock ticks does indeed take significant time away from useful processing. Although Linux is slightly more efficient when compared at the same clock frequency, the difference is so small that MINIX at 60 Hz is clearly faster than Linux at 250 Hz. This shows that there is a trade-off between speed and accurate time measurement.

Within the category of benchmarks that does not (directly) use operating system services, `flow_call` is a notable exception. Unlike the others MINIX is performs badly at it, but only when running on QEMU. Using the opcode histogram feature, this can be found to happen because the generated code contains additional instructions to add the stack segment base on stack references. This is optimized out on Linux, where segmentation is not used and each segment has base zero. Linux uses paging instead to isolate processes from each other. The `flow_call` benchmark performs recursive function calls, causing many stack references. This explains that the difference is particularly noticeable for this specific benchmark.

All in all, there are some areas where MINIX performs substantially worse than Linux as a guest operating system. These differences are more pronounced when running natively than when running on QEMU. Most performance differences could be explained from differences in design choices made for Linux and MINIX, with the Linux generally going with the approach that focusses on performance. Most of these issues are not inherently tied with the microkernel and it is probably possible to resolve them within this framework. I was unable to determine why scattered memory accesses are slower on MINIX than Linux when running natively; this warrants further research to be able to find out whether the difference can be addressed.

Performance of QEMU itself

Figure 15 shows the impact of emulation, comparing the operating systems running natively with them being emulated by QEMU. This allows one to assess the strong and weak points of QEMU itself and determine whether, given certain performance requirements, using virtual machines is feasible. For maximal comparability, native Linux is compared with Linux on Linux while native MINIX is compared with MINIX on MINIX.

On average, the benchmarks have slow-down factors close to ten. The blended benchmark, which recompiles MINIX from source, is also just above the 10× slow-down mark. This benchmark is a better approximation of real-life usage patterns than the others, which suggests that the slow-down factor experienced in practice will be close to ten as well. This base slow-down can be explained from the fact that the original code is not run directly, but rather dynamically translated into a version that interacts with QEMU's CPU state stored in memory. This means that more instructions are needed and that memory is used rather CPU registers. Profiling shows that code translation itself does not have much of an impact on

performance. Previously translated basic blocks are stored in a large cache organized as a hash table, so they can quickly be retrieved later on.

It is notable that the slow-down for the integer benchmark is considerably lower than for the floating point benchmark. This is the case even for MINIX, which emulates the FPU using integer arithmetic. A glance at the opcode histograms reveals that the opcodes used by the `arithmetic_float` benchmark on MINIX are very diverse, while a large part of the `arithmetic_int` benchmark are conditional branching instructions. The latter benchmark tests whether integers are prime in an inefficient way, namely by attempting to divide them by all integers from 2 up to their square root. This does indeed involve much conditional branching. Moreover, these branches are hard to predict. Hence it turns out that, on MINIX the floating point benchmark actually provides a better test for integer arithmetic than the integer benchmark does. Hard-to-predict branching instructions are slow to execute even natively. The CPU attempts to predict the path taken and speculatively executes the instructions on the expected path of execution, so incorrect predictions require instructions to be rolled back. This incurs a large performance penalty for modern CPUs, which have large pipelines. Hence the added overhead of emulation is relatively less, causing a lower slow-down. A similar slow-down factor can be observed for the `flow_conditional` benchmark, which also focusses on conditional branching instructions. Some of the branches in this benchmark are more predictable than those in the `arithmetic_int` benchmark, which explains the fact that its slow-down is slightly higher.

The slow-down for `flow_call` is typical for Linux, but somewhat on the high side for MINIX. As mentioned before this is due to the fact that MINIX has a nonzero stack segment base. This causes additional instructions to be generated for adding this base on any stack memory reference.

The `flow_exception`, `flow_syscall` and `flow_taskswitch` benchmarks all involve context switches. On MINIX, multiple user processes are involved in each of these benchmarks since the process manager is always involved. On Linux process management is performed by the kernel, so that the `flow_exception` and `flow_syscall` benchmarks only call the kernel and return to the same process. It is clear that the benchmarks switching between user processes experience severe slow-down, while those involving only kernel calls are close to the default slow-down factor of ten. On MINIX, a task switch involves changing the local segment table while on Linux it requires activating a different page table. Neither is needed when switching between the kernel and a user process. This involves invalidating some memory-related caches, although changing the page table has considerably more impact because the entire translation look-aside buffer (TLB) becomes invalid. This buffer caches page table entries and is important for performance of the memory management unit (MMU). To rebuild it costs time both for a physical CPU and for QEMU, but the latter is affected most as it has to do the translations entirely in software. This involves looking up pages in the page table and determining access rights on memory accesses after the flush. This explains that context switches increase the slow-down and in makes clear why the slow-down is particularly large for the `flow_taskswitch` benchmark on Linux.

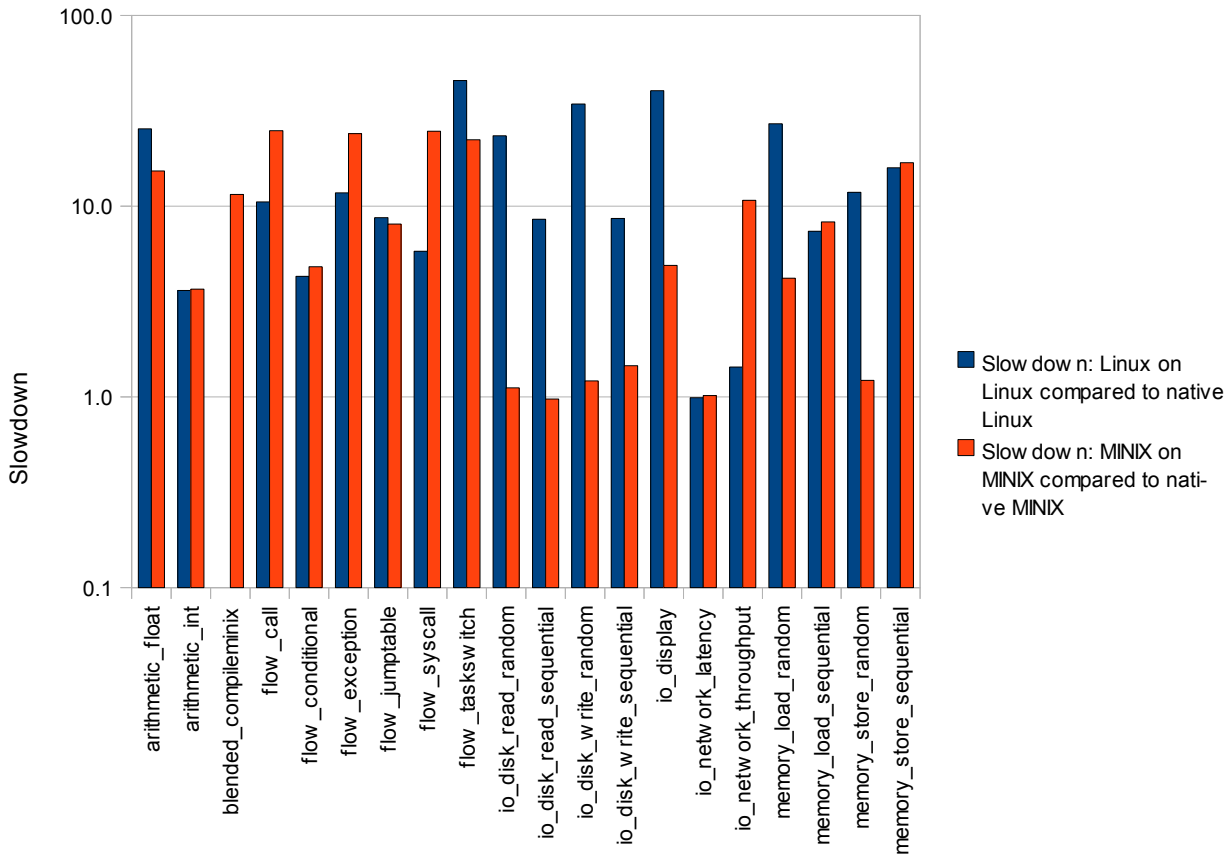

Figure 15: Slow-down caused by emulation

Regarding disk input and output, it is notable that slow-downs are typical for Linux but very small for MINIX. As noted during the previous comparison, this can be explained from the different implementations of disk caching between the operating systems; while Linux has a vast amount of memory available to store disk blocks and speed up reading and writing, the small and fixed cache used by MINIX is insufficient to accommodate the file used for the benchmark. As a result, MINIX spends nearly all its time reading disk blocks, which is so slow that it eclipses the slow-down caused by QEMU. Linux, on the other hand, barely spends any time waiting for the hard disk, which explains that slow-down rates do not differ much from the other benchmarks. The random-access benchmarks operate on small block of data and involve more switches to the kernel, while the sequential benchmarks transfer more data. This explains that the former experience much slow-down like the context switching benchmarks, while the latter are affected to a lesser extent and are closer to the memory benchmarks.

As mentioned in the previous sub-section, the large slow-down on Linux for the `io_display` benchmark can be explained from the use of hardware acceleration when Linux runs natively. This cannot be used when running on QEMU, causing a substantial slow-down for drawing graphics.

Network latency is barely affected by emulation, while throughput is substantially lower on emulated MINIX. The former is easily explained; the time it takes for the another host to respond over the network is so large that the overhead of emulation is small in comparison. A similar reasoning applies to throughput on Linux, which also experiences little slow-down. A

more typical slow-down is found on MINIX, which suggests that overhead per byte sent or received over the network is much larger than on Linux. Comparison of all measurements shows that the throughput benchmark is slow if and only if MINIX is used as a host OS. Using a packet sniffer on the MINIX side, one quickly notices that the bad performance is caused by occasional one-second pauses in the outgoing data. This always occurs in the following pattern:

1. The client (benchmark program) sends a packet not containing a full payload with the 'push' flag set;
2. The server acknowledges the packet of data;
3. Nothing happens for approximately one second, counting from step 1;
4. The client sends a packet containing little data, often just a single byte, with the 'push' flag set;
5. The server acknowledges the packet of data;
6. Normal operation continues.

The pauses are also clearly visible if one plots the amount of data sent against the time elapsed. Figure 16 shows that nearly all time is spent waiting. The first wait occurs after 45056 bytes have been sent and each subsequent pause after exactly 40960 more bytes. Most packets carry the maximum of 1260 bytes of data, so this is not a multiple of the packet size. It is, however, ten times the send buffer used in the benchmark.

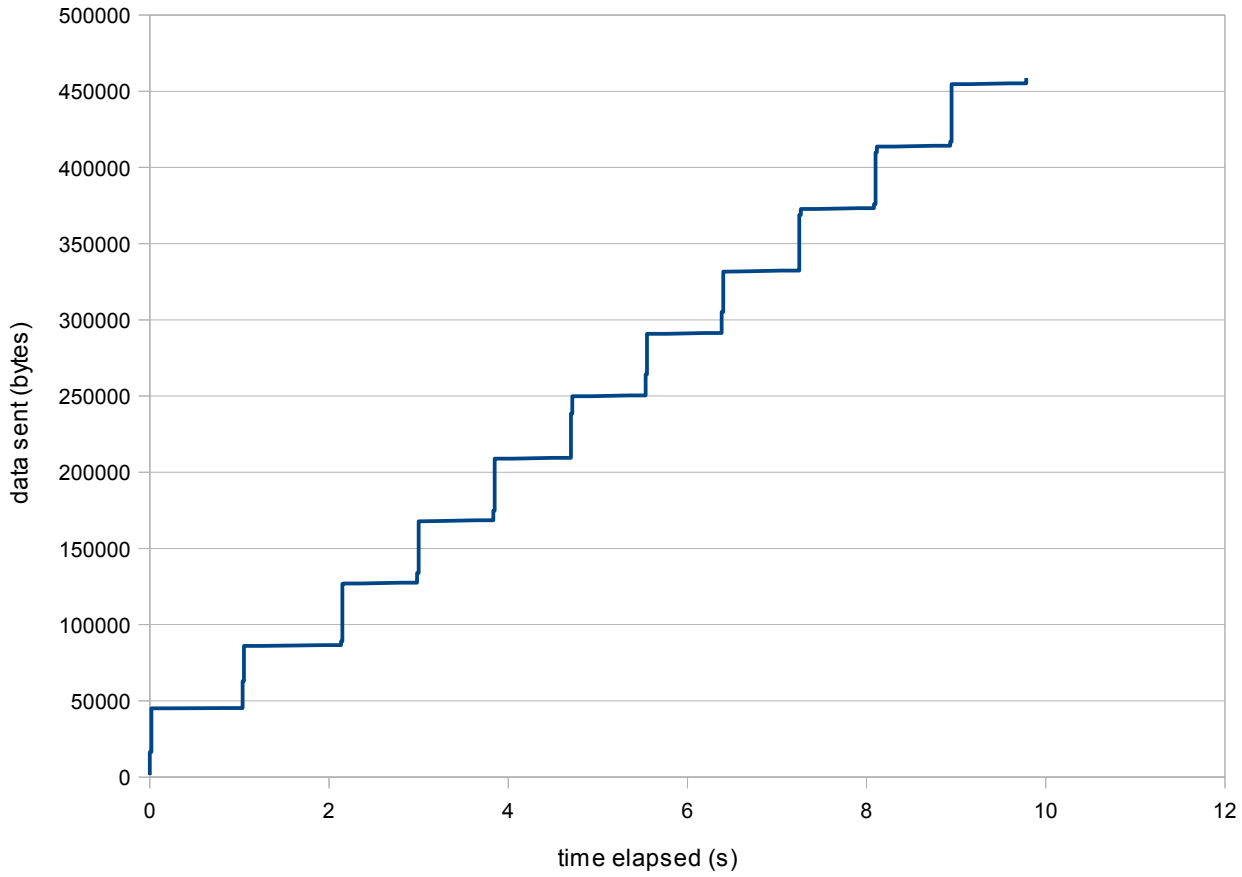

Figure 16: Amount of data sent in the `io_network_throughput` benchmark

It is clear that the TCP driver is intentionally waiting before sending the rest of the data, while the packets received do not provide any reason for this. At the time of the wait, all outgoing packets have been acknowledged by the server and the window size allows many more packets to be sent without delay. This suggests that Nagle's algorithm may be involved. This algorithm causes packet sending to be delayed in certain circumstances to avoid sending small packets. Unfortunately MINIX does not support the `TCP_NODELAY` socket option to disable Nagle's algorithm, which makes it harder to test whether this is the origin of the problem. It does, however, have the `NWIOTCPPUSH` `ioctl` code which causes queued data for a TCP socket to be sent. Adding this call after each write should bypass Nagle's algorithm as well. I have tried this and found that it does not solve the problem, so Nagle's algorithm is not to blame.

More research will be needed to find out why MINIX waits for such long periods of time while sending out data. It should be noted that the network card on the computer used to perform the tests is not supported by MINIX, so I had to modify the Intel Pro/100 driver to include support for it. The modified driver is found on the CD-ROM in the `/minix/minix-3.1.2a-fxp` directory. Although it is possible that the driver would cause problems, I do not consider this to be likely. The packet sniffer shows that incoming packets are received correctly and that MINIX does not respond to them in a timely manner, which is a sign that the TCP driver is the more likely to be the cause of the problem.

Since QEMU is configured to emulate the memory management unit (MMU) in software, lack of hardware support for page translation is an additional factor causing slow-down. This explains the fact that Linux performs much worse on the random-access memory benchmarks than MINIX; Linux actively uses paging while MINIX disables it. This is compensated in part by using a translation look-aside buffer (TLB) to store translated pages, but this approach is effective only if successive memory accesses refer to the same pages. This is the case for the sequential benchmark, where MINIX and Linux suffer from similar slow-downs, but not for the random-access benchmarks.

The fact that, for MINIX, the slow-down is relatively low for the random-access memory benchmarks can be explained by the role of caching. Sequential memory accesses profit from the level 1 and level 2 CPU caches, which speeds up access substantially when running natively. This means that the overhead added by QEMU is relatively more when compared to the slower random-access memory references that do not benefit from CPU cache.

Performance of MINIX as a host operating system

Finally, Figure 17 compares MINIX and Linux as host operating systems. This shows in which areas the port performs well and in which it is lacking. Many benchmarks do not show a meaningful slow-down or speed-up, which shows that they do not depend on the host operating system. The `blended_compileminix` benchmark has only a minor slow-down when emulated by MINIX, which suggests that the benchmarks where no difference is observed are dominant regarding real-world performance. I now turn to the benchmarks that do show a difference to be able to determine what causes them.

The first difference is the `arithmetic_float` benchmark. A Linux virtual machine performs much worse at this test when running on top of MINIX than when running on top of Linux. This is due to the lack of floating point support in MINIX, which causes QEMU to use library routines to perform the floating point computations on the Linux guest machine. When

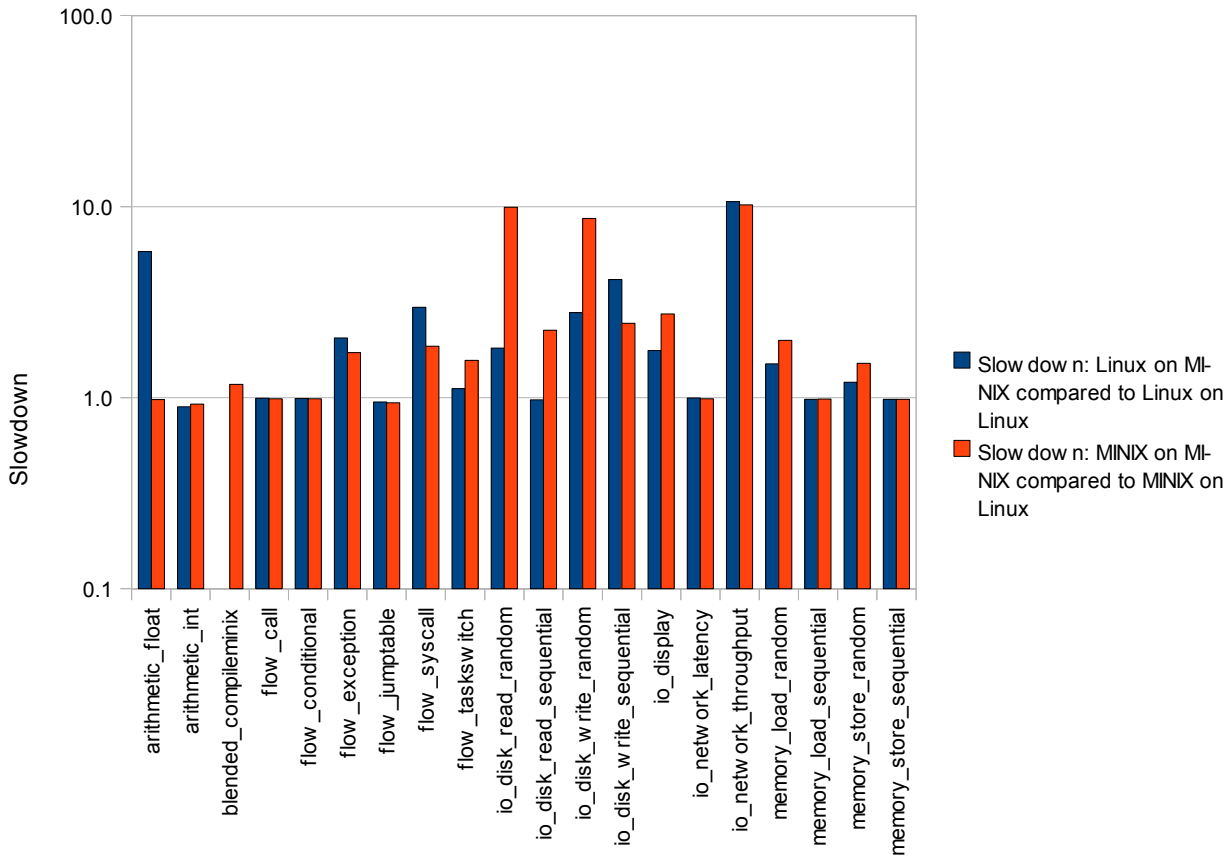

Figure 17: MINIX vs. Linux as a host platform

MINIX is running as a guest, the host platform does not make much of a difference because the guest never uses any floating point instructions. In this case the floating point library runs on the guest and the host operating system does not make much of a difference.

The `flow_exception`, `flow_syscall` and `flow_taskswitch` benchmarks have in common that they use interrupts to shift control from a user process to the kernel on the guest machine. QEMU implements these interrupts using the `setjmp/longjmp` pair to escape from the generated code back to the main loop. The fact that benchmarks involving interrupts are consistently slower on MINIX suggests that these functions perform badly on MINIX. These calls involve more context switches on MINIX, where they are handled by a separate user-mode process manager, than in Linux, where they are processed completely by the kernel.

The disk benchmarks are substantially slower when MINIX is used as a host platform. As explained before, Linux has a disk cache that is typically orders of magnitude larger than MINIX' disk cache. This can be shown by making QEMU log the amount of time spent on reading and writing. Using the `io_disk_read_random` benchmark as an example, a MINIX guest reads 2386 disk blocks while Linux does not need to read a single disk block as all have been retained in the cache from the time at which they were written. Caching also makes a difference at the host side; a Linux host reads all blocks from the cache and takes only 27.5 ms, while MINIX reads them from disk, using 11797.3 ms. This causes the disk benchmarks to perform substantially worse when MINIX is used as a host platform, in particular when the guest platform is also MINIX. This effect is less for the sequential benchmarks than for the random-access benchmarks, as the former can benefit even from a small cache.

The impact of MINIX as a host operating system on graphics, network throughput and random memory access was discussed previously. MINIX uses a generic driver for the display, making hardware acceleration impossible. The effect is not nearly as dramatic as the difference between native and emulated Linux because hardware acceleration is mainly useful for drawing, but apparently a specialized driver is also better at just getting the pixels to the video card. Additionally MINIX does not support the FPU, MMX and SSE instruction sets and for this reason assembly routines have been disabled in SDL, causing further slow-down compared to Linux. Network throughput is low due to occasional one-second pauses when MINIX is used as a host operating system. The bad performance for scattered memory accesses was blamed on cache flushes, although it is unclear why this happens.

Impact of the deterministic mode

As has been discussed previously, a 'deterministic mode' has been added to QEMU. If QEMU is configured to use this by specifying `--enable-deterministic` to the configure script, it avoids using the `setitimer` function. Instead, guest time and clock interrupts are based on the number of instructions executed by the guest. For each translated basic block, a test is added to determine whether a clock interrupt should be delivered. This slows down the program, but has the following advantages:

- There is no need to add the `setitimer` function to MINIX;
- Clock interrupts are delivered more accurately for guest operating systems that request high frequencies, as they no longer depend on the clock resolution of the host;
- It is possible to reproduce the exact same execution of a virtual machine any number of times, because no factors external to the guest influence timing.

To determine whether it is feasible to use this mode as a default, I have compared its performance to that of the regular QEMU running on MINIX. The results are shown in Figure 18. Since my aim is only to determine whether deterministic mode is suitable as a default, I do not discuss the per-benchmark differences in-depth. The overall slow-down factor, as measured by the benchmark which re-compiles MINIX, is just below 1.5. Benchmarks that involve much branching are affected most by the deterministic mode. This is due to the fact that these benchmarks have smaller basic blocks, which means that the instruction count must be updated and checked more often. The disk benchmarks also suffer from deterministic mode, but only with a Linux guest. Since it was found before that Linux does not have to wait much for the disk due to its large disk cache, this means that the code to retrieve blocks from the cache has many branches as well. All in all, the performance penalty is not extreme but is too high for making deterministic mode the default configuration. Adding the `setitimer` system call to MINIX and having less accurate timers on the guest is more acceptable than this performance loss.

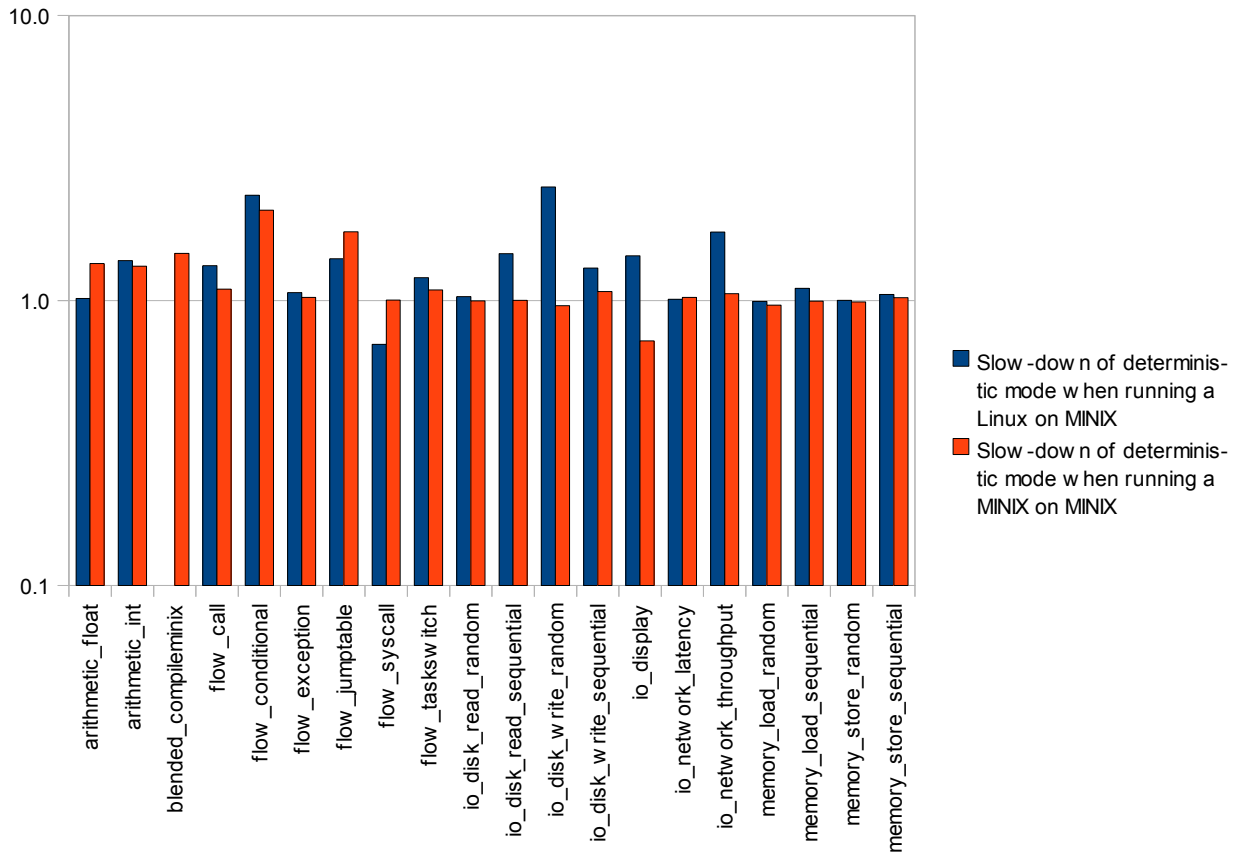

Figure 18: Slow-down of deterministic mode compared to default mode on MINIX

Recursive emulation

To test whether recursive emulation—running QEMU within QEMU—is feasible, I installed QEMU on the MINIX QEMU image and used it to run that same image. Although this may not have much practical benefit, it is important theoretically that recursive emulation is possible. If it can be done, this shows that the emulated environment is in a way equivalent to the host computer. I compiled MINIX at all levels to determine the slow-down factor of recursive emulation. Using the regular benchmark suite to compare performance was not feasible since it takes too much time when going through two layers of emulation. I used the following steps to measure performance:

```
cd /usr/src/tools
make clean
time make image
```

The `time` command executes `make image`, which causes MINIX to be recompiled, and displays the amount of time this took afterwards. It distinguishes between real time, user time and system time. User time refers to the amount of time the process itself has been running and system time to the amount of time the operating system has been working on its behalf, excluding waiting time.

The resulting timings are shown in Table 6. These figures are all based on a single measurement and are therefore less reliable than the previous results. The first layer of emulation causes a slow-down factor just over ten, consistent with the previous results using the Benchmark program. The slow-down for the second layer of emulation is considerably larger; it is over 25 times slower than the benchmark running directly in QEMU.

The x86 architecture has only eight general purpose registers, three of which are used to store temporary registers `T0`, `T1` and `A0` used by QEMU and a fourth of which is used to store a pointer to the CPU context. The stack pointer, although considered a general purpose register, is unavailable for general use because it must point to a valid location on the stack when signals are received. The three remaining registers, `EAX`, `ECX` and `EDX`, are not preserved between function calls and therefore cannot be used for long-term storage. As a result, none of the registers of an x86 guest can be stored in x86 registers on the host. They are all stored in memory, which means that code generated by QEMU uses contains more memory references than would generally be the case. This is a consequence of the limitations of the x86 architecture and the fact that guest operations have been implemented in C rather than in assembly to increase portability.

Based on the reasoning in the previous paragraph, one finds that each register reference at the second layer causes a memory reference at the first layer of emulation. It was also found before that memory references on a MINIX guest are relatively slow on MINIX because it uses segments with nonzero bases. These have to be added to determine the actual address to be read from or written to. Hence, at the native layer, the resulting code has much overhead. The `flow_call` benchmark, which has many memory references to the stack, was found to experience a large slow-down due to this added overhead. This appears to explain why the second level of emulation adds a slow-down factor that is considerably larger than the first.

Impact of the `HZ` constant

The measurements to determine impact of the clock frequency on the overhead of the operating system have been performed in various configurations:

- Unmodified MINIX (running with a 60 Hz clock);
- MINIX with the `HZ` constant set to 250 Hz but otherwise unmodified;
- MINIX patched to use a variable clock frequency, which has to be a divisor of 2400 Hz;
- Unmodified Linux (running with a 250 Hz clock);
- Windows XP (running with a 100 Hz clock);
- Windows Vista (running with a 64 Hz clock).

By comparing the results, it is possible to estimate the impact of the clock frequency, the variable clock frequency patch and the operating system as a whole. In each situation, the

	real	user	system
native	8,08	2,96	1,23
QEMU	84,85	53,86	30,55
QEMU on QEMU	2150,56	1388,80	665,63

Table 6: Performance of recursive emulation; time to recompile the MINIX image in seconds

operating system overhead has been computed. I define this as the decrease in the amount of work that can be done in a fixed period of time due to operating system code executing, in particular handling interrupts. Hence, a 0% overhead indicates that there is no operating system and all sources of interrupts have been disabled, while 100% overhead means that no useful work can be done because the operating system takes up all time. Each measurement has been performed five times, computing the average to obtain more reliable results. These averages as well as the overhead percentages computed from them have been included in appendix B.3.

Figure 19 shows the overhead at various clock frequencies for the patched version of MINIX. As was to be expected overhead increases linearly with clock frequency, so on average each additional clock tick takes a fixed amount of time away from user processes. The slope of the line, which indicates the impact of adding a single clock tick per second, is $7.29\ 10^{-4}$ %/Hz. This suggests that increasing the clock frequency to a few hundred Hz should not noticeably decrease performance. The intercept indicates how much time would be used by the operating system in the (hypothetical) case that there was no overhead from timer interrupts. As the graph shows, the fixed overhead is 0.124%.

To put the numbers measured with the patched into perspective, they can be compared with the other measurements. Unmodified MINIX has an overhead of 0.158% at 60 Hz and 0.292% at 250 Hz. These figures are only slightly below the ones measured for the patched MINIX, which shows that the additions to make the clock frequency flexible do not impact

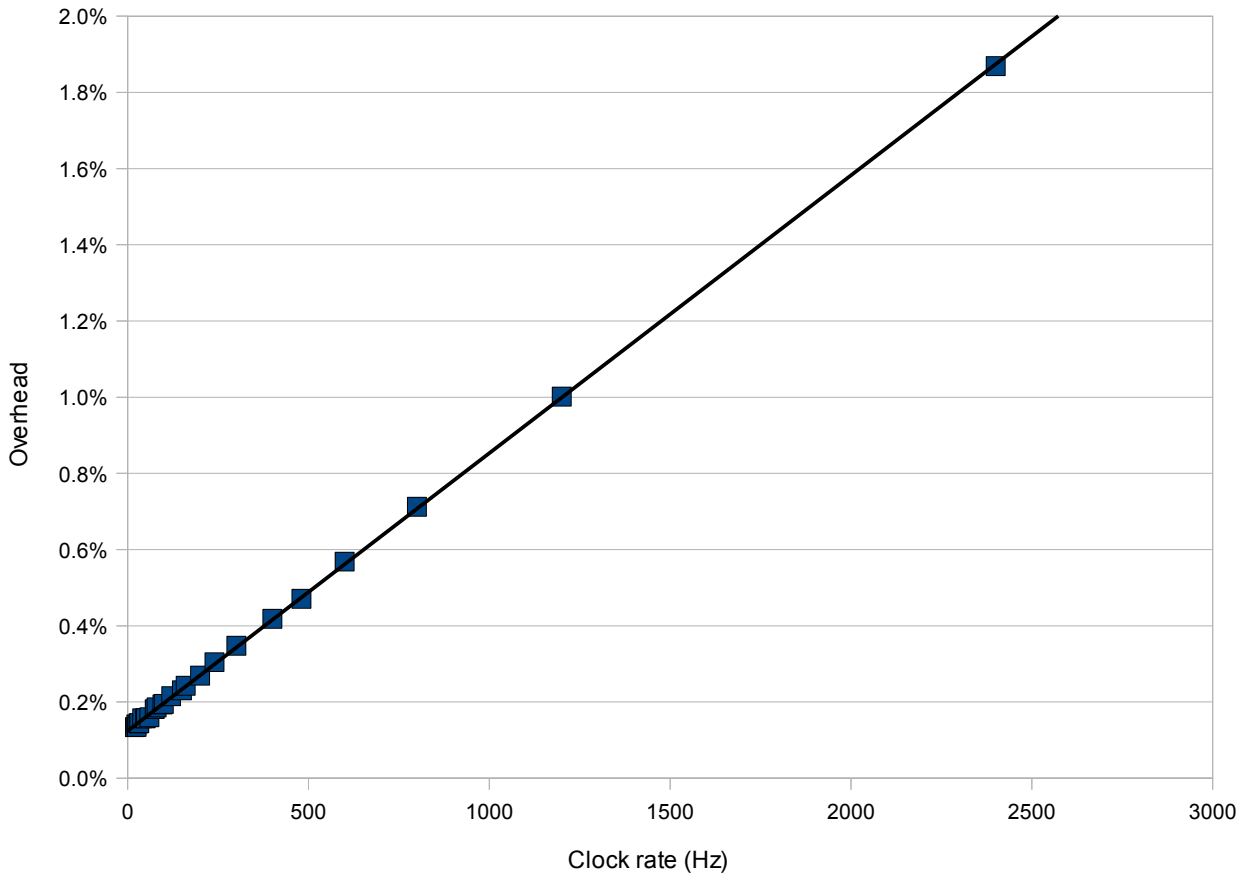

Figure 19: Relationship between overhead caused by the operating system and timer frequency, on MINIX patched to have a variable clock frequency

overhead much. Linux causes an overhead of only 0.262%, which is somewhat better than MINIX at 250 Hz. This shows that the overhead on MINIX is higher than it needs to be, but definitely not excessively high. Windows XP, on the other hand, has an overhead of 0.596% at 100 Hz. Windows Vista, run on different computer (AMD Turion 64 X2 TL-60, 2.0 GHz), has 1.402% overhead on the first core and 0.566% on the second despite a higher CPU frequency and a clock frequency of only 64 Hz.

From the results, it is clear that clock ticks do not cause excessive overhead in MINIX. At similar clock frequencies, MINIX performs slightly worse than Linux and much better than Windows. The clock frequency could be increased to 600 Hz before it reached the overhead level of Windows XP and well over 1000 Hz before it would cause as much overhead as Windows Vista. This shows that increasing the clock frequency to be able to better run QEMU might be worthwhile. A clock frequency of 250 Hz would allow it to accurately emulate the operating systems tested here, while the performance impact of such a change would most likely not be noticeable. However, before such a change could be made, the previously described overflow that causes MINIX to crash after 2^{31} clock ticks ought to be fixed.

5.3 - Discussion

Both as a guest and as a host operating system, there are some areas where MINIX does not perform as well as Linux. This chapter has revealed the kinds of tasks for which this is the case and has discussed the main causes for some of these differences. In most cases, poor performance can be explained by design decisions that have little to do with MINIX' microkernel design. Even though a number of areas have been mentioned in which MINIX' performance can be improved, it should be noted that in practical applications the difference is much smaller because these features are not used as intensively as in the benchmarks.

I have found several ways in which MINIX can be improved to more closely match the performance of its competitors without compromising the microkernel design. Graphics and floating point computations can be sped up substantially by using the hardware acceleration provided by respectively the graphics card and the FPU. The file system server can be modified to use free memory for caching disk blocks, as the kernel already provides system calls that would allow it to access this memory. Network throughput is low due to occasional one-second pauses. Although more research will be needed to find out why MINIX behaves this way, it is clear that context switches cannot cause such large delays.

The only meaningful performance difference that may be due to the microkernel design is the fact that benchmarks which use signal handling or the `setjmp`/`longjmp` functions perform badly. These features involve many task switches due to the fact that they are not handled within the kernel. It is likely that their bad performance is related to this.

One difference of which the origin remains unclear is that fact that scattered memory reads are slower on MINIX than on Linux. This is most likely due to the memory cache being flushed more often. More research is needed to determine whether and why this happens. In particular, it would be important to know whether this is inherent in the microkernel design or is due to some system instructions that can be avoided. Although x86 provides performance counters to measure how effectively the memory cache is used, MINIX does not yet provide access to them. If this were added, it would make such performance issues easier to debug.

Regarding QEMU timer accuracy, it has been found that the deterministic mode I implemented in QEMU is too slow as a general solution to solve the issue. MINIX' handling of clock interrupts is reasonably efficient however, so it would be feasible performance-wise to increase the frequency of the clock. If the `HZ` constant were to be increased to 250 Hz this would substantially improve the quality of QEMU's emulation. I have decided, however, not to do this in the MINIX patch for now as it would cause MINIX' timekeeping variable to overflow sooner, reducing the maximum possible uptime. This can be fixed first or alternatively a tickless kernel can be implemented, which would provide even larger advantages.

6 - Conclusions

My main finding is that it is possible to run virtualization software on MINIX. However, having it run in an stable and efficient way does require that some changes be made to the operating system. These changes are relatively small and do not deviate from MINIX' design goals, so they can be merged into the source tree. Most changes involve adding system calls that are generally available on other POSIX systems. These additions will also make other ports easier to perform.

It has also been found that QEMU's performance is only slightly worse on MINIX than on Linux for practical use, although there are some specific areas where MINIX performs poorly. Each of these areas has been investigated, leading to recommendations for changes that could take away performance bottlenecks. Examples are making use of hardware acceleration wherever available and using free memory to increase the size of the disk cache. In some cases my tests could not reveal the origin of the difference, but pointers have been provided for future research to be able to find them. Few of the performance issues can be ascribed to MINIX' microkernel design, which shows that there is potential for improvement within the current framework.

It should be noted that I have chosen to have QEMU run entirely in user space on MINIX and have compared it to the same situation on Linux. If QEMU has access to kernel mode, as is possible on Linux by installing the KQEMU kernel module, it can achieve a substantial performance gain by executing some guest code directly. Loading additional code into the kernel is at odds with the MINIX philosophy, as it increases the probability of fatal kernel-mode bugs. Hence MINIX is likely to remain an underperformer in the area of virtualization when compared to operating systems that virtualize in kernel mode, although an alternative may be to use recent instruction set additions for hardware-level virtualization. Using these instructions, it is possible to protect the kernel even better than the kernel/user-mode distinction allows.

For many purposes, QEMU is usable in practice on MINIX. The main issues that might reduce usability are a lack of available of memory and poor performance. Regarding the first issue I have added features (see section 3.4) and provided recommendations (see section 4.2) that should allow problems due to a lack of available memory to be addressed. Moreover, this issue will be eliminated when virtual memory is adopted in MINIX' stable branch. Whether performance is sufficient depends on one's aims. QEMU is certainly fast enough to run MINIX for debugging purposes and other simple tasks. Graphically browsing the Internet, currently not natively possible in MINIX, is fast enough for practical use when running Internet Explorer on Windows 98. A more up-to-date configuration, Mozilla Firefox running on a recent version of Linux, is too slow to be useful in practice on my test system. It might, however, be feasible on more recent hardware.

This project has made several contributions besides answering the research questions. It has made virtualization available on MINIX and this thesis provides instructions for those porting other software to this operating system. Moreover, it provides a list of possible improvements to MINIX. Some of these recommendations would make porting software to MINIX easier when implemented (see section 3.8), while others could improve its performance while leaving its basic design intact (see section 5.3).

Bibliography

[1] A. Baratz. Virtual machine shootout: VMware vs. Virtual PC, Ars Technica, August 8, 2004, http://arstechnica.com/reviews/apps/vm.ars (visited July 23rd, 2009).

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt and A. Warfield. Xen and the Art of Virtualization, Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM Press, New York, NY, USA, 2003, pp. 164-177.

[3] V. R. Basili and B. T. Perricone. Software errors and Complexity: An Empirical Investigation, Communications of the ACM, ACM Press, New York, NY, USA, Volume 27, Jan. 1984, pp. 43-52.

[4] F. Bellard. QEMU, A Fast and Portable Dynamic Translator, Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track, April 2005, pp. 41-46.

[5] F. Bellard. QEMU internals, http://www.nongnu.org/qemu/qemu-tech.html (visited July 23rd, 2009).

[6] K. J. Gough. Stacking them up: a comparison of virtual machines, Proceedings of the 6th Australasian conference on Computer systems architecture, IEEE Computer Society, Washington, DC, USA, 2001, pp. 55-61.

[7] J. Gray. Go green, save green with Linux, Linux Journal, Specialized Systems Consultants Inc., Seattle, WA, USA, Volume 2008, Issue 168, April 2008, Article No. 1.

[8] Intel Corporation. Intel Virtualization Technology Specification for the IA-32 Intel Architecture, C97063-002, April 2005, http://www.intel.com/cd/ids/developer/asmo-na/eng/214273.htm (visited July 23rd, 2009).

[9] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, order number 253665-027US, April 2008, http://download.intel.com/design/processor/manuals/253665.pdf (visited July 23rd, 2009).

[10] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z, order number 253667-027US, April 2008, http://download.intel.com/design/processor/manuals/253667.pdf (visited July 23rd, 2009).

[11] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, order number 253668-027US, July 2008, http://download.intel.com/design/processor/manuals/253668.pdf (visited July 23rd, 2009).

[12] P. H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root, Proceedings of the 2nd International System Administration and Networking Conference, 2000, http://phk.freebsd.dk/pubs/sane2000-jail.pdf (visited July 23rd, 2009).

[13] I. Kelly. Porting MINIX to Xen, May 8, 2006, http://choices.cs.uiuc.edu/cache/Report.pdf (visited July 23rd, 2009).

[14] E. van der Kouwe and J. F. de Smit. Code execution absurdity in Minix. Usenet thread on comp.os.minix, April 16, 2008. Available from Google Groups at http://groups.google.com/group/comp.os.minix/browse_thread/thread/d6aa979of256711e/a15e6facce08c4f1 (visited July 23rd, 2009).

[15] M. Krasnyansky. Universal TUN/TAP device driver. 2000. http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt (visited July 23rd, 2009).

[16] J. LeVasseur, V. Uhlig, M. Chapma, P. Chubb, B. Leslie and G. Heiser. Pre-Virtualization: Slashing the Cost of Virtualization, Technical Report PA005520, National ICT Australia, October 2005, http://ertos.nicta.com.au/publications/papers/LeVasseur_UCCLH_05-tr.pdf (visited July 23rd, 2009).

[17] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, ACM Press, New York, NY, USA, Volume 17, Issue 7, July 1974, pp. 412-421.

[18] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor, Proceedings of the 9th USENIX Security Symposium, August 2000, pp. 129-144.

[19] J. E. Smith and R. Nair. The architecture of virtual machines, Computer, IEEE Computer Society Press, Los Alamitos, CA, USA, Volume 38, Issue 5, May 2005, pp. 32-38.

[20] A. S. Tanenbaum. The MINIX 3 operating system. http://www.minix3.org/ (visited July 23rd, 2009).

[21] A. S. Tanenbaum and A. S. Woodhull. Operating Systems: Design and Implementation, third edition. Pearson Prentice Hall, Upper Saddle River, NJ.

[22] The Open Group. The Authorized Guide to Version 3 of the Single UNIX Specification. The Open Group, 2004.

[23] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung and L. Smith. Intel Virtualization Technology, Computer, IEEE Computer Society Press, Los Alamitos, CA, USA, Volume 38, Issue 5, May 2005, pp. 48-56.

[24] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation, ACM SIGPLAN Notices archive, ACM Press New York, NY, USA, Volume 35, Issue 7, July 2000, pp. 41-51.

[25] VMWare Inc. ESX Server 2 Security White Paper, 2004, http://www.vmware.com/pdf/esx2_security.pdf (visited July 23rd, 2009).

[26] VMWare Inc. Virtualization Architectural Considerations and other evaluation criteria, 2005, http://www.vmware.com/pdf/virtualization_considerations.pdf (visited July 23rd, 2009).

Appendix A - Contents of the CD-ROM

A CD-ROM containing source code and benchmark results is bundled with this thesis. The contents of its directories are as follows:

`/doc`	The thesis itself in PDF format;
`/downloads`	Files downloaded by the install script;
`/images`	The disk images that were used to benchmark QEMU;
`/measurements`	Raw data resulting from the performance measurements;
`/minix`	Patches for MINIX 3.1.2a to be able to run QEMU;
`/qemu`	QEMU 0.8.2 for MINIX: source, binaries and install script;
`/SDL`	SDL 1.2.13 for MINIX: source and binaries;
`/utils`	Utilities used for benchmarking and testing.

Wherever relevant, each directory contains a file named `contents.txt` that indicates what can be found in the files and/or subdirectories.

The directories for MINIX, QEMU and SDL contain complete copies of the source code, patch files and archive files containing the source code. The complete copies of the source code are most useful to browse through the source code, while the patches can be used to get an overview of the changes I made. To be able to apply the patches, one needs to use the correct version of the source code to be patched. Therefore, the original source code of these programs is also provided on the CD-ROM. The archive is most useful if one wants to use the patched source code on MINIX, because it is inconvenient to copy an entire directory using the `isoread` tool.

The `utils` directory contains various utilities that have been created as a part of this project and have been used to perform the experiments this thesis reports on. Each of them has a `Makefile` that compiles all executables that are needed. The programs in the `benchmark` subdirectory should be compiled using GCC 4, while the others can be compiled using the default ACK compiler. Since for the `benchmark` program the C source files are compliant and both ACK and GCC assembly files are provided, one could easily change the `Makefile` to allow compilation using ACK. However, the results would no longer be comparable with those on other platforms due to the fact that GCC performs better at optimization.

The `utils/bench-hzctl` directory contains the `bench-hzctl` program which has been used to determine how much overhead the operating system causes at different clock frequencies. It can be compiled on MINIX 3.1.2a, Linux and Windows (under Cygwin) to allow for comparison between them. If the `hzctl` MINIX patch (found in `/minix/minix-3.1.2a-hzctl` on the CD-ROM) is installed, performance is measured at multiple clock frequencies. These clock frequencies are all divisors of the `HZ` constant, which the patch sets to 2400 by default. In the same directory, the `test-hzctl` program is also provided. This program was used to test the `hzctl` patch and would only be useful if one wants to make further changes to that patch. It only compiles on patched MINIX and has therefore been left out of the `Makefile`.

The programs which have been used to coordinate benchmark runs in various configurations is found in `utils/benchdriver`. The two executables are `benchdriver` and

`benchdriverserver`, the former of which is run on the host machine and the latter on the guest machine. This server is included on the guest images in the `images` directory of the CD-ROM, in which it is started at boot time. The `benchdriver` program makes a number of assumptions on the configuration in which it is running:

- The QEMU source code should be available in `/usr/src/qemu-0.8.2`;
- The benchmark program should be available over FTP on the server identified by the `$SERVER_IP` variable in `benchdriver.sh`;
- Disk images should be stored in `/usr/_`.

These locations can be found and changed in the following files:

- `benchdriver.sh`: FTP server address;
- `benchdriver.c`: commands and file names;
- `config.h`: directories.

The benchmark program itself is found in the `utils/benchmark` directory. This program also comes with a server program, `benchmarkserver`, which is used by the network benchmarks. When used with `benchdriver`, it should be run at port 1234 on the same computer that provides the benchmark source code over FTP. If `benchmark` is run without having access to `benchmarkserver`, the network-related benchmarks will fail.

Finally, `utils/msniff` provides a primitive packet sniffer for MINIX. It intercepts all incoming and outgoing packets using the `/dev/eth` device and prints information about them to the standard output. The output format and the level of detail can be specified on the command line.

Appendix B - Performance measurements

B.1 - Benchmarking guest operating systems running on QEMU

Table 7 provides raw data on the benchmark results. These data are split by benchmark and by configuration. Each number is the average of all runs measured for that particular combination. As was described in section 5.1 on test methodology, there are at least five measurements for each but more for benchmarks that take only a short time. In practice, the average number of measurements per cell is 52 and only the benchmark which recompiles MINIX has the minimal number of measurements on some configurations. This benchmark takes longer than the others because it cannot be adjusted; the others have been configured to assure that their runtime is neither so small that it causes inaccurate measurements nor so large that only few measurements are possible in the available time. Many of the cells are based on the maximum of 80 measurements.

It is important to note that the slow-down factors cannot be computed directly from these data; for comparison purposes only measurements for which the sequence number matches are used. Hence when comparing a cell with 40 measurements to one with only 30, the last 10 measurements are ignored. The raw data on the CD-ROM do include all cases and can be used to reproduce the comparisons.

configuration	native		default QEMU				deterministic QEMU				histogram QEMU			
host OS			linux		minix		linux		minix		linux		minix	
guest OS	linux	minix	linux	minix	linux	minix	linux	minix	linux	minix	linux	minix	linux	minix
benchmark														
arithmetic_float	0,003	0,395	0,083	6,189	0,484	6,049	0,089	8,054	0,492	8,158	0,129	25,127	0,527	24,709
arithmetic_int	0,990	0,877	3,570	3,472	3,197	3,215	4,635	4,656	4,415	4,240	6,868	6,845	6,526	6,608
blended_compileminix		7,845		75,661		88,978		116,435		129,956		224,687		236,251
calibrate_busy	4,996	4,995	4,996	4,989	21,225	5,007	4,180	5,331	8,412	7,601	4,997	4,997	20,937	5,030
calibrate_idle	4,999	4,995	4,999	4,992	20,857	4,998	6,104	5,440	0,149	1,701	5,000	4,998	20,811	4,996
flow_call	0,795	0,797	8,374	20,123	8,327	19,853	10,757	21,791	11,039	21,758	23,200	38,162	22,343	36,934
flow_conditional	1,083	1,087	4,646	5,274	4,601	5,215	10,267	10,547	10,762	10,814	24,481	25,792	23,671	25,577
flow_exception	0,017	0,206	0,204	2,865	0,418	4,932	0,305	3,702	0,445	5,051	0,592	7,323	0,791	9,361
flow_jumptable	0,904	0,712	7,883	6,095	7,488	5,734	10,203	9,723	10,506	9,977	18,271	17,677	17,392	17,062
flow_syscall	0,152	0,605	0,883	7,900	2,625	14,736	0,906	10,409	1,842	14,780	2,217	20,473	4,009	27,101
flow_taskswitch	0,069	0,588	3,159	8,330	3,518	12,948	3,674	11,190	4,227	14,087	5,245	21,259	5,674	25,558
io_disk_read_random	0,003	13,017	0,064	1,462	0,117	14,503	0,083	1,806	0,120	14,450	0,155	3,457	0,204	16,380
io_disk_read_sequential	0,124	10,856	1,059	4,685	1,032	10,568	1,451	5,861	1,506	10,585	2,725	11,402	2,599	16,588
io_disk_write_random	0,004	11,578	0,143	1,621	0,398	13,983	0,188	1,937	0,996	13,422	0,341	3,582	0,633	15,710
io_disk_write_sequential	0,169	4,936	1,455	2,866	6,030	7,194	1,927	3,905	7,684	7,755	3,877	7,864	9,342	11,298
io_display	0,125	3,051	5,043	5,433	8,954	14,908	7,467	6,516	12,874	10,735	15,027	12,496	27,476	28,442
io_network_latency	1,024	1,004	1,012	1,030	1,008	1,019	1,013	1,012	1,017	1,044	1,017	1,021	1,018	1,027
io_network_throughput	0,465	0,843	0,665	0,927	9,761	9,843	2,586	1,067	16,960	10,249	0,531	2,009	9,679	6,401
memory_load_random	0,351	2,958	9,508	6,108	14,326	12,405	9,294	7,345	14,223	11,954	10,683	7,484	15,765	13,427
memory_load_sequential	0,787	0,785	5,820	6,597	5,714	6,491	6,157	7,138	6,304	6,470	33,215	39,311	32,280	38,617
memory_store_random	0,554	5,075	6,556	4,069	7,901	6,180	6,530	4,426	7,907	6,123	7,438	5,027	8,670	6,794
memory_store_sequential	0,486	0,484	7,717	8,368	7,570	8,197	7,833	8,409	7,937	8,396	20,544	24,476	19,840	23,867

Table 7: Average run-times (in seconds) for each benchmark on each configuration

B.2 - Impact of the LLDT instruction

Changes made to MINIX

As described in section 5.2, I have found that scattered memory reads and writes are substantially slower on MINIX than on Linux. The most likely cause of this was found to be the use of the `LLDT` instruction in MINIX, which I expected to interfere with CPU caching of main memory. To test whether this is indeed the case, I have modified the following code in the `kernel/mpx386.s` MINIX source file:

```
_restart:

! Restart the current process or the next process if it is set.

        cmp   (_next_ptr), 0            ! see if another process is scheduled
        jz    0f
        mov   eax, (_next_ptr)
        mov   (_proc_ptr), eax    ! schedule new process
        mov   (_next_ptr), 0
0:      mov   esp, (_proc_ptr)    ! will assume P_STACKBASE == 0
        lldt  P_LDT_SEL(esp)            ! enable process' segment descriptors
        lea   eax, P_STACKTOP(esp)      ! arrange for next interrupt
        mov   (_tss+TSS3_S_SP0), eax    ! to save state in process table
```

This assembly code prepares a user process that is to be run after the kernel has handled an interrupt. The new process has been selected in advance and `_next_ptr` points to its CPU context. In many cases, however, the current process has not yet used up its quantum and this variable is set to `NULL` to indicate that the current process need not be changed. Even in this case, however, does the `LLDT` instruction reload the table containing local segment descriptors for the process. This is unnecessary; the kernel itself does not use local segment descriptors and therefore the `LDT` register is changed only when switching back to a user process, so if `_next_ptr == NULL` then the old value is still valid. Therefore the line can safely be moved, resulting in the following code (the line which was changed and moved is shown in boldface):

```
_restart:

! Restart the current process or the next process if it is set.

        cmp   (_next_ptr), 0            ! see if another process is scheduled
        jz    0f
        mov   eax, (_next_ptr)
        mov   (_proc_ptr), eax    ! schedule new process
        mov   (_next_ptr), 0
        lldt  P_LDT_SEL(eax)            ! enable process' segment descriptors
0:      mov   esp, (_proc_ptr)    ! will assume P_STACKBASE == 0
        lea   eax, P_STACKTOP(esp)      ! arrange for next interrupt
        mov   (_tss+TSS3_S_SP0), eax    ! to save state in process table
```

After this change, the `LLDT` instruction is executed only if a new process has been selected. If `LLDT` interferes with the CPU memory cache, one would expect this to result in improved performance on the scattered memory read/write benchmarks.

For the change to be as effective as possible, `_next_ptr` should be `NULL` often. In the default implementation of the scheduler, however, this variable is set to a non-`NULL` value even if the same process has been selected as before. This may happen if there is only one runnable

process or if one process has a higher priority than the others. In these cases, unnecessary use of the LLDT instruction remains every time the process has consumed its quantum. This can be addressed by changing the pick_proc function in kernel/proc.c. This is the function which applies the scheduling policy. It originally contained the following loop:

```
for (q=0; q < NR_SCHED_QUEUES; q++) {
    if ( (rp = rdy_head[q]) != NIL_PROC) {
       next_ptr = rp;                        /* run process 'rp' next */
       if (priv(rp)->s_flags & BILLABLE)
           bill_ptr = rp;                    /* bill for system time */
       return;
    }
}
```

To reduce the number of times that next_ptr is set, it can be changed as follows (boldface marks the changed line):

```
for (q=0; q < NR_SCHED_QUEUES; q++) {
    if ( (rp = rdy_head[q]) != NIL_PROC) {
       next_ptr = (rp == proc_ptr) ? NULL : rp; /* run process 'rp' next */
       if (priv(rp)->s_flags & BILLABLE)
           bill_ptr = rp;                    /* bill for system time */
       return;
    }
}
```

Performance impact

To determine whether the changes have an impact on performance, I have compared the results of the memory benchmarks between regular MINIX 3.1.2a and a version of MINIX 3.1.2a in which the changes were applied. The LLDT instruction is executed approximately 60 times per second in the regular MINIX (on each clock tick). Since the quantum size for a user process is eight clock ticks, the patched version executes the instruction at most 7.5 times per second; it may not be executed at all if only a single process is runnable. Hence, if the LLDT instruction is the cause of the difference in memory performance, the memory benchmarks should run significantly faster with the patch applied. Tests do not provide any indication that this is the case, so the hypothesis that LLDT causes the performance difference is not supported by my measurements.

More research is needed to determine what else can explain the difference. One possible approach would be to use the x86 performance counter registers. These allow one to obtain statistics about low-level performance indicators, such as the number of cache hits and misses, which could provide clues as to why Linux performs better than MINIX in this area. This would require kernel support, as the WRMSR instruction needed to control the counters is privileged. If the kernel allows an interface for applications to program the performance counters, they can read them using the unprivileged RDPMC instruction.

B.3 - Benchmarking the impact of the clock frequency

In Table 8, benchmark results are provided for the test measuring the impact of the clock frequency. Most measurements have been done with the dynamic clock frequency patch applied. These frequencies are divisors of 2400, which is the value the patch uses for the HZ constant. Two reference measurements have been included on unmodified MINIX, as well as measurements on Windows XP and Windows Vista. It is important to note that the Windows

Vista measurements have been performed on an AMD Turion 64 X2 TL-60 at 2 GHz, while all others were done using a Pentium M 725 at 1.6 GHz, which explains the difference in CPU cycles per iteration; apparently the implementation of the `RDTSC` instruction is considerably faster on the AMD CPU.

Operating system	Clock frequency	CPU cycles/iteration	Iterations/2^{32} CPU cycles		Overhead
			Mean for 5 measurements	Max. possible	
Minix	60	43	99725571	99882960	0,158%
Minix	250	43	99591656	99882960	0,292%
Minix (patched)	20	43	99749005	99882960	0,134%
Minix (patched)	24	43	99741906	99882960	0,141%
Minix (patched)	25	43	99748562	99882960	0,135%
Minix (patched)	30	43	99739272	99882960	0,144%
Minix (patched)	32	43	99738630	99882960	0,144%
Minix (patched)	40	43	99725995	99882960	0,157%
Minix (patched)	48	43	99725593	99882960	0,158%
Minix (patched)	50	43	99726206	99882960	0,157%
Minix (patched)	60	43	99723378	99882960	0,160%
Minix (patched)	75	43	99701798	99882960	0,181%
Minix (patched)	80	43	99697512	99882960	0,186%
Minix (patched)	96	43	99689373	99882960	0,194%
Minix (patched)	100	43	99687968	99882960	0,195%
Minix (patched)	120	43	99667821	99882960	0,215%
Minix (patched)	150	43	99652383	99882960	0,231%
Minix (patched)	160	43	99640502	99882960	0,243%
Minix (patched)	200	43	99614066	99882960	0,269%
Minix (patched)	240	43	99578759	99882960	0,305%
Minix (patched)	300	43	99535874	99882960	0,347%
Minix (patched)	400	43	99465186	99882960	0,418%
Minix (patched)	480	43	99412915	99882960	0,471%
Minix (patched)	600	43	99315152	99882960	0,568%
Minix (patched)	800	43	99171257	99882960	0,713%
Minix (patched)	1200	43	98882554	99882960	1,002%
Minix (patched)	2400	43	98016450	99882960	1,869%
Linux	250	43	99620976	99882960	0,262%
Windows XP	100	43	99287422	99882960	0,596%
Windows Vista, 1st core	64	8	529345339	536870912	1,402%
Windows Vista, 2nd core	64	8	533834494	536870912	0,566%

Table 8: Averaged performance measurements of various operating systems at various clock frequencies