**Diplomarbeit**

# Adding SMP Support to a User-Level VMM

Markus Partheymüller

30. November 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

| | |
|---|---|
| Betreuender Hochschullehrer: | Prof. Dr. rer. nat. Hermann Härtig |
| Betreuender Mitarbeiter: | Dipl.-Inf. Julian Stecklina |
| Externer Betreuer: | Dipl.-Inf. Udo Steinberg, Intel Labs |

Technische Universität Dresden
Fakultät Informatik

Dresden, 23.05.2013

## **Aufgabenstellung für die Diplomarbeit**

|  |  |
|---|---|
| *Name des Studenten:* | Markus Partheymüller |
| *Studiengang:* | Informatik Diplom |
| *Immatrikulationsnummer:* | 3394043 |
| *verantwortlicher Hochschullehrer:* | Prof. Dr. Hermann Härtig |
| *Betreuer:* | Dipl.-Inf. J. Stecklina, Dipl.-Inf. Udo Steinberg |
| *Institut:* | Systemarchitektur |
| *Beginn:* | 01.06.2013 |
| *einzureichen:* | 30.11.2013 |

Thema: *Adding SMP Support to a User-Level VMM*

Vancouver is a low-complexity virtual machine monitor running on top of the NOVA microhypervisor. While otherwise fully featured, there is only proof-of-concept SMP support in Vancouver. The aim of this task is to extend the rudimentary SMP support to allow multiple virtual CPUs to execute concurrently. The major points in this task are to:

- reduce or eliminate synchronization inside the VMM and serialization of VM Exit handling,
- move device emulation from executing on the virtual CPU's context to one or more dedicated threads.

Additional goals may be to support posted writes to virtual MMIO/PIO regions (the virtual CPU resumes before the write is completely processed) and flexible migration of virtual CPUs to support load consolidation and power management.

[illegible]

Prof. Dr. H. Härtig
betreuender Hochschullehrer

## Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 30. November 2013

Markus Partheymüller

## Abstract

Virtualization is a well-known technique to facilitate a variety of use cases in both desktop and server environments. Increasingly, security becomes an important aspect of virtualization because it allows for consolidating multiple workloads on a single physical machine in a protected manner. Recent processor development shifted from increasing single-core performance to integrating multiple cores onto one chip, resulting from the physical limits imposed on the design of microprocessors. It was only a question of time until virtualization followed this trend and supported running several virtual machines truly in parallel, at first multiple single-core ones, then even virtual multi-core. In the full virtualization solution of the TU Dresden, the NOVA OS Virtualization Architecture (NOVA), SMP is only supported in the first variant, while multiple CPUs of a guest system have to share the same physical core. The goal of this thesis is to explore how this limitation can be eliminated with maximum efficiency, passing on as much available compute power to the guest as possible. By identifying and enhancing the critical synchronization mechanisms inside the VMM, the presented solution accounts for maximum scalability in the current environment while keeping the modifications modular and maintainable. A detailed evaluation of the intermediate implementations justifies the final solution and an outlook shows opportunities for improving and extending the virtual multiprocessor support in NOVA.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

Be it the simple fact that software developed for a specific operating system should be run on top of a different system, the large-scale consolidation efforts in data centers across the globe, or any other spot within this spectrum that best describes the particular use case: Virtualization is a problem solver. It provides convenience by supporting multiple operating systems on a single machine without the need to reboot. It helps save power consumption by allowing for server consolidation.

Power issues are not only relevant in environmental or economic regards, but can also impose physical limits on how higher performance in computer systems can be achieved. After a long time of simply increasing the performance of a single processor, semiconductor technology was facing obstacles like the *power wall* [12], which essentially limited the amount of improvement possible with the system design at that time. In order to further gain performance, multiple cores were integrated onto one processor die to herald the era of *multi-core* systems.

In times where nearly every available computer provides several processor cores, it is imperative for virtualization techniques to also leverage this vector of performance improvements and, most importantly, not to impede this progress. Many workloads in server environments hold great opportunities for parallel execution, e.g. web servers, databases or cloud services. However, providing a virtual multi-core system is not a simple matter. Overheads and anomalies can have an extensive impact on the degree of scalability that can be accomplished when virtualizing parallel workloads. As for instance recent cloud computing research discovered, the mere deactivation and reactivation of temporarily idle cores (e.g., due to synchronization) can cause severe performance degradation compared to running it natively [35].

However, not only performance, but also security becomes an important topic when consolidating multiple machines onto one physical host. Unrelated services that were previously physically separated are now potentially vulnerable to a new kind of attacks. Taking over the virtualization layer means full control over all software components running on top of it. To prevent this, a secure and robust isolation has to be in place, guaranteeing that a virtual machine can at most compromise itself. However, the complexity of most modern virtualization solutions renders it almost impossible to maintain this property. In order to exhibit as little attack surface as possible or even for security properties to be formally verifiable, the *Trusted Computing Base* (TCB)[1] has to be kept to a minimum.

[1] Amongst a variety of expressions, the term TCB was defined by Lampson et al. as “a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security [19]”.

One approach to a minimal TCB in research is the use of *microkernels*. By reducing the amount of code allowed to execute security-critical instructions, it is possible to greatly reduce the size of the TCB with respect to the operating system kernel. Just recently, the importance of secure virtualization and the potential of microkernels was shown by a commercially available smartphone called *SiMKo3*, providing a private and a business virtual machine running side by side, securely isolated from each other [8]. Working environments handling sensitive data are kept safe in a restricted, secure VM. Private or untrusted activities like multimedia can take place still on the same device, but in their own VM without the restrictions, but separated from sensitive data. The technology behind this *high security phone* is a combination of virtualization and security components. An underlying *L4 microkernel* developed at the TU Dresden implements the basic mechanisms needed to concurrently operate two securely isolated instances of the *Android* mobile operating system on the high security phone.

Although the technique used in the secure smartphone is a form of virtualization, it still requires modifications to the operating system (i.e., it uses *Paravirtualization*). With the aim of supporting unmodified guest software, another architecture was developed at the TU Dresden from scratch with strong focus on hardware-assisted virtualization. A minimal microkernel together with a user-space application formed the base of the *NOVA OS Virtualization Architecture (NOVA)*. To date, it supports running unmodified guest operating systems including Linux and several L4 variants.

While NOVA already supports multiple processors and the user-level environment is able to run different services and virtual machines on different physical CPUs, the virtual machines themselves have no true multi-core support. Although several virtual processors can be configured, they all execute on the same physical core. Solutions based on NOVA would thus not be able to make use of one of the main performance improvement mechanisms available today.

To remove this limitation, virtual processors have to be distributed among the physical ones and the virtualization environment needs to provide efficient and scalable mechanisms to pass the gain in resources on to virtual machines. Extending the findings of power management in a virtual machine being a problem [35], the scalability of virtual multi-core systems is not only determined by the frequency of virtualization events, but also by the degree of parallelism they can be served in. The goal of this thesis project is to enable guest software to leverage multiple processors as well as to identify performance-critical code paths within the virtualization layer that need to be improved with respect to scalability characteristics.

After describing the technical background in terms of concepts and software in Chapter 2, I will describe the design space, the proposed solution and its implementation details in Chapters 3 and 4. After evaluation of the achieved functionality in Chapter 5, an overview about future work in Chapter 6 will lead to a conclusion and outlook in Chapter 7.

# 2 Technical Background

In this chapter, I will introduce the technical prerequisites my work is based on and describe the concepts of the environment it has been developed in. Besides explaining essential terms and surrounding software components, a short section about related and similar solutions will help classify the scope of the thesis.

## 2.1 Basic Concepts

### 2.1.1 Microkernels

One of the main reseach areas of the *TU Dresden Operating Systems Group* (TUD:OS) is reducing code complexity of security-critical systems. A suitable approach to this was found in the use of *microkernels* [1]. Already in 1995, Jochen Liedtke published a description of concepts and construction primitives that have to be incorporated into such microkernels in order to provide an efficient and secure base for an operating system on top of them [22].

Liedtke stated that “a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system’s required functionality.”, thereby defining the minimality criterion of those kernels. The identified abstractions to be provided by the kernel are *address spaces, threads* and *inter-process communication* (IPC). Based on these mechanisms, services outside the kernel can be implemented to provide for example device drivers, user interaction, etc. By adhering to the *Principle Of Least Authority* (POLA), where every subsystem is only given the minimum amount of rights, componentized systems can be built securely.

As a result of the different approach pursued by microkernel-based systems, software like device drivers would have to be completely rewritten in order to function with the interface provided by the kernel. However, commodity operating systems often already include such drivers and applications. To be able to reuse this legacy software on top of a microkernel, several techniques have been researched, including the use of virtualization to use the original operating system as a wrapper around the component that should be reused.

### 2.1.2 Virtualization

The use of virtualization to create a simulated environment mostly transparent to the software running on top of it (i.e., to *guest software*) is by no means a modern concept. It has been pioneered by IBM in mainframe systems in the 1960s, and in 1973 Robert P. Goldberg defined the architectural principles of virtualized systems in his PhD thesis [10].

To enable the virtualization environment to provide the illusion of a real system, *privileged* instructions that are critical to the host system (e.g., they mask hardware interrupts) have to be protected from being used by guest software, although these instructions are expected to have the desired effect. A common solution to this gap is called *trap and emulate*, which means that every operation that must not be executed directly on the hardware is being trapped to the *Virtual Machine Monitor* (VMM), the software part responsible for maintaining the virtual machine state for the guest. The trapped instruction then gets emulated, before the guest resumes its operation. If the VMM models a complete system such that guest software does not need to be modified, the technique is also called *faithful virtualization*.

A very common architecture for commodity and server hardware is *x86*, first introduced in the 8086 processor by Intel [14]. The original design, however, was not entirely compatible with *trap and emulate* due to so-called *virtualization holes* [26]. These are instructions with different effects depending on the privilege level, but do not trap. One possible solution to close this gap was binary translating the executed code, replacing the affected instructions by code correctly handling the different behavior. Amongst other problems, this caused early virtualization solutions to be very complex, which led processor vendors to develop virtualization extensions for their hardware (Intel VT-x [15], AMD SVM [2]). By introducing a new mode of operation (*guest mode*), it was possible to execute guest code natively even in privileged mode, while additional control structures and instruction set extensions allowed for the necessary traps, transferring control to the *host mode*, where the virtualization layer then emulates the expected behavior. In succeeding generations of hardware virtualization support, not only complexity, but also performance issues became subject of improvements.

The aforementioned problems and evolved solutions are not limited to the x86 architecture, but can be found in a very similar shape in other systems (e.g., POWER, ARM, MIPS), as well. However, this thesis project will remain focused on VT-x enabled systems.

### 2.1.3 NOVA OS Virtualization Architecture

In the context of secure virtualization environments, the TUD:OS group developed a research vehicle with the aim of a minimal *Trusted Computing Base* (TCB) [28]. Componentizing the virtualization layer into separate subsystems already led to a shift in terminology in related publications [21]. Adhering to their definition, in this thesis, the terms *hypervisor* and *VMM* will describe distinct pieces of software. The privileged part of NOVA is called the *microhypervisor* due to its microkernel approach and is responsible for secure and efficient isolation and communication mechanisms. In contrast to the privileged part of other solutions like *KVM* or *Xen*, the microhypervisor only implements critical parts with respect to security, isolation, and performance. Examples are the management of virtual machine control structures, memory management, scheduling, and driving physical interrupt controllers. Device drivers or virtualized components that are not considered critical are provided by the user-level environment.

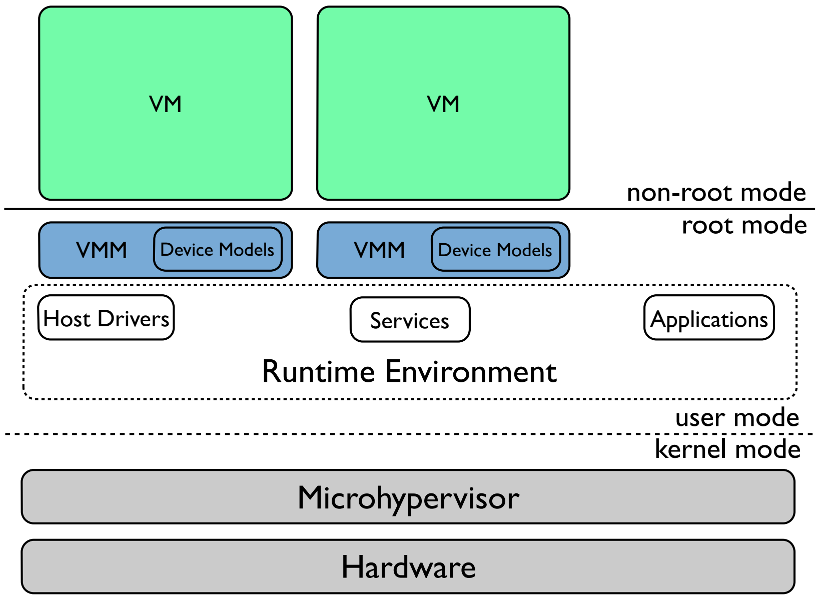


Figure 2.1: Overview of NOVA design

#### 2.1.3.1 NOVA Microhypervisor Primitives

To provide a basic understanding of the responsibilities of the hypervisor, this short section shall explain the essential mechanisms provided and how they relate to virtualization events.

Following the microkernel approach, the NOVA microhypervisor provides the three basic abstractions:

- *Address spaces* isolating processes are called **Protection Domains** (PD),
- *Threads* are formed by **Execution Contexts** (EC) and **Scheduling Contexts** (SC),
- *Inter-process communication* is established via **Portals**.

The traditional notion of a thread as an *activity* that can be dispatched by the OS scheduler is equivalent to an EC with an SC attached to it. Communication between components is performed by assembling a message payload and then calling the respective portal using a *system call*. These portals have a handler EC assigned which then gets invoked. The differentiation between EC and SC is important in this process, because the caller *donates* its scheduling context to the handler. Consequently, the communication resembles a simple remote function call and the request is being handled with the priority and on the CPU share of the initiator. A more in-depth discussion of architectural and conceptual details can be found in [28].

#### 2.1.3.2 VM Handling

Each virtual machine in the system is managed by a user-level (i.e., unprivileged) VMM named *Vancouver*. All components of a physical system (e.g., functionality integrated in the motherboard chipset, storage and network devices, etc.) are emulated by the VMM. Connection to real peripherals like physical network or storage can be provided by dedicated user-level services.

The VMM itself is based upon a design resembling the architecture of a real system as illustrated by Figure 2.2. Central element is a collection of busses (e.g., memory, I/O, etc.), connecting the various components within the VMM, the *virtual motherboard*. Device models can be attached to those according to their needs. Executing device code is done by sending messages of the respective type, although this kind of message-passing differs from the traditional meaning. Rather than sending a message to another entity and letting it process it, the message infrastructure in Vancouver translates into pure function calls at compile time.

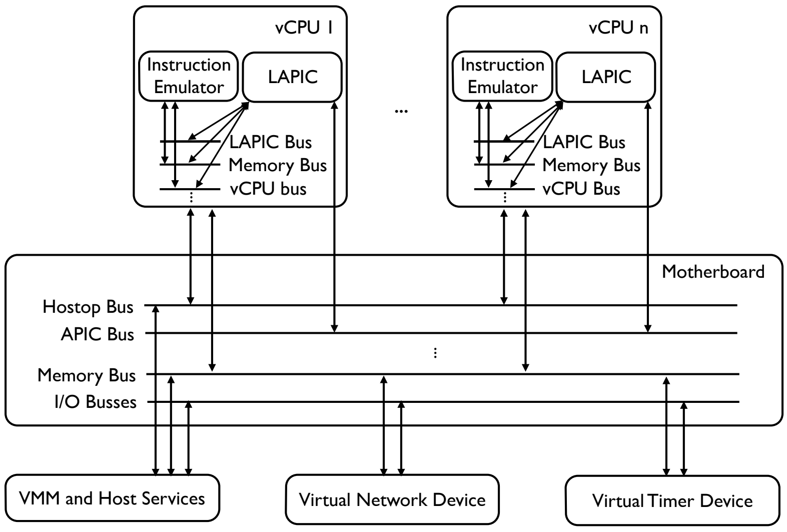
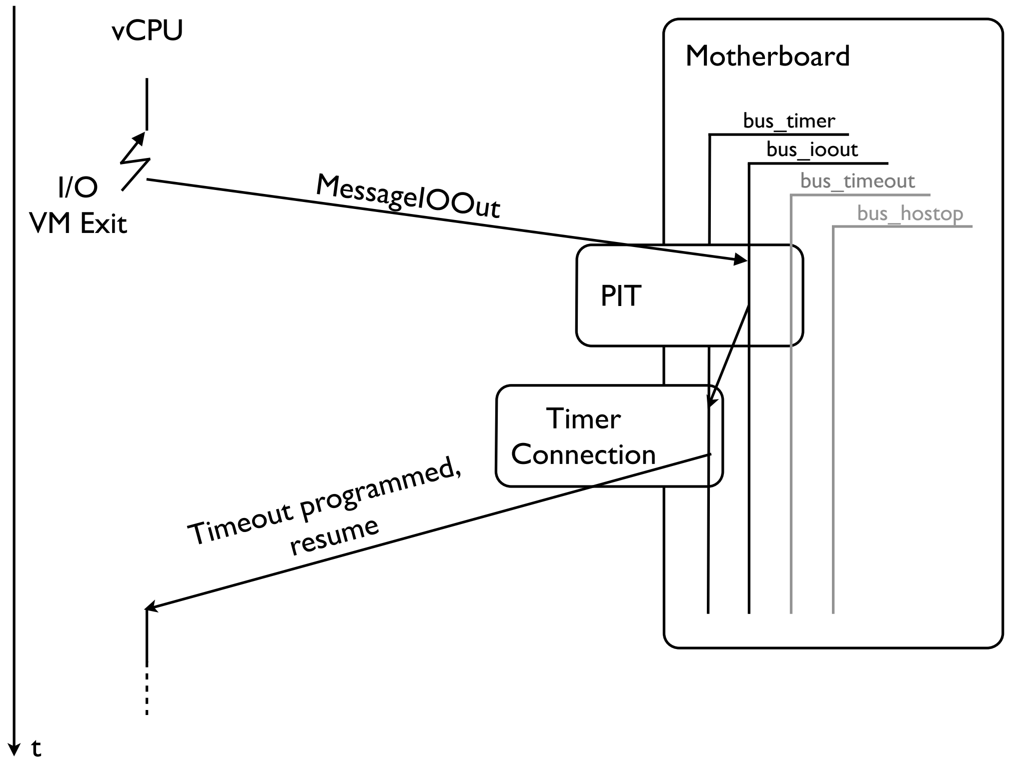


Figure 2.2: Overview of Vancouver bus topology (excerpt)

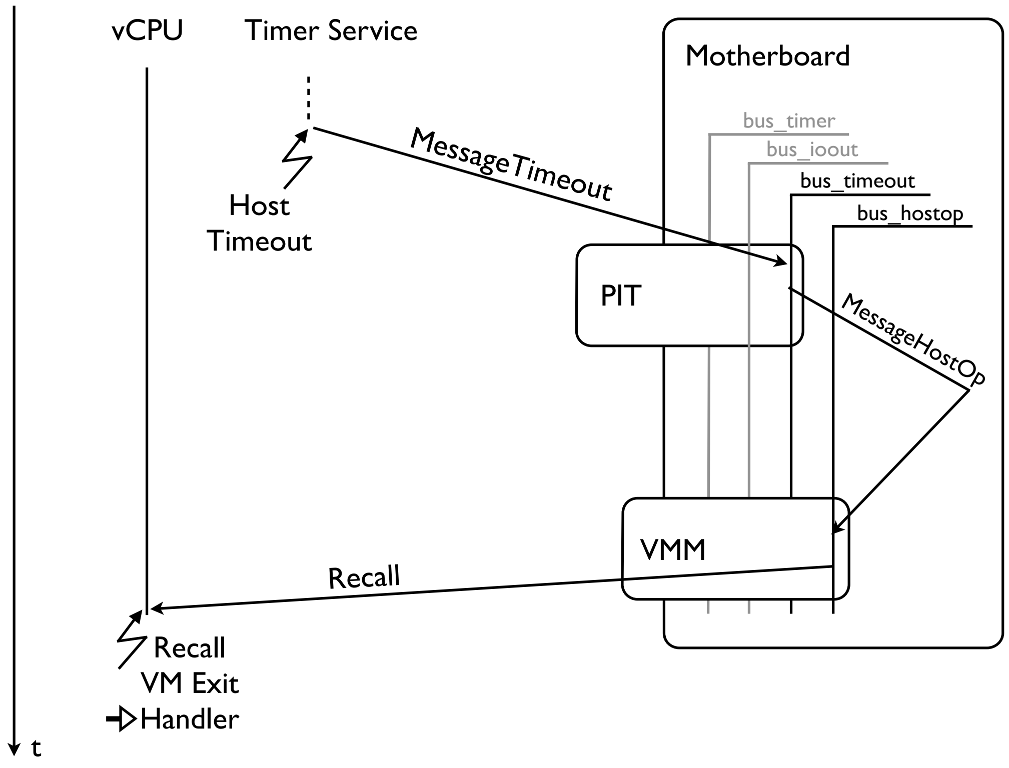
As a better understanding of how device emulation is achieved in this particular environment is essential for the later described problems and solutions, Figure 2.3 on the next page illustrates a simplified scenario of a VM programming a virtual *Programmable Interval Timer* (PIT), using the example of setting up a one-shot timer interrupt at a specific time. The OS issues a port I/O write to the PIT, loading a counter register with the respective timeout value, and then waits for the timer interrupt to occur.

Every device emulation event originates from either a trapped guest instruction or an external event signalled by an interrupt. Whenever this trap was related to the virtual

Figure 2.3: Example scenario of device emulation in Vancouver



*(a)* Program virtual timer



*(b)* Process virtual interrupt

machine and is not handled directly by the hypervisor, the control flow will be directed to the VMM by calling the *portal* installed for the respective event. Effectively, the vCPU thread then runs VMM code rather than guest code. In the particular example of programming a PIT, the VM exit would result in executing the I/O exit portal handler, with the information about the exit reason transferred as message payload. Receiving a port I/O event, the handler code now assembles a *MessageIOOut* and places it on the bus. Within the device models attached to it, the destination of the message is checked and appropriate actions are taken. In the virtual PIT, the port write results in a call to the host timer connection to effectively simulate a counter, eventually resulting in a virtual interrupt. After the vCPU has completed all those steps, it resumes operation directly behind the trapped instruction.

Instead of trapping due to guest instruction execution, the guest can be deliberately called in to execute VMM code to react to external events. If for instance the timeout programmed earlier finally expires, the timer service will be notified and now on its part use the timeout bus to emulate an elapsed countdown in the virtual PIT. After prioritizing and sanity checking, the thread will then finally issue a request to the vCPU to exit and process the virtual interrupt request.[1] This mechanism is called a *recall* and is achieved by sending a *MessageHostOp* on the bus. In the assigned recall portal, the vCPU would then take appropriate action.

### 2.1.4 Symmetric Multiprocessing (SMP)

During the last decade, single-processor performance began to level as technology hit the *power wall* [12, p. 4]. As a result, processor design started to integrate multiple compute units in order to improve performance. Most commonly, multiple identical processors, connected to a unified main memory, are available to a single operating system. This ongoing trend of *Symmetric Multiprocessing* resulted in multi-cores being found almost everywhere, even in mobile phones. It is only a logical consequence that virtualization should also leverage this new source of performance both for running several VMs on one host and even allowing virtual machines equipped with multiple cores.

Basically, the existing concept was easily extensible to SMP by just assigning more than one *virtual CPU* (vCPU) to the guest. However, synchronization issues in the emulation process explained in Section 2.1.3 now become more prominent and, most importantly, can be major reasons for lacking scalability. With $n$ vCPUs trying to access the same device model and possibly a host service doing the same, $n+1$ instead of two threads are executing critical paths at the same time. Obviously, this problem becomes more prominent, the larger the SMP system grows. It is therefore of high importance to a holistic VMM design to use efficient mechanisms to allow for minimum overhead.

#### 2.1.4.1 Scalability and Synchronization in SMP Systems

Merely adding $n$ additional compute units to a system does not automatically improve performance by the factor $n$. Software running on multiple processors has to divide the work

[1] For simplicity reasons, minor intermediate steps of bus interaction are omitted.

across the available cores in order to achieve maximum performance. Generally speaking, the speedup resulting from an improvement is determined by the amount of acceleration the enhancement provides, and the fraction of time the improvement can be used [12, p. 46], known as *Amdahl's Law* [3]. In the specific case of multiple processors, the theoretical maximum speedup $S$ depending on the number of cores $N$ and the parallel fraction of the algorithm $P$ can be expressed using an equational form of the law as given in Equation 2.1.

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}} \tag{2.1}$$

One important influencing factor of the serial fraction is the amount of synchronization between concurrently executing threads. Critical sections that must never be executed by more than one thread at a time are by definition a serial part of the algorithm. There exist several techniques for protecting critical sections:

- Coarse-grained locking: Data structures or objects are protected by a single lock, guarding larger sections of the program even though the critical part may be significantly smaller.
- Fine-grained locking: Locks are held specifically where it is necessary. Usually this results in a higher number of lock objects and a more complex design, but allows for more parallelism.
- Atomic operations: Locks are applied on instruction level. Common hardware architectures provide instructions or instruction modifiers that guarantee the result to be visible atomically with no intermediate inconsistent states.
- Transactional memory: Critical sections can be marked as *transactions* that get rolled back and re-executed when concurrent updates led to inconsistencies.

Besides compromising scalability, locking mechanisms can also entail execution flow issues. Lock implementations have to deal with *fairness* (i.e., every thread can enter the critical section after a comparable waiting time) and *deadlocks* (when locks form a circular dependency and no one can proceed). Atomic operations can be used to tackle these issues by enabling algorithms to present with one of the following properties:

- Lock-freedom: Eventually, *at least one* thread makes progress.
- Wait-freedom: Eventually, *every* thread makes progress.

In fact, every wait-free algorithm is automatically lock-free [13, p. 60]. When at least one thread always advances in its program execution, deadlocks are not possible. But it can still happen that some threads get *starved* because another thread always interferes with the successful finalization of their update. Wait-free algorithms, on the other hand, provide fairness in that starvation is not possible.

However, the more parallelism is allowed, the more complex thread synchronization gets, and potentially produces more locking overhead. Furthermore, there exist cases where for example wait-freedom is not feasible. This leaves the fairness to be achieved by other measures.

## 2.2 Related Work

Before discussing the design possibilities to be considered in the design of an SMP-enabled Vancouver VMM, a short overview of existing research and development in this field shall offer examples of similar solutions as well as surrounding research topics not covered in this thesis.

### 2.2.1 Solutions in Other Virtualization Architectures

Other virtualization solutions mostly use locks for *mutual exclusion* (also called *mutex*) in the emulation code paths. For example, *qemu* [4] uses a global mutex for everything within the VMM. Traditionally, all guest execution and I/O handling has been performed in one single thread. Even with multiple virtual CPUs, this single thread would multiplex execution among all activities. Later, the architecture was migrated to using a multi-threaded model. Communication with external entities (i.e., files, host network, etc.) are now handled by an *I/O thread* which is blocked until it gets assigned work. Every virtual CPU is executed by a dedicated thread. However, all code paths of device emulation and I/O thread communications are still synchronized using the global mutex. The original design of qemu as an application running in user-space entirely achieved x86 virtualization by a component called *Tiny Code Generator* (TCG), providing dynamic binary translation [5]. Even though parallel execution of vCPUs is supported, the TCG is not thread-safe and therefore serializes guest instruction whenever such translation is necessary [27].

To be able to benefit from multiple CPUs, the second mode of operation uses the *Kernel-based Virtual Machine* (KVM) [11], which features hardware-assisted virtualization and fine-grained locking of affected subsystems or critical sections and therefore a higher degree of parallelism in virtualization handlers. Guest execution control is done using system calls on a special *device node* `/dev/kvm`, provided by the kernel. Whenever guest execution causes a VM exit, the KVM kernel module gains control over the system and decides which action should be taken according to the exit reason. If the event is handled by the kernel (e.g., it was an access to the interrupt controller), KVM applies the respective modifications to the guest and device model states and resumes guest execution. Events that are not handled by the kernel are propagated into userspace, where virtual devices like input, disk or network are provided. Figure 2.4 on the facing page depicts the execution loop as found in the publication about the architecture and operational details of KVM [18].

The architecture of qemu in combination with KVM is very similar to NOVA in that a user-level component provides VMM functionality and interfaces with the hypervisor via system calls. However, integrated device models into KVM (PIC, APIC, I/O APIC) and the code complexity of the Linux kernel where KVM resides form two fundamental differences to the microkernel approach.

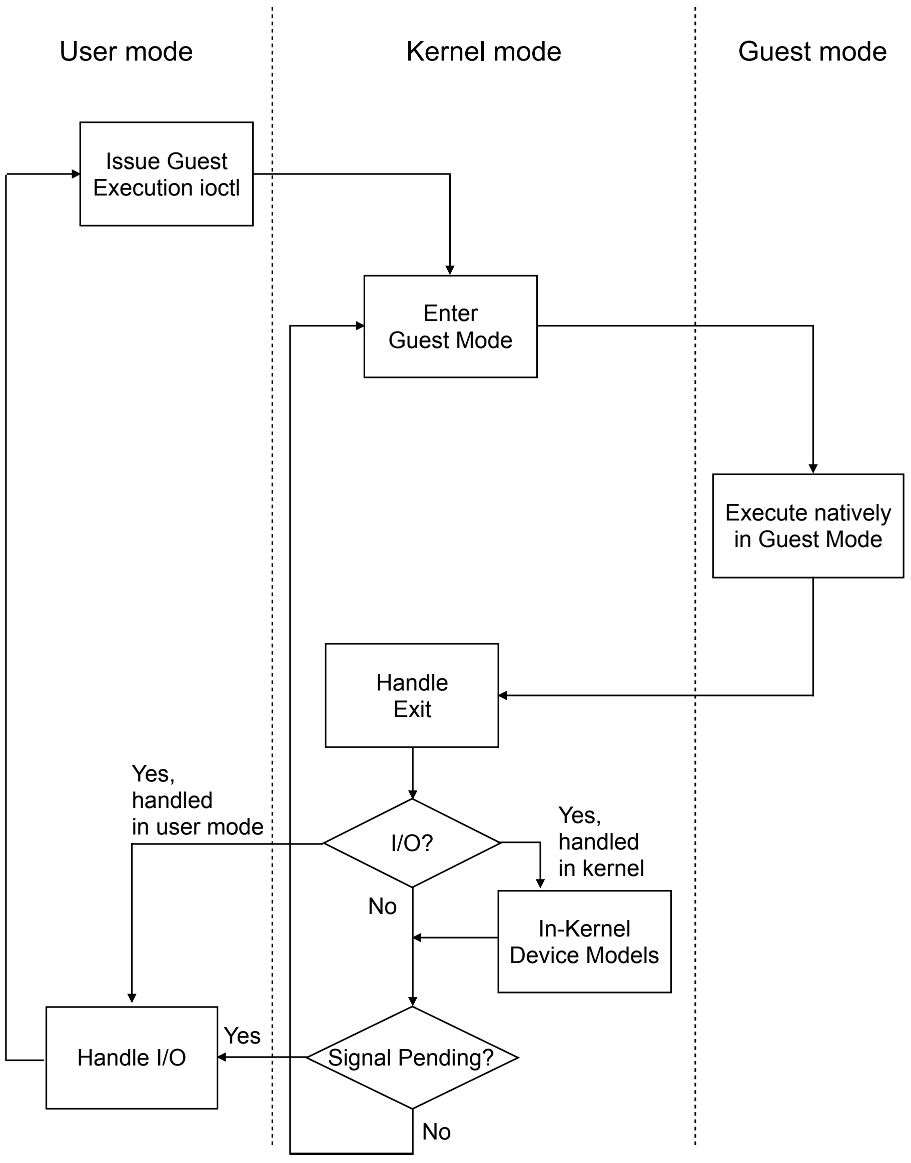


Figure 2.4: Guest execution loop in KVM [18]

### 2.2.2 Virtualization Overhead Impairing SMP Scalability

While synchronization in VMM functionality is not being actively researched in publications, dependencies between multiple processors inside a VM have been analyzed. The findings of a severe slowdown of a virtualized benchmark compared to native execution could be traced back to a large overhead when virtual CPUs get shut down due to guest synchronization (i.e., mutex blocking). Rescheduling and wake-up IPIs were identified as major contributors to the overhead responsible for the performance degradation [35]. A similar effect was found with *OpenMP*-workloads, where the particular implementation of a barrier involved the hypervisor and thereby caused parallel execution to be even slower than the sequential one [29]. These publications show that virtualized SMP is not always faster than uniprocessor execution and that exit handling paths can have a severe impact on scalability.

### 2.2.3 Multiple Virtual Machines on the Same Host

A very common usage of virtualization, especially in cloud server architectures, is to consolidate several machines that are not fully utilized into one large server with higher utilization to reduce waste of energy resources. Hence, servers often run multiple virtual machines concurrently on the same host.

As long as each virtual machine has its own distinct set of physical cores to execute on, there is no special treatment necessary. However, idle VMs again impede maximum global utilization of the system, which was the goal of the consolidation in the first place. Therefore, vCPUs should share physical cores. But this could have serious effects on the performance of a VM, if for instance the virtual CPU holding a kernel lock gets preempted in favor of a different VM. All other vCPUs of the first VM then potentially do *busy-waiting*, thus wasting CPU cycles without doing any useful work. This effect is called *Lock Holder Preemption* [31].

Consequently, most ongoing research revolves around the question how to schedule vCPUs of different VMs on the same host achieving maximum utilization of available resources. Especially when overcommitting the system (i.e., assigning more vCPUs than physically existing processors), this issue becomes very challenging [34].

While in KVM vCPU threads are being treated as regular threads subject to the Linux scheduling algorithm, the open-source hypervisor *Xen* even provides three different dedicated CPU schedulers, developed over several years [30]. They allow for assigning weight values to VMs that the scheduler then uses to allocate CPU resources accordingly. The main advantage of their most recent version, the *Credit Scheduler*, is automatic load balancing on multiple processors.

Next, I will present possible solutions to improve the current proof-of-concept SMP implementation of Vancouver, enabling it to run highly scalable VM configurations utilizing a high number of cores.

# 3 Design

As described in Section 2.1.3.2, virtualization events are being handled by several dedicated threads in the VMM. Besides the vCPUs, each of which represented by its own handler, connections to I/O and host services are established via another set of threads (e.g., the timer service handler). Because all those threads potentially access the same data structures, it has to be guaranteed that no data corruption can occur. Even if they all execute on the same physical CPU, preemption can still lead to quasi-concurrent access. However, the guest operating system only implements synchronization mechanisms with respect to the behavior of physical hardware. Consequently, the VMM has to provide device models that behave as expected. In order to be able to meet this requirement, two properties of a virtualized system have to be considered:

- Virtual devices are implemented in software. In some cases, they behave differently from hardware with respect to timing and atomicity.
- Some devices are accessed by not only the vCPU threads, but also host services.

To provide the guest with the virtual system it expects, additional protection against concurrent accesses has to be in place such that the virtual hardware *appears* to behave like real hardware, although the underlying mechanisms might differ. Intended as a proof-of-concept, the original VMM used a global semaphore which synchronized all threads in the VMM, i.e., vCPUs and services.

However, the goal of efficient virtualization is to introduce as little overhead as possible. When virtual devices are emulated in software, an inherent virtualization overhead is introduced. Even though this overhead can be mitigated to a certain degree by leveraging hardware extensions and optimizing the device model code in general, there will always remain a constant performance loss compared to native execution when there are virtualization events to be handled. Moving towards multiprocessor systems, it is important to avoid introducing additional overhead that might even increase with the number of CPUs. The reason why such performance issues can arise is due to synchronization among the concurrently executing threads. It is the first and foremost goal to exploit as much parallelism as possible and not to have vCPUs wait for events unrelated to their own execution. The global semaphore in the original implementation is not optimized in this regard as it serializes everything within the VMM.

In the remainder of this chapter, I will analyze the implications of the serializing behavior and discuss solutions to how the VM exit and device access handling of virtual CPUs can be done more efficiently.

## 3.1 Analysis of Contention on the Global Lock

Before developing a design to solve the problem of global lock contention, it is necessary to assess the way it manifests itself when a virtual SMP system is running and get an idea of what can be achieved by a better design. In the first stage, I distributed the vCPUs among distinct physical cores and ran a kernel compile with up to eight cores to evaluate the scalability that can be achieved with this approach. During the experiment, I took samples of the amount of time threads were blocked on the global semaphore. With *Amdahls's Law* in mind, as described in Section 2.1.4.1, this is expected to be the factor that influences scalability the most, because it is in essence lost computation time and directly affects the parallel fraction.

Figure 3.1 on the next page shows the average number of clock cycles that a thread had to wait for the global lock. The tail of the distribution (i.e., wait times longer than 50000 cycles) is aggregated into the last data point. It can be clearly seen that with a higher number of cores, longer wait times become more frequent. The tail can be seen increasing from zero for the single-processor run up to over 6000 times for eight CPUs. Although being only an excerpt of the complete run, the distribution clearly indicates the potential for performance improvements when removing the global lock and replacing it with a more targeted synchronization mechanism. Figure 3.2 on the facing page supports this assessment, as it shows a continously growing performance loss compared to native execution when scaling up to a higher number of cores, and sets the baseline that improved solutions will be compared against.

Having established that serializing all requests greatly impedes SMP performance, the global lock approach shall now be replaced by a more sophisticated mechanism developed in the process of this thesis. There exist two major categories in the design space:

- Inspired by the use of threads handling events related to external events, a dedicated *I/O thread* is installed. Virtual CPUs and other threads merely queue their requests and synchronize with the outcome, if necessary. While all requests are still being serialized, it is still possible to improve scalability using this model, because the vCPU only has to wait for a certain portion of its events.
- Requests of virtual CPUs and other service threads (e.g., the host timer connection) are handled in their own context. From accounting perspective, this approach is preferable, as the threads pay for their own I/O handling. However, synchronization overhead in the VM exit path directly reduces available compute time for the VM.

In the following sections, I will describe different possibilities in the two categories mentioned above, assess their advantages and disadvantages and identify potential difficulties in implementing them in the final solution.

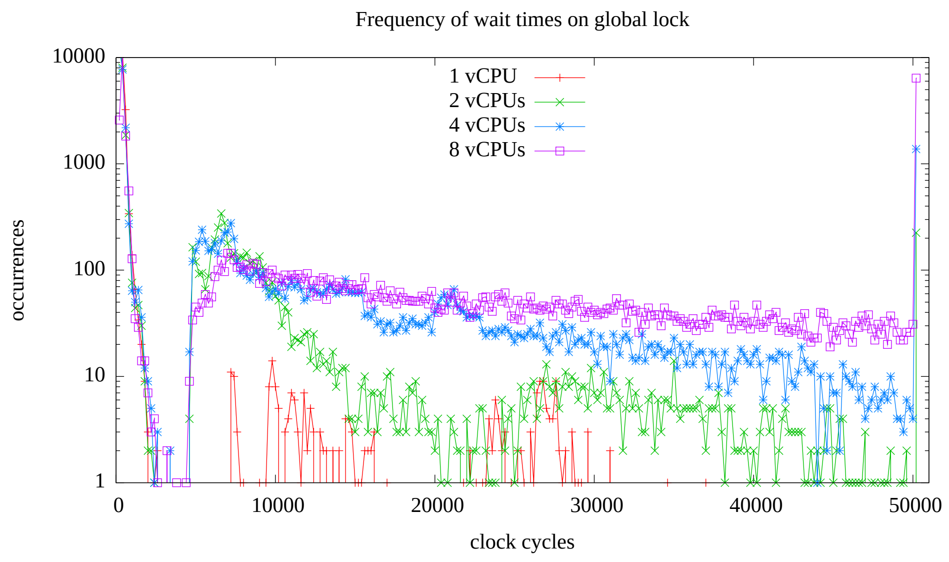


Figure 3.1: Frequency of lock wait times depending on lock contention

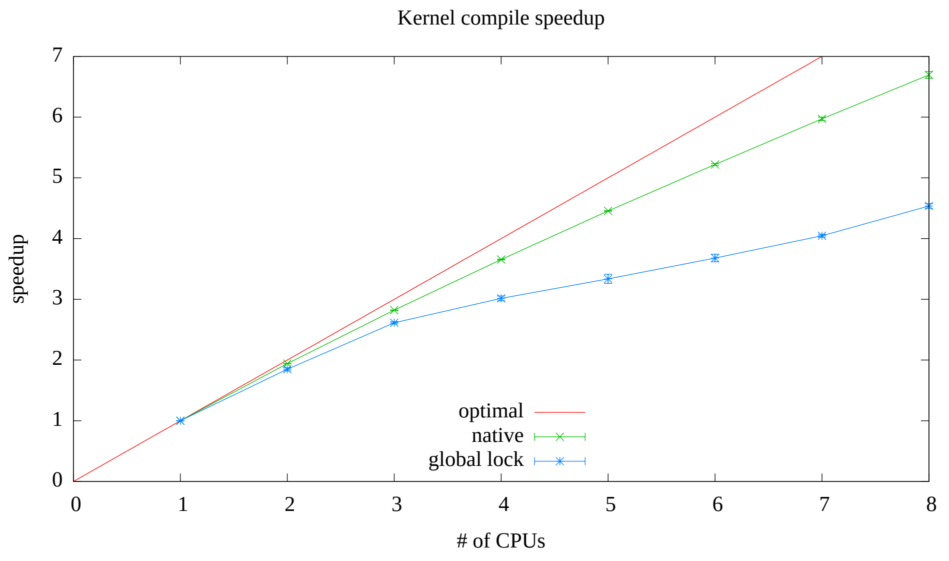


Figure 3.2: Native vs. global lock speedup of a kernel compile

## 3.2 Dedicated I/O Thread

Figure 3.3 on the next page shows a high-level overview of how the I/O thread approach works in general. Whenever a request has to be handled on behalf of a vCPU or a service thread, a queue element is created, containing all information needed to process it. After the enqueue operation, the client then waits for the handler to finish. The I/O thread maintains a simple *First In First Out* (FIFO) queue, handling incoming requests one by one. This way, it is guaranteed that device models are never being accessed by more than one thread at any point in time. When the request has been completed, the client gets notified and can extract the result of the operation. Effectively, this naive concept is no improvement over the global lock. In fact, it would even decrease performance because of the overhead of assembling and enqueueing requests, the I/O thread would simply act as a global lock.

### 3.2.1 Posted Writes

However, the enqueuer does not need to wait for the completion of all kinds of events. The I/O thread enables the synchronization algorithm to distinguish between *synchronous* and *asynchronous* requests. While *read* operations are inherently synchronous, most *write* operations can be handled asynchronously. As shown in Figure 3.4 on the facing page, the client can return to its work immediately after the enqueue completed. This method is also referred to as *posted write* and is found for example in the PCI bus. The overall design of this approach then resembles the *PCI root complex* of modern computer systems, which is capable of transparently buffering posted write transactions and thereby decreasing the latency of such requests. But is important to be aware of the implications of posted writes. Transparent buffering indicates the completion of the request before the effect is globally visible. It has to be ensured that subsequent read accesses to the same location process the latest data from the buffer, if necessary. The I/O thread design guarantees that reads issued after write requests are always executed *in-order* and never access stale data.

High-performance devices optimized for those PCI access latency properties fit the I/O thread model well. Because avoiding read requests can improve performance also when using physical hardware, drivers increasingly rely on write accesses as much as possible. With the advent of *MSI* and *MSI-X* (Message Signalled Interrupts) capabilities of PCI devices, shifting high-speed I/O handling towards a more write-oriented approach became possible. Instead of receiving a generic interrupt event and determining the cause by reading registers containing the device state, certain interrupt events can now be directly routed to the respective handler without the need for expensive I/O read operations. Drivers that are designed this way could then also benefit from the I/O thread approach.

### 3.2.2 CPU Assignment / Co-Location

Another aspect of a virtual machine using an I/O thread is the characteristics of resources assigned to it. While in the traditional approach the assignment of virtual to physical CPUs can be realized one-to-one, the I/O thread needs additional resources of its own. In order to provide a minimal latency, it should be run on a dedicated *I/O core* without interfering

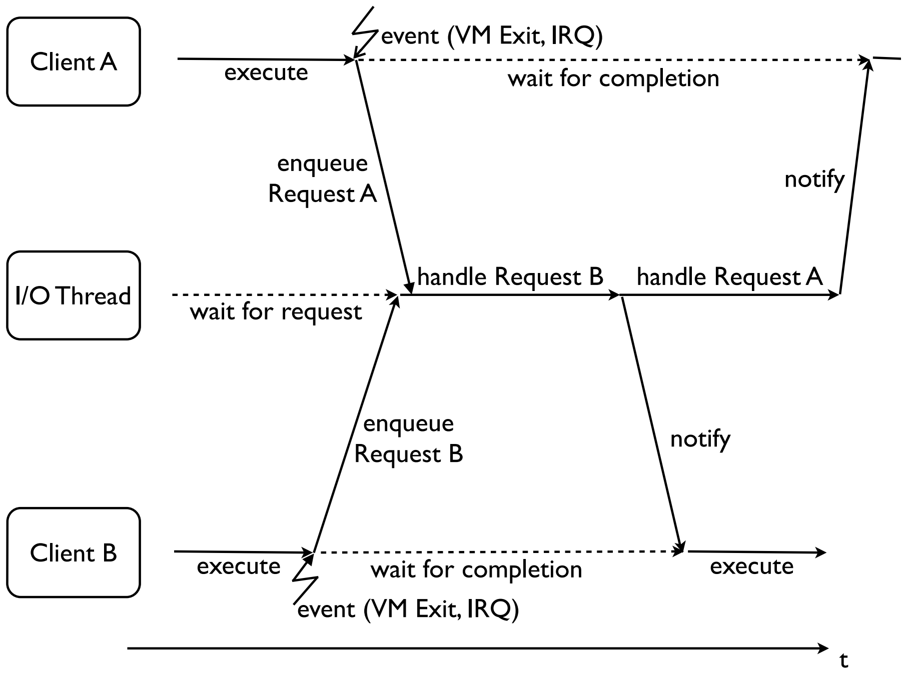


Figure 3.3: High-level overview of the I/O thread approach

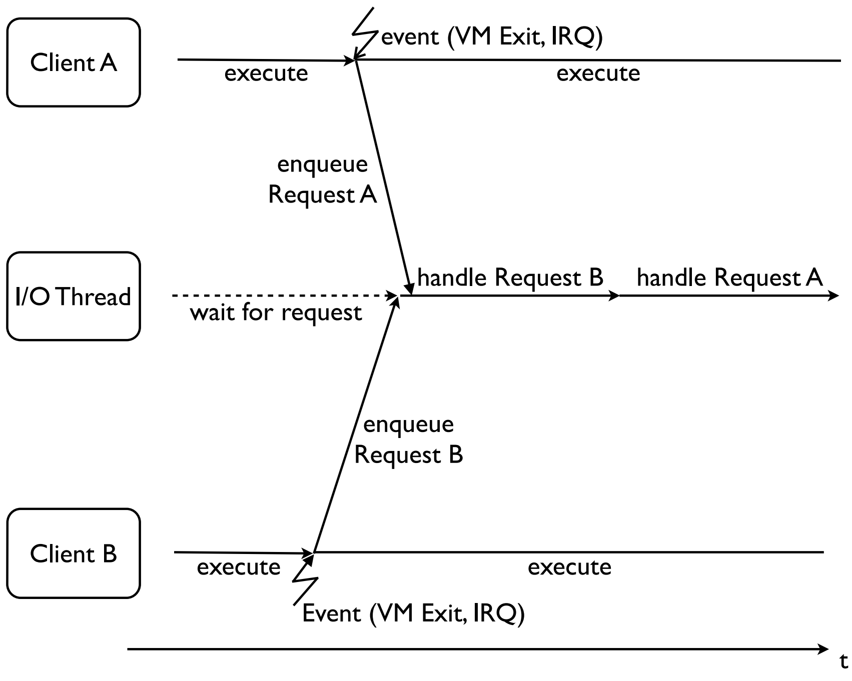


Figure 3.4: High-level overview of the asynchronous I/O thread approach

with virtual CPUs. Otherwise, a vCPU could be delayed by an entire timeslice in the worst case, which has to be avoided. However, this results in a different resource assignment. A virtual machine with the same number of vCPUs requires one additional physical core compared to the original case. Furthermore, if all $n$ available physical cores should be used, the virtual machine would only have $n - 1$ virtual cores at its disposal.

This factor has to be considered when planning a virtual SMP system using an I/O thread. When tackling scheduling issues of multiple virtual machines as mentioned in Section 2.2.3, this approach will add another vector to the problem, because now not only the vCPUs, but also the I/O threads have to be regarded. Within the scope of this thesis, the I/O thread is assigned to a dedicated core, while the virtual CPUs are distributed among the remaining ones.

While it is a comparably simple solution, the scalability of the I/O thread heavily depends on the workload characteristics. Predominantly synchronous requests decrease the performance and even in the asynchronous case, parallelism is not exploited at maximum level. All events are still handled sequentially, even when they would not interfere with each other.

## 3.3 I/O Handling in Initiator Context

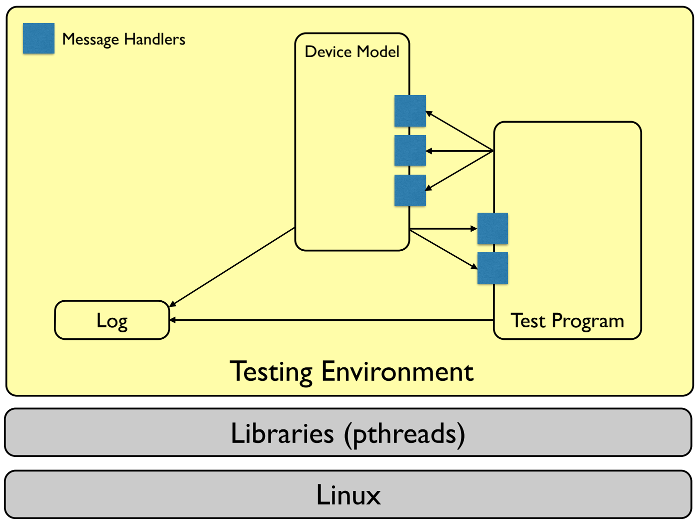


Figure 3.5: Design of synthetic testing environment

A solution which enables harnessing available parallelism more effectively can be found when threads handle their own requests. Rather than serializing them using a global lock, device models could be built in a thread-safe way by using atomic operations and fine-grained locking. In essence, only the *critical sections* have to be protected against thread

| **Parameter** | **Value** |
|---|---|
| compile time | 470s |
| exits | 867341 |
| average exit cycles | 2301 |
| CPU frequency | 2.67 GHz |
| synchronous messages | **759141** |
| thereof in vCPU subsystem | **728509** |
| asynchronous messages | 446213 |
| thereof in vCPU subsystem | 317191 |

Table 3.1: Characteristic numbers of single-core kernel compile

interference caused by parallel access. All non-conflicting paths can be executed in parallel. However, the major downside of this approach is the complexity. It is necessary to identify and correctly protect any critical code sections. While atomic operations can be integrated seamlessly, introducing fine-grained locking to the device models is likely to increase code complexity. Although the VMM design principle of message-passing function calls described in Section 2.1.3.2 allows for locking on the API level, it also has to be ensured that nested calls never lead to deadlocks, e.g. by releasing locks before sending a nested message.

The improved parallelism that can be achieved with this solution is attended by a high complexity. Even with extensive synthetic testing, hidden errors could manifest themselves only on certain machines under rare conditions. While the I/O thread is safe as per design, the fine-grained locking is difficult to verify and test.

### 3.3.1 Synthetic Testing Environment

In order to find and correctly protect critical sections in device models, I decided to create a simplified test environment. By using an isolated test program separated from the runtime environment and the hypervisor, it is possible to create targeted test scenarios restricted to the minimum of components around the device to be tested. Figure 3.5 on the preceding page shows how the message-passing API of the device models provided by the VMM is stress-tested in a controlled environment, while maintaining a log of the course of events in both the device and the test program. In the event of an observed failure, the log can be inspected to find the exact interleaving of code execution which caused the problem. Using long-running tests of the corrected version the modifications can then be verified.

## 3.4 Hybrid I/O Thread

Because the fine-grained approach is complex and difficult to maintain, having an encapsulated synchronization mechanism would be the desired solution. However, using basic characteristics of an example workload, it can be shown that the full I/O thread mechanism has severe performance impacts. During a Linux kernel compile benchmark, the main determining factors are the frequency of VM exits and their handler execution time. Table 3.1 on the preceding page shows numbers extracted from the original NOVA publication [28]. The amount and properties of messages the I/O thread has to process (i.e., if they are synchronous or asynchronous and if they are related to the vCPU subsystem) were collected during a sample kernel compile run of the original I/O thread approach. As the intended use of these numbers was only to determine relative ratios, absolute numbers did not matter and the fact that they were determined using a different setup is not of importance. Using the VM exit statistics shown in Figure 3.6, it is possible to predict the slowdown of the I/O thread depending on the ratio of synchronous to asynchronous messages.

$$\begin{aligned}
\textit{exits per second} &= \frac{\textit{exits}}{\textit{compile time}} = \frac{867341}{470} && = 1845.4 \\
\textit{exit handling per CPU} &= \textit{exits per second} \cdot \frac{\textit{average exit cycles}}{\textit{CPU frequency}} && = 0.159\%
\end{aligned}$$

Figure 3.6: Exit handling statistics of a kernel compile

To derive the projected speedup of the I/O thread for a given number of cores, the parallel fraction of *Amdahl's Law* is determined by the original parallel fraction of the algorithm, reduced by the amount of serialization caused by the synchronization mechanism. The model multiplies the exit frequency by the number of cores to determine the load on the I/O thread caused by VM exits. Although this might not be entirely the case in real setups, it is still expected to provide a reasonable estimate. By running a pilot benchmark under native Linux, the base parallel fraction of $P = 97.5\%$ could be deduced for a kernel compile. Figure 3.7 shows the calculation of the theoretical estimate for the maximum speedup of the I/O thread compared to native Linux.

$$\begin{aligned}
\textit{exit handling} &= \textit{exit handling per CPU} \cdot \#\ \textit{of CPUs} = 0.159\% \cdot 16 && = 2.54\% \\
\textit{synchronous percentage} &= \frac{\textit{synchronous messages}}{\textit{total messages}} = \frac{759141}{759141 + 446213} && = 62.98\% \\
\textit{parallel fraction} &= P \cdot (1 - (\textit{exit handling} \cdot \textit{synchronous percentage})) && \\
\textit{I/O thread } S(16) &= \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(1-95.94) + \frac{95.94}{16}} && = 9.94 \\
\textit{native } S(16) &= \frac{1}{(1-97.5) + \frac{97.5}{16}} && = 11.64 \\
\textit{performance loss} &= 1 - \frac{\textit{I/O thread speedup}}{\textit{native speedup}} = 1 - \frac{9.94}{11.64} && = 14.6\%
\end{aligned}$$

Figure 3.7: Speedup calculation for 16 CPUs using parallel fraction

| Solution | Pros | Cons |
|---|---|---|
| Fine-grained | Best scalability | Complexity<br>No incremental deployment<br>Error-prone |
| Full I/O Thread | Simplicity<br>Robustness | Performance issues<br>More complex scheduling<br>Fewer vCPUs |
| Hybrid I/O Thread | Performance<br>Flexibility<br>Incremental deployment | More complex scheduling<br>Complexity (partly)<br>Error-prone (partly) |

Table 3.2: Pros and cons of the different solutions

As Figure 3.7 on the preceding page shows, the performance impact of the full I/O thread approach is significant. Mainly caused by the high percentage of synchronous messages, the achievable speedup is estimated to be located between native and global lock results. To assess the potential of optimizations, I used the parameters listed in Table 3.1 on page 19 to predict the improvement enabled by a lower percentage of synchronous messages. If it were possible to leave out the entire vCPU subsystem[1], the synchronous portion would drop to $\frac{759141-728509}{759141+446213} = 2.54\%$. In an analogous calculation using *Amdahl's Law*, the parallel fraction would increase to 97.44%, allowing for $S(16) = 11.64$.

Given the high potential for performance improvement by excluding devices or subsystems from the I/O thread, it is reasonable to choose the design of it with this mechanism in mind. Code paths bypassing the I/O thread correspond to the fine-grained approach and hence have to be synchronized explicitly.

This solution of combining both approaches would mean to use a dedicated I/O thread design which is capable of leaving certain devices or code paths to be executed by the initiating thread. The underlying principle is comparable to the one used in similar scenarios like *KVM*, where performance-critical code paths are executed directly in the kernel, protected by fine-grained locks, while the rest is handled by *qemu*'s I/O thread infrastructure.

A perfect example to illustrate this concept is the *ACPI*[2] *Power Management Timer (ACPI PM)*. The device model integrated into the VMM merely acts as an up counter with a fixed frequency of ~3.58 MHz, using the host TSC as timebase. Being read-only, it does not need to be synchronized at all. With a carefully designed hybrid I/O thread, it can simply be excluded from the queue and handled directly by the virtual CPU instead, as shown in Figure 3.8 on page 23. Analogously, thread-safe device models or even entire subsystems can be configured to bypass the I/O thread. This way, the fine-grained solution could be implemented incrementally, taking out entities which do not need synchronization by the

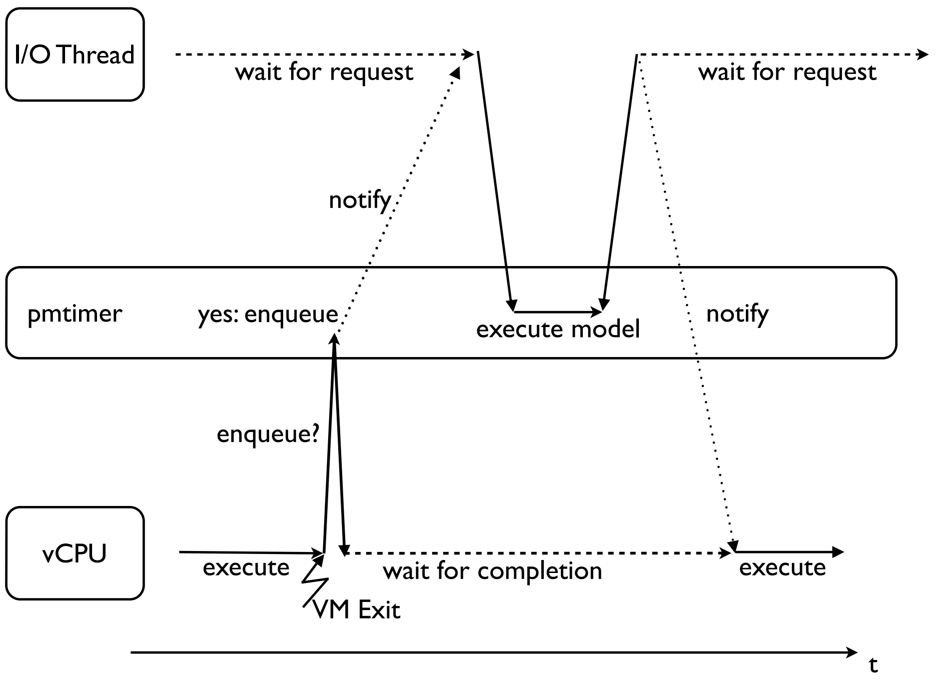
[1] For simplicity reasons, excluded messages are counted as asynchronous requests.
[2] Advanced Configuration and Power Interface

I/O thread one by one. The major advantage of the hybrid model is that it is safe from the beginning and is expected to allow for higher scalability even in the basic version. Furthermore, it encapsulates most code complexity in one piece of new software, rather than modifying all the existing code.

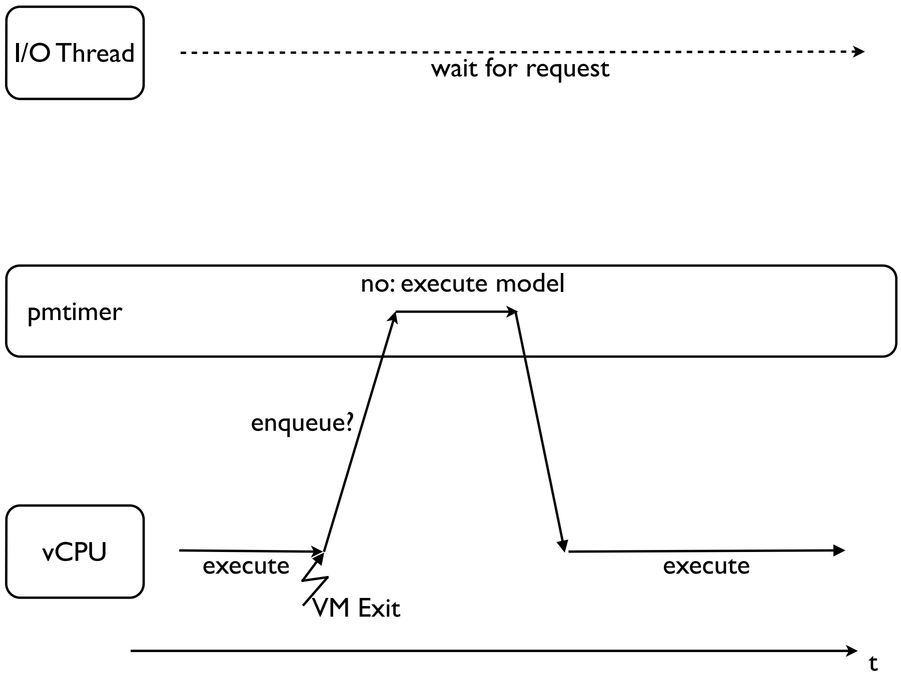
## 3.5 Summary

In this chapter, I described the possible solutions for improving the virtual SMP performance. The advantages and disadvantages of the three approaches are listed in Table 3.2 on the preceding page. While the full I/O thread is still expected to show performance issues and a purely fine-grained locking scheme is very complex and has to be deployed all at once, I chose the hybrid approach as the best compromise in between. This way, the complexity and extensive testing can be kept to a minimum, incrementelly applied only where the scalability depends on it.

In the following Chapter 4, I will describe the resulting implementation of the hybrid I/O thread solution in the existing VMM environment in greater detail. Complexity and particular problems when configuring specific devices for bypass are also covered.



*(a)* I/O Thread



*(b)* Bypass

Figure 3.8: Example I/O thread bypass flow for the ACPI PM timer device

# 4 Implementation

As laid out in Chapter 3, after distributing the vCPU threads among multiple physical cores, the globally serializing semaphore has to be replaced by a more sophisticated mechanism in order to allow for maximum scalability. The chosen solution of a hybrid I/O thread approach could be implemented incrementally. In this chapter, I will point out the most challenging aspects of implementing SMP support for a virtualized guest operating system. After a short recapitulation of the basic SMP implementation I will describe specifics of the I/O thread in greater detail. Afterwards, a detailed demonstration of the hybrid bypassing capabilities using the example of the PIC subsystem will reveal both the flexibility of the I/O thread and the complexity of the fine-grained locking approach.

## 4.1 CPU Assignment

The current version of the Vancouver VMM supports multiple virtual CPUs for guest operating systems. Each virtual CPU is represented by a dedicated vCPU thread. It is up to the user-level environment to distribute these threads among one ore more physical CPUs (pCPUs). At the time I started this thesis, there existed two such environments with SMP support, the *NOVA UserLand (NUL)* and *NOVA Runtime Environment (NRE)*. Because Vancouver was extracted from NUL recently into a standalone component called *Seoul* and only NRE has already been migrated to use this new version, I focused on NRE for the course of this project.

However, currently both NUL and NRE use different physical cores only for entire instances of Vancouver (i.e., VMs). Within one instance, all vCPU threads are assigned to the same core. This limitation effectively results in a proof-of-concept implementation. The performance of VMs with more than one vCPU on the same physical core is expected to be worse than with only one vCPU, because the guest operating system will schedule timer and scheduling interrupts to all the virtual processors, deploy synchronization mechanisms and distribute work among them even though the theoretical maximum of CPU performance is still only that of one single physical processor.

Fortunately, this limitation can be easily removed by two simple changes to the glue code connecting the Vancouver VMM to the rest of the userland. These changes consist of assigning vCPU threads to different physical CPUs and ensure that every vCPU thread is able to receive timer messages from the host system. For simplicity, I decided to use one so called *Timer Session* per pCPU, because it was the least intrusive change and the focus of this thesis is the VMM itself, not the software components around it.

## 4.2 Dedicated I/O Thread and Posted Writes

Leaving the existing codebase largely untouched was one of the main goals behind the I/O thread implementation. Therefore, I decided to directly hook into the *DBus*[1] (DeviceBus) infrastructure which connects host services and device models to a virtual *motherboard* as described in 2.1.3. By adding a new message type called *MessageIOThread* and modifying the *send* methods of the *DBus* class, I equipped the bus system with two *callbacks*:

- The *enqueue callback*, if installed, calls the enqueue operation for the respective message, provided by I/O thread. The I/O thread then marshals all essential information into an instance of *MessageIOThread* and enqueues it for future handling. It also uses the message type and specific information to determine whether or not to block the sender until the message has been processed.
- The *claim callback* can be used by device models or handlers to indicate that the message should not be enqueued but rather handled directly by the initiator.

In the object of type *MessageIOThread*, messages are encoded in a structure as illustrated by Listing 4.1.

Listing 4.1: Layout of MessageIOThread structure

```
struct MessageIOThread
{
  /* ... queue-specific element members ... */
  enum Type {
    /* ... event types ... */
  } type;
  enum Mode {
    MODE_NORMAL, // standard LIFO delivery
    MODE_EARLYOUT, // stop delivery after one receiver
    MODE_FIFO, // FIFO delivery
    MODE_RR // round-robin delivery
  } mode;
  enum Sync {
    SYNC_SYNC,
    SYNC_ASYNC
  } sync;
  /* ... additional fields ... */
  void *ptr; // pointer to original message
  void *sem; // pointer to synchronize object (semaphore)

  /* ... constructors ... */
}
```

Messages on the bus are marshalled into such a *MessageIOThread* as a pointer to the original message, together with environmental information, such as the event type, send mode (e.g., FIFO, round-robin, etc.), ability for being sent asynchronously and a pointer to an object providing the synchronization mechanism for synchronous messages. The default *send* function now works according to the pseudocode shown in Listing 4.2.

[1] Not to be confused with *D-Bus*, part of the *freedesktop.org* project.

Listing 4.2: Send function calling enqueue callback

```
template <class M>
class DBus
{
  /* ... */

  /**
   * Send message LIFO asynchronously.
   */
  bool send(M &msg, bool earlyout = false)
  {
    if (iothread callbacks present) {
      /* execute all registered callbacks */
    }
    if (no callback succeeded and enqueue callback present) {
      /* enqueue message */
    }
    /*  ... original send code ... */
  }

}
```

Other send functions (e.g., the FIFO or the round-robin version) are implemented analogously using the appropriate flags. To enable the I/O thread to use the same bus infrastructure for actually sending the requests, I added another function called *send_direct*, which encapsulates the original send functionality, including FIFO and round-robin. The respective callbacks can be configured using two newly added functions of the DBus object: *set_iothread_enqueue* and *add_iothread_callback*. Picking up the example of the ACPI *PM Timer* mentioned in Section 3.4, I will now describe the entire workflow for both the original and the hybrid bypass scenario.

The situation is initiated by the guest operating system issuing an I/O read instruction (*INL*) to read a 32-bit value from the PMTimer port (usually port *0x8000*). This causes a VM exit, passing control to the I/O handler of the VMM. This handler in turn tries to send a message of type *MessageIOIn* on the bus, where the device model would then receive the message and reply to it. Using the I/O thread, the flow would present as demonstrated by an exemplified code section in Listing 4.3. However, if the PM Timer registers a *claim callback* at the bus, the entire enqueue mechanism will be skipped and the vCPU executes the receive handler of the device model itself.

In the asynchronous case, the client is not blocked but rather returns immediately. Usually, the message then gets destroyed, so the I/O thread has to operate on a local copy. Other than that, the process is exactly the same as in the synchronous case. However, one important aspect mentioned in Section 3.2.1 is the ordering of requests. While device accesses are serialized as in the PCI root complex (i.e., the I/O thread guarantees that read requests always read the latest data), access to guest memory can pose a challenge. If for example the write instruction results from instruction emulation, but the subsequent read request is executed natively, the assumption of the guest that the write request has completed is wrong. Consequently, access to guest memory is always excluded from the I/O thread. As there is no need for additional synchronization in these accesses, no special precautions have to be taken.

Another special case has to be handled when a message handler itself sends another message on the bus. Obviously, the I/O thread must not enqueue its own messages. To prevent this, the enqueue function features a simple detection and returns *false*, so the original send code path in *DBus* is used.

Listing 4.3: Accessing the PM Timer through the I/O thread

```
/* vCPU exit handler: */
MessageIOIn msg(MessageIOIn::TYPE_INL, 0x8000);
bus_ioin.send(msg);
/* send method redirects to following enqueue */
bool IOThread::enqueue(MessageIOIn &msg,
                       MessageIOThread::Mode mode,
                       MessageIOThread::Sync sync,
                       unsigned *value) {
  /* ... check for self-enqueueing ... */
  // I/O port reads are always sync
  sync = MessageIOThread::SYNC_SYNC;
  MessageIOThread *enq = new MessageIOThread(
                    MessageIOThread::TYPE_IOIN,
                    mode, sync, value, &msg);
  /* configure for synchronous (semaphore) */
  syncify_message(enq);
  this->enq(enq);
  /* sync_message will block until finished */
  sync_message<MessageIOIn>(enq, &msg, sizeof(msg), sync);

  return true;
}

/* I/O thread: */
MessageIOThread *msg = queue.pop();
MessageIOThread::Sync sync = msg->sync;
MessageIOThread::Type type = msg->type;

switch (type) {
  /* ... */
  case MessageIOThread::TYPE_IOIN:
    {
      MessageIOIn *msg2 =
       reinterpret_cast<MessageIOIn*>(msg->ptr);
      /* access device model */
      bus_ioin.send_direct(*msg2,
                         msg->mode, msg->value);
      /* wake enqueuer */
      sync_msg<MessageIOIn>(msg);
    }
    break;
  /* ... */
}
```

For the correct operation of the I/O thread the specifics of the queue itself are not of importance. During the work on this project, I used both a simple list guarded by a lock and a more sophisticated lock-free queue interchangeably. Obviously, to replace the original global lock by another globally serializing lock is not the most promising approach with respect to performance. However, as the critical section now consists solely of the enqueue and dequeue operation rather than the entire device handling, the performance impact of a lock-free queue implementation is not as noticeable as expected, as can be seen later in Section 5.3.1.

## 4.3 Testing Environment / Stress Testing

As explained in Section 3.4, certain device models shall be configured to bypass the I/O thread. Before this is possible, the affected models have to be made thread-safe by for example the use of fine-grained locking. To simplify this process and to provide for reliable and reproducible results, I decided to use a testing environment described in Section 3.3.1. It consists of a simple, stand-alone test program which is linked against the device model or subsystem under test. By implementing all receive handlers for messages sent from within the device and attaching to the motherboard provided by the program and passed to the device model, the program can effectively act as the VMM around it without having to incorporate the entire complexity. Furthermore, the synthetic test does not necessarily have to re-implement the complete functionality of components attached to the device, but only has to mimic the basic behavior it expects on its message passing API.

Listing 4.4 shows example code from the program used to test the PIC model, simulating highly concurrent load on two of its interrupt pins. Instances of the `trigger_fn` function send a fixed number of IRQ messages to the PIC as fast as possible, while the `receiver_fn` mimics a virtual CPU receiving the resulting interrupt signals. If not all of the IRQs sent by the trigger functions are received by the emulated vCPU, a race condition in the PIC led to inconsistencies and needs to be avoided, as described later in Section 4.4.1. The log created by the `logger.log` calls can be used to determine the course of events that caused the problem and help identify the critical section.

Using this environment, I was able to detect critical sections in the vCPU and interrupt subsystem, install the respective synchronization primitives and test their correctness in a systematic and incremental way. In the following section, I will describe in detail what race conditions occurred and what changes were necessary to solve them.

Listing 4.4: Example code of the synthetic PIC test program

```
template <unsigned char IRQ>
static void * trigger_fn(void *) {
  logger.log(LOG_INIT);
  /* ... */
  while (sent < IRQ_COUNT) {
    /* ... wait for re-raise condition ... */
    logger.log(LOG_SEND, IRQS[IRQ-1]);
    mb.bus_irqlines.send(msg);
    sent++;
  }
  return nullptr;
}

static void * receiver_fn(void *) {
  while (true) {
    if (!__sync_fetch_and_and(&intr, 0)) {
      /* ... exit condition ... */
      continue;
    }

    logger.log(LOG_INTA_TX, check.value);
    MessageLegacy inta(MessageLegacy::INTA, 0);
    waitcount = 0;
    mb.bus_legacy.send(inta);
    logger.log(LOG_INTA_RX, inta.value);

    if (inta.value == IRQS[0]) irq_received_1++;
    if (inta.value == IRQS[1]) irq_received_2++;

    eoi(inta.value);
    logger.log(LOG_EOI, inta.value);
  }
}

/* message handlers */
static bool receive(Device *, MessageLegacy &msg) {
  /* ... set or clear virtual INTR pin ... */
}
static bool receive(Device *, MessageIrqNotify &msg) {
  logger.log(LOG_NOTIFY, msg.baseirq << 8 | msg.mask);
  /* ... set re-raise condition for IRQ ... */
}
```

## 4.4 I/O Handling in Initiator Context

With the hybrid I/O thread mechanism in place, certain devices can be configured to bypass the queue and be handled by the vCPU or service thread itself. As a consequence, those devices have to be thread-safe.

A prominent example to illustrate the fundamental difference between device models and real hardware mentioned in 2.1.4 is an interrupt controller like the *PIC* or the *Local APIC*. Firstly, the device model implements the prioritizing in software while the real chip features hard-wired pins that are asserted immediately. Secondly, and this in fact extends the first point, it receives the signal for the next timer interrupt to be delivered not from a piece of hardware (e.g., an oscillator or an external device) asserting a PIN, but through a timer notification from the host system. This message is typically received and handled by a dedicated timer thread of the VMM. Now if for example the guest tries to reconfigure the PIC and at the same time a virtual timer interrupt has to be asserted, the two threads execute code in the PIC model concurrently, which can lead to race conditions. In hardware, asserting, prioritizing and delivering happens atomically within one clock cycle. Conversely, the software emulation can take several hundred cycles for the same event, which is why synchronization is necessary to avoid inconsistencies in the device state.

Because situations like the one mentioned above can happen in a virtual environment, it is only possible to remove the global lock or the protection provided by the I/O thread when at the same time any of such dependencies are identified and all guarantees retained.

In general, there are two alternatives to protect affected sections: Fine-grained locking of the critical sections directly in the device models, or performing state updates using atomic operations (e.g., atomic and/or, compare and exchange). Conceptually, an atomic operation is an update of a memory location where the hardware guarantees that the result is the same as when the update were protected by a lock. These operations are basically the finest-grained locking mechanism available. They allow for the shortest critical sections and thereby bear the highest potential for parallelism, but are often more complicated to use to protect larger objects like a device state. The more complicated the entity to be protected gets, the more difficult or even inefficient it gets to use atomic operations. In some cases they are even not sufficient at all.

In the following subsections I will present synchronization issues using the example of two device models, the approaches I used to solve them and explain why and how I implemented them.

### 4.4.1 Example: Race Conditions in the PIC Model and Their Solutions

The virtual PIC is a simple device model. It basically consists of three eight bit wide registers: The Interrupt Request Register (IRR), Interrupt Service Register (ISR) and Interrupt Mask Register (IMR). Furthermore it implements virtual IRQ pins for eight interrupts (vectors 0-7), prioritizing logic and a notification mechanism to signal an IRQ to the vCPU. To simulate level triggered interrupts, the model uses one more eight bit wide register to notify other device models that an IRQ has been acknowledged and can be raised again. A notification is sent if the respective bit is set in the notify field and cleared in the IRR. The rest of the functionality (special mode settings, cascading) is not covered in this section. A detailed description of the PIC can be found in the specification [7].

When a thread (e.g., the timer thread or another device interrupt) signals an IRQ, it sets the bit in the IRR corresponding to the interrupt vector and runs the prioritizing algorithm. The vector with the highest priority whose bit in the IMR is not set and where no higher priority interrupt is currently being served is the result of this computation. If there is such a vector, the thread notifies the vCPU that there is an interrupt pending. The vCPU then issues a so called *INT ACK* cycle, where the respective bit is transferred from the IRR to the ISR and the vector is returned. After handling the interrupt, the guest will issue an *End Of Interrupt* (EOI), which clears the bit in the ISR and in turn runs the prioritization and notifies the vCPU whether or not there is another interrupt pending.

As already explained in Section 4.4, the virtual PIC is subject to concurrent accesses from the signalling thread and the virtual CPU thread. One possible situation is that the guest performs an EOI while another thread delivers the signal in form of an IRQ. Both actions trigger a prioritize cycle and perform state updates that are not atomic. The interleaving execution of these two operations cause two race conditions which I had to prevent as described in the following sections.

| Trigger Thread | Receiver Thread |
|---|---|
| Set IRR | |
| Prioritize (result: INTR) | |
| Send INTR | |
| | INT ACK |
| | Clear INTR |
| | Send EOI: Clear ISR |
| Set IRR | |
| Prioritize (result: INTR) | |
| | Send EOI: Prioritize (result: INTR) |
| | Send INTR |
| | INT ACK |
| | Clear INTR |
| | Send EOI: Clear ISR |
| | Send EOI: Prioritize (result: no IRQ) |
| Send INTR | |
| | INT ACK (result: spurious interrupt) |

Table 4.1: Sequence of events leading to a spurious interrupt in the PIC

#### 4.4.1.1 Delayed INTR and DEASS Messages

The first issue can be reduced to the simple fact that between prioritizing and performing the respective state update can be an arbitrary delay due to preemption. Both signalling an IRQ and performing an EOI trigger the prioritize algorithm and in case an IRQ is pending send an INTR message to the vCPU. Now if the interleaved accesses happen in a particular sequence as listed in Table 4.1, the vCPU receives an INTR message when no interrupt is pending anymore. This occurs when the guest performs an EOI exactly at the time where the signalling thread has calculated the highest priority vector and would send an INTR message to the vCPU next. But before this message, the vCPU thread completes the EOI, recognizes the pending interrupt itself, and handles it completely (INT ACK and EOI). The outstanding INTR message is then obsolete and leads to a so-called spurious interrupt in the PIC. That is, an interrupt was signalled to the vCPU, but the INT ACK cycle does not return a valid vector. A very similar course of events can lead to an outdated message clearing the INTR state (DEASS).

The simplest solution to avoid this problem is to modify the public interface of the PIC model such that a lock serializes all PIC accesses. This ensures that prioritizing and notifying are done atomically. However, the parallelism would suffer because for example a vCPU thread would have to wait for other threads to release the lock until it can perform an EOI. Furthermore, many operations within the PIC are already atomic or could be very easily modified. For instance, the state updates to the IRR and ISR can be done using atomic *AND* and *OR* operations. The prioritizing function can also be used concurrently, only the state updates to the vCPU have to be taken care of.

Analysis of this problem showed that there is no way to avoid the described race condition other than using a coarse-grained lock in the PIC preserving the original concept. The only other solution is to modify the vCPU such that it can cope with spurious interrupts and implement a so called *pull model*. Effectively, the INTR message of the PIC only acts as hint that the PIC needs a state update. When such a message is received, a sanity check is performed by prioritizing in the context of the vCPU. Only if this sanity check does not yield a valid interrupt, the INTR state is cleared. This also solves the problem that outdated DEASS messages would cause lost interrupts.

Although this creates additional overhead (the prioritize function is called twice every time), it allows for a high degree of parallelism within the device model. Because the frequency of such spurious interrupts is low (a kernel compile on eight virtual CPUs showed an overhead of 0.03 %, which lies in the range of measuring inaccuracies), the additional overhead caused by unnecessary recalls can be neglected.

#### 4.4.1.2 Notify Mechanism

Another issue regarding concurrent state updates in the PIC was the notify mechanism used for level-triggered interrupts, e.g. for PCI devices. Another use case is to relieve interrupt load on devices like the *PIT*. By programming a new timeout only when the previous interrupt was seen by the guest, a timer device can be effectively limited in the interrupt load it can generate to match the handling speed of the guest.

However, because the notification is sent from within the prioritize function and both the IRR and the notify field have to be evaluated to determine if there are notifications pending, messages could be sent incorrectly or multiple times. If the triggering thread updates the notify field and afterwards the IRR, a concurrent prioritize could already send the message and clear the notify flag. The notification would be incorrect and could cause an immediate new interrupt request to be dropped and the signaller would never be notified again.

Reversing the order can avoid incorrect notifications, but the message could still be sent multiple times. Basically, the accesses to the IRR and the notify field have to be atomic. The notifier has to check which bits are different in the two values and clear those bits in the notify field. Because atomic operations only work on single memory locations, they are not sufficient here. I had to restructure the mechanism such that instead of sending notifications in every prioritize call it specifically notifies when an *EOI* command is being handled. This way, the bit in the IRR is already known and can be atomically cleared in the notify field. By checking the value of the field before the update, which is returned by the atomic operation, the notification can be sent if necessary.

The only other way would be to consolidate the four eight bit values into one 32 bit wide register that can be updated atomically. But since the notification then becomes a read-modify-write operation, the logic to implement the desired behavior would be complicated and not wait-free. That is, it would have to use *compare and exchange* in a loop until it performed a consistent update. Therefore I decided to use the simpler wait-free solution, although it possibly sends more notifications. The potential overhead is acceptable because these notifications are only used for legacy components (i.e., the *PIT* and legacy PCI interrupts, which are increasingly superseded by *MSI* and *MSI-X*). Both the *PIT* and legacy level-triggered interrupts are not concerned with respect to multiprocessor scalability and therefore I took no special precautions in this regard.

### 4.4.2 Local APIC

The existing implementation of the *Local APIC* device model was susceptible to corruption through concurrent access to the timer functionality, if not protected by the global lock. Triggering timer interrupts was integrated into the *get_ccr* function responsible for providing the caller with the virtualized *Current Count Register (CCR)* value. The rationale behind this decision was to trigger interrupts as early as possible: When an interrupt is found to be due while calculating the current count value, it is raised directly rather than waiting for the host service to notify the device model.

However, this code path is hence being accessed by both the host timer service and the virtual CPU, which can potentially happen concurrently. As atomic operations did not suffice in this case and a locking mechanism within the device is more complex and expected to lead to performance degradation, I chose to modify the mechanism such that virtual timer interrupts are triggered only by the host service and *get_ccr* is called only by guest code execution.

The only disadvantage of this approach is that virtualization artifacts may become visible. Because virtual timer interrupts are always delivered with a slight delay caused by the host service notification, guest code could read a counter value which indicates an elapsed timeout while the actual interrupt has not yet arrived. However, this is not explicitly precluded in the specification ("After the timer reaches zero, an timer interrupt is generated [17, p. 10-17]"). Furthermore, guest software could discover such artifacts already because time virtualization is a complex topic [32].

Because the local APIC could be assisted by hardware extensions in the future, this issue is ignored in my implementation. Although hardware-assisted APIC virtualization could reduce overhead and complexity, it is not considered mainly because of the need for support in the microhypervisor. The focus of this project was to enable efficient SMP support in the user-level VMM without having to modify the underlying microkernel solution.

## 4.5 Summary

In this chapter, I described how the implementation of virtual SMP support in Vancouver was modified to support multiple physical cores. After illustrating the I/O thread message queueing mechanism, I presented the testing environment used for identifying and eliminating race conditions in unprotected device emulation code. By resolving the specific issues as described in Section 4.4, I showed how the hybrid approach could be enabled for the vCPU and interrupt controller subsystem. In the next chapter, I will evaluate the different stages of the SMP implementation with respect to scalability and overall performance.

# 5 Evaluation

Assessing multicore performance is usually done using tailored benchmarks revealing scalability achieved by distributing work among CPUs. The nature of these benchmarks ranges from pure arithmetic computation to more realistic workloads like compile jobs or even I/O-intensive software using network or disk.

In virtualized systems, the impact that is most noticeable is the handling of VM exits. Workloads that primarily do calculation without causing considerable amounts of virtualization overhead will not be able to reveal possible scalability inhibitors. As number crunching is thus not very challenging from VMM perspective, I focused my evaluation on workloads involving a higher number of VM exits in different varieties. One prominent scalability benchmark in this area is to compile a Linux kernel in a ramdisk, because it utilizes SMP very well while still causing a relevant amount of exit handling due to interrupts and timer programming. It is expected that this workload will reveal performance issues in the vCPU and interrupt device models and will be a perfect match to evaluate the hybrid I/O thread approach.

Because compiling a kernel is still considered to be a CPU-intensive workload, I also conducted I/O-intensive experiments for examining network and disk performance. I used the widely established network benchmark *netperf* [6] to measure inter-VM network throughput and ran kernel compile benchmarks on a virtual disk for I/O device model involvement. As opposed to the scalability focus of the ramdisk kernel build, this class of experiments has an intrinsically serializing I/O component and therefore targets the overall synchronization overhead of the virtual SMP solutions, which is not expected to vary significantly with the number of vCPUs.

For evaluation I used three machines listed in Table 5.1 on the following page, serving separate purposes. *System A* is a server that provides a high number of cores and therefore suits the kernel compile benchmark perfectly. On the other hand, *System B* is a commodity desktop computer with the possibility to use peripheral I/O devices (e.g., disk, network) for measuring the overall overhead and I/O performance. For heavy I/O utilization, *System C* features a 10 GBit NIC and a relatively high number of cores, which can be used to assess parallel I/O performance. To create load on the test machines for performing the I/O benchmarks, I used two load generators as shown in Table 5.1 on the next page.

| | **System A** | **System B** | **System C** |
|---|---|---|---|
| *CPU name* | Xeon (Nehalem) | Xeon (Haswell) | Xeon (Westmere EP) |
| *CPU frequency* | 2.261 GHz | 3.5 GHz | 2.67 GHz |
| *Number of cores* | 32 | 4 | 12 |
| *Number of sockets* | 4 | 1 | 2 |
| *Hyper-Threading* | yes | yes | inactive |
| *RAM* | 128 GiB | 16 GiB | 32 GiB |
| *Peripheral devices* | none | SATA SSD<br>Intel I210 1 GBit | Intel 82599EB 10 GBit |

Table 5.1: Test systems technical specification

| | **System L1** | **System L2** |
|---|---|---|
| *CPU name* | Core i5 (IvyBridge) | Core i7 (Nehalem) |
| *CPU frequency* | 2.8 GHz | 2.8 GHz |
| *Number of cores* | 2 | 4 |
| *Number of sockets* | 1 | 1 |
| *Hyper-Threading* | yes | inactive |
| *RAM* | 4 GiB | 2 GiB |
| *Peripheral devices* | Intel 82579LM 1 GBit | Intel 82599EB 10 GBit |

Table 5.2: Load generators technical specification

## 5.1 Benchmarks Comparing I/O Thread and Fine-Grained Locking

To evaluate the improvement of the approaches implemented during this thesis to the existing solution and native performance, I ran a Linux kernel compile in both a ramdisk and a virtual SATA drive as well as inter-VM network throughput measurements using *netperf*. The solutions involved (global lock, fine-grained locking, full and hybrid I/O thread) are compared with respect to scalability and overall overhead. Qualitatively, fine-grained locking is expected to be the fastest, while the global lock will have severe performance deficits. The full I/O thread would rank between both, leaving the hybrid approach to match the fine-grained one as close as possible. The overall goal is to achieve near-native performance with the most elegant solution.

### 5.1.1 Kernel Compile in a Ramdisk

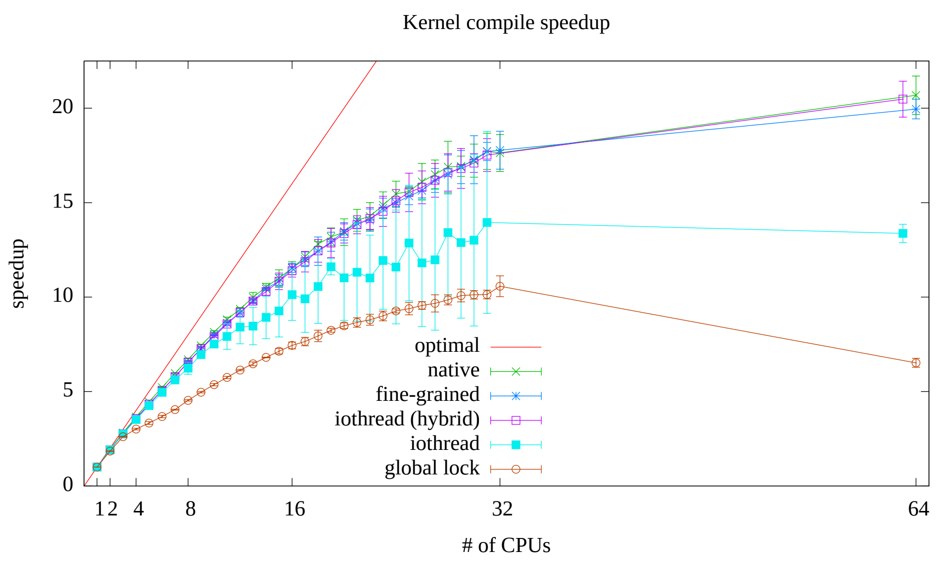


Figure 5.1: Speedup comparison of the different synchronization mechanisms

The experiment was undertaken on *System A*, in a VM with up to 32 (31 when using an I/O thread) cores and 2 GiB of RAM. An additional data point was added by activating *Hyper-Threading* [23] and thereby allowing for 64 and 63 vCPUs, respectively. Figure 5.1 shows the speedup achieved by the given number of cores (error bars indicate the minimum and maximum, respectively). Matching the expectation, the fine-grained synchronization achieves near-native performance throughout the range of available vCPUs, while the global lock significantly impedes scalability. The anticipated impact of a full I/O thread implementation can be clearly seen in the performance gap to the other solutions. Interestingly, the variability of results is considerably higher for the full I/O thread, which is caused by varying contention on the queue, leading to for instance delayed *inter-processor interrupts*

*(IPI)*. The results also show that the hybrid version where the vCPU and interrupt controller subsystem is bypassing the I/O thread can keep up with the near-native speedups.

### 5.1.2 Workloads Involving I/O

In order to prove the holistic claim of the I/O thread solution, I also measured I/O-intensive workloads to show performance of virtual disk and network device models superior to what can be achieved by a global lock. These experiments were conducted on *System B* with activated Hyper-Threading, providing for up to eight virtual CPUs. Figure 5.2 shows the network throughput of two VMs running on the same host, using a virtual 82576vf network device and a virtual network bridge to connect them. Again, the minimum and maximum throughput is given by the error bars. The results of both a *TCP STREAM* and *TCP MAERTS* test provided by *netperf* are compared for the I/O thread and the global lock.

The virtual SATA drive was tested by running a Linux kernel compile on eight virtual CPUs, but instead of using a ramdisk, the SATA model connected to a virtual disk provided by the runtime environment. Although this still leaves the data in RAM, the disk service and the respective device emulation within the VMM are involved. Performance issues caused by the synchronization mechanism to protect critical paths could by easily identified. Figure 5.3 compares the compile times of the three different solutions mentioned earlier. Note that this graph includes results of seven and eight virtual CPUs for the global lock and KVM. This is due to the fact that a VM with seven vCPUs and a dedicated I/O thread does not exactly compare to either seven or eight cores synchronized by a global lock. The dedicated I/O core cannot be regarded as a full computation core, because it can be shared between the I/O threads of multiple VMs, as done for the network benchmark. Neither can it be completely ignored, because a small portion of the work of a VM is done in parallel on this core. To remain as fair as possible, both results are shown without clearly defining one as the direct competitor of the I/O thread.

## 5.2 The Perfect Device Model

The I/O experiments evaluated so far did not incorporate heavily parallel I/O. Access to the SATA drive is inherently serial and the current version of the virtual network bridge is serialized by a global semaphore. To evaluate parallel I/O performance, I included another setup where the virtual machine is stressed by parallel requests to a network adapter featuring multiple send/receive queues. The VM was executed on *System B* and tested by load generator *L1*. To measure the scalability characteristics of a concurrently accessed network device model I decided to conduct a modeled experiment using pass-through NICs. By forcing the VMM to emulate all *Memory-mapped I/O* (MMIO) accesses to device memory, the setup acts like a perfect device model where device emulation code does not consume CPU time. As virtual interrupts and device accesses are still being handled by the VMM, the results can be used to qualitatively predict the performance of the respective synchronization solutions. Network scalability is measured by running an increasing number of parallel instances of *netperf*'s *TCP_RR* test. Figure 5.4 shows the aggregated number of transactions

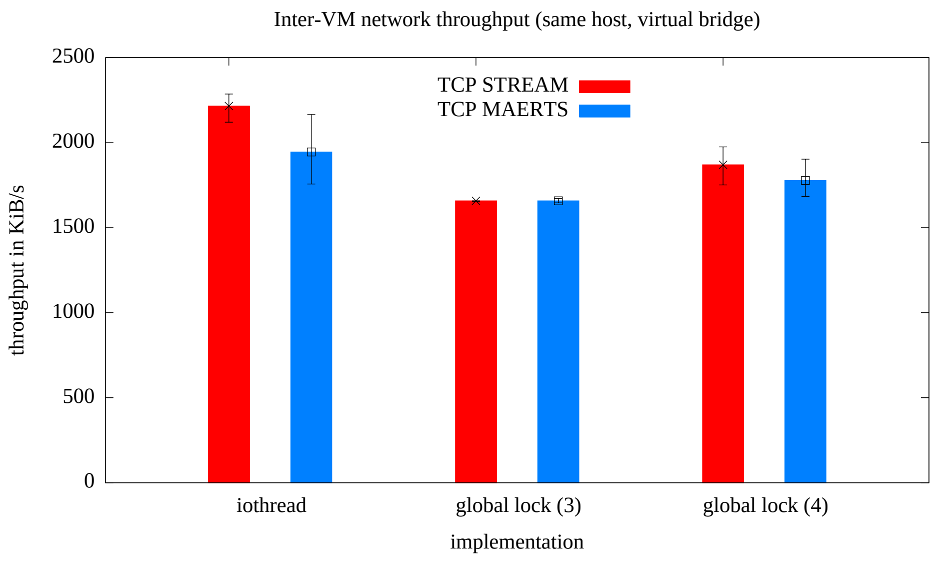


Figure 5.2: Network throughput comparison

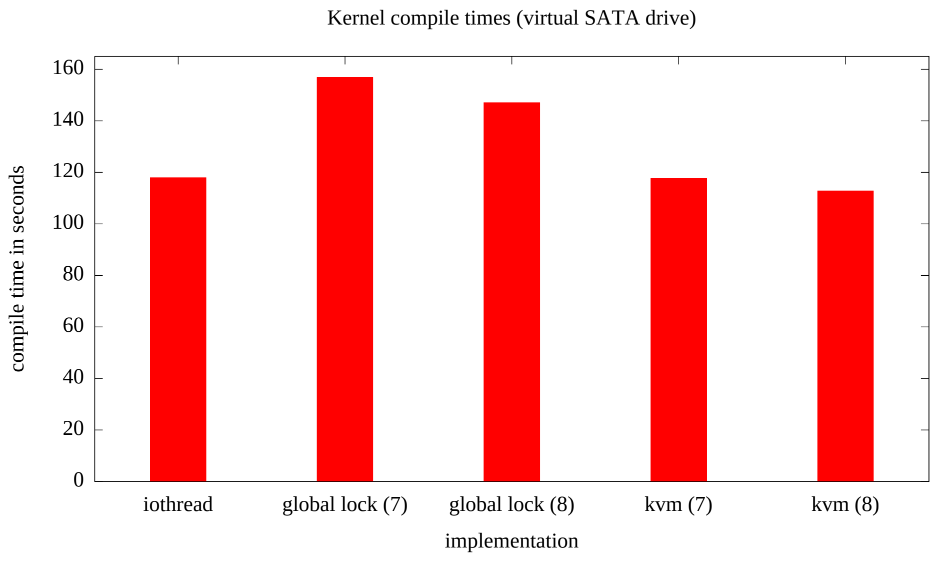


Figure 5.3: Kernel compile time in a virtual SATA drive

per second achieved by the different solutions. The maximum achieved by native Linux is depending on the load that *L1* can generate.

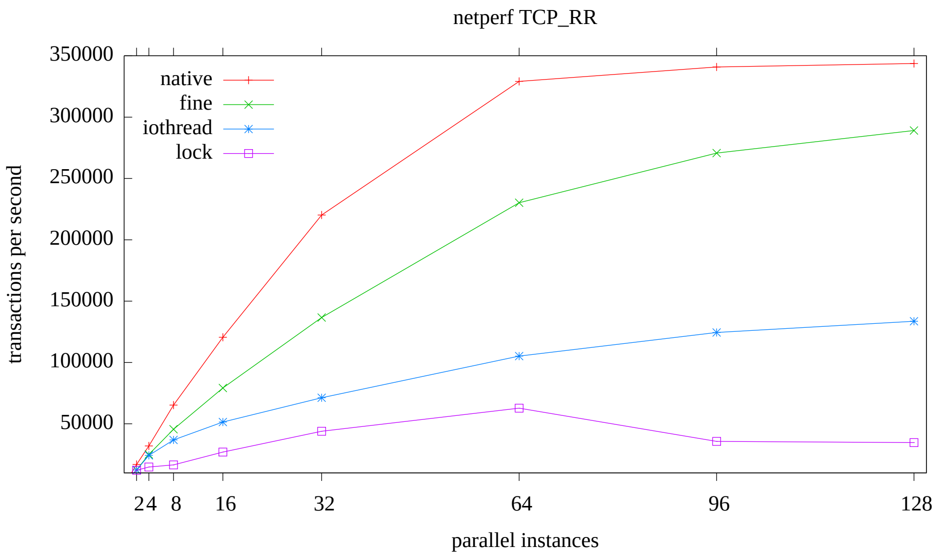


Figure 5.4: Scalability test of multi-queue networking

Because this experiment revealed a major disadvantage of the I/O thread, I decided to run another benchmark involving parallel I/O. While throughput measurements on GBit networks are not meaningful (achieving wire-speed is not an issue), with 10 GBit it becomes interesting again. To ensure that the machines used for testing are not the bottleneck, I used a third system (*System C*) with more cores and a more powerful load generator *L2*. Both were equipped with a 10 GBit network adapter and the throughput was measured by *iperf*. Figure 5.5 shows the compared throughput achieved by a varied number of parallel connections. It can be seen that only native Linux and the fine-grained approach are able to achieve and keep up wire-speed throughput by using two or more connections. Although the hybrid I/O thread performs better than the global lock due to the bypassing parts of the system, it still can not provide the available throughput.

## 5.3 I/O Thread Statistics

As pointed out in Section 3.4, the ratio of synchronous to asynchronous messages is an essential key number for the performance of the I/O thread. To illustrate this, I extended the example calculation for 16 CPUs to a basic scalability model of the I/O thread, producing a projected speedup graph for any number of CPUs running a given workload. Equation (5.1) on the facing page shows the formula for an estimated speedup with a given parallel fraction function $P_{I/O\ thread}(N)$, which determines the parallel fraction according to the formula shown in Figure 3.7. When the parallel fraction was increased by configuring certain subsystems for bypassing, Equation (5.2) on the next page can be used with the new $P_{hybrid}$.

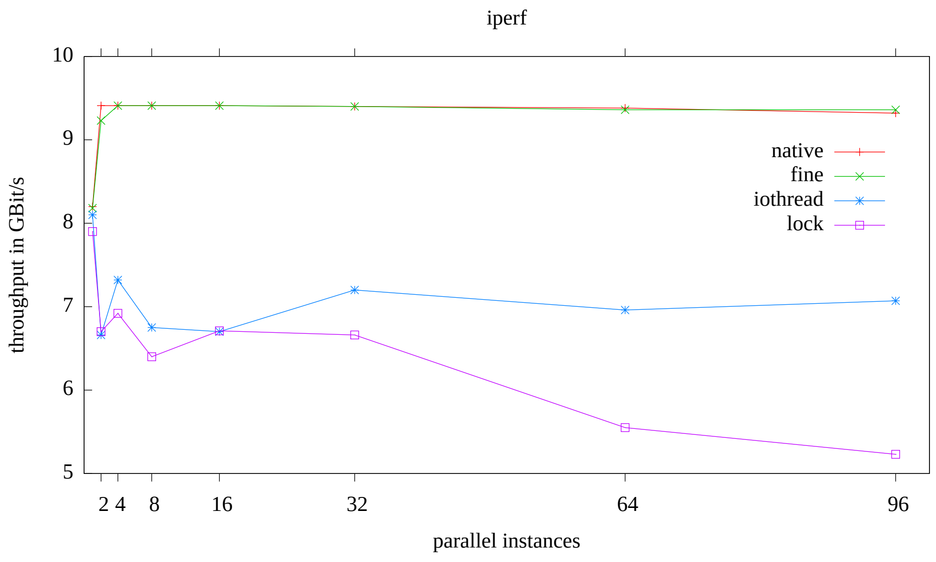


Figure 5.5: Throughput comparison of *iperf* with 10 GBit NIC

Using the kernel compile example from Section 5.1.1, I could project an estimate of the performance gain of the hybrid approach before even implementing it. Figure 5.6 shows the result of this projection for the given workload characteristics of a kernel compile. Although the simple nature of the model expectedly shows inaccuracies in the full I/O thread speedup calculation, it clearly indicates an estimated speedup trend that can be used to assess the effect of planned improvements. It therefore proved to be useful in the decision to implement the fine-grained locking part in the vCPU and interrupt subsystem and to integrate it into the hybrid I/O thread approach.

$$S_{I/O\ thread}(N) = \frac{1}{(1 - P_{I/O\ thread}(N)) + \frac{P_{I/O\ thread}(N)}{N}} \tag{5.1}$$

$$S_{hybrid}(N) = \frac{1}{(1 - P_{hybrid}(N)) + \frac{P_{hybrid}(N)}{N}} \tag{5.2}$$

### 5.3.1 Comparison of Queue Implementations

Although I expected the queue characteristics to be a key aspect of the achievable performance with an I/O thread, the *TCP_RR* experiment revealed only a slight and unstable advantage of the lock-free implementation over the locked queue as shown in Table 5.3. The low difference and the outliers where the lock-free queue performed slightly worse suggest that the two implementations form no difference with respect to scalability. During the experiments, it became clear that most of the performance impact comes from the serializing behavior itself and the queue operations are almost negligible. In order to achieve

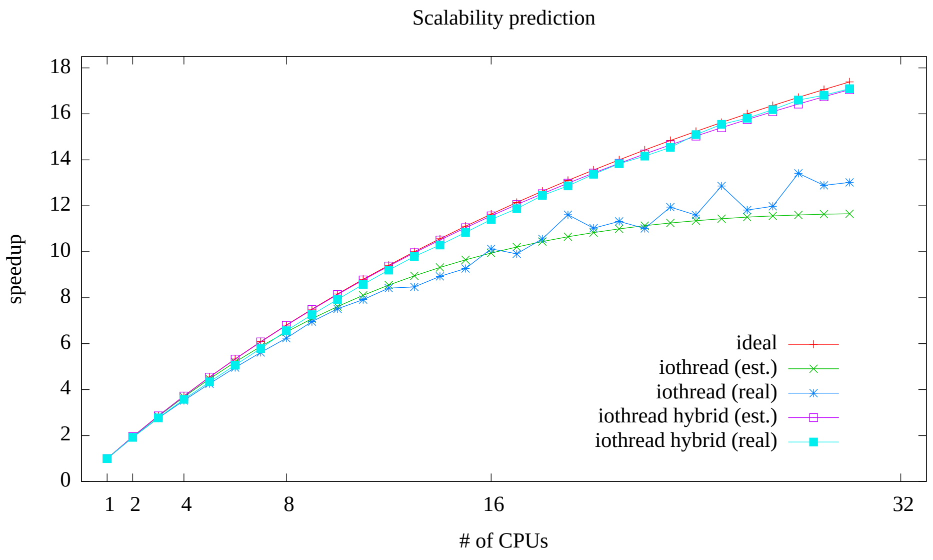


Figure 5.6: Estimated vs. real speedup of full and hybrid I/O thread

maximum performance, highly concurrent accesses should be prevented from being queued in the first place by means of the hybrid approach.

## 5.4 Performance of Multiple Virtual Machines

Although not specifically part of this thesis project, running multiple virtual machines on the same host is an interesting use case that is to be considered. Especially for a fair evaluation of the I/O thread, it is necessary to show if dedicated I/O cores can be shared among VMs and if there is a performance impact. During my network experiments I discovered that a shared I/O core significantly improves inter-VM network throughput as shown in

| instances | tps (locked) | tps (lock-free) | difference (percent) |
|---|---|---|---|
| 1 | 6216.51 | 6322.41 | +1.70 |
| 2 | 12271 | 12391.3 | +0.98 |
| 4 | 23388.6 | 24288.1 | +3.85 |
| 8 | 35052.9 | 36915.1 | +5.31 |
| 16 | 49650.1 | 51496.2 | +3.72 |
| 32 | 71972.1 | 71272.1 | −0.98 |
| 64 | 106370 | 105234 | −1.07 |
| 96 | 123803 | 124459 | +0.53 |
| 128 | 133878 | 133559 | −0.24 |

Table 5.3: TCP_RR performance comparison lock-free vs. locked queue

Figure 5.7, most likely because of caching effects. On the other hand, unrelated VMs may suffer slowdown effects when the CPU executing the I/O threads becomes fully utilized.

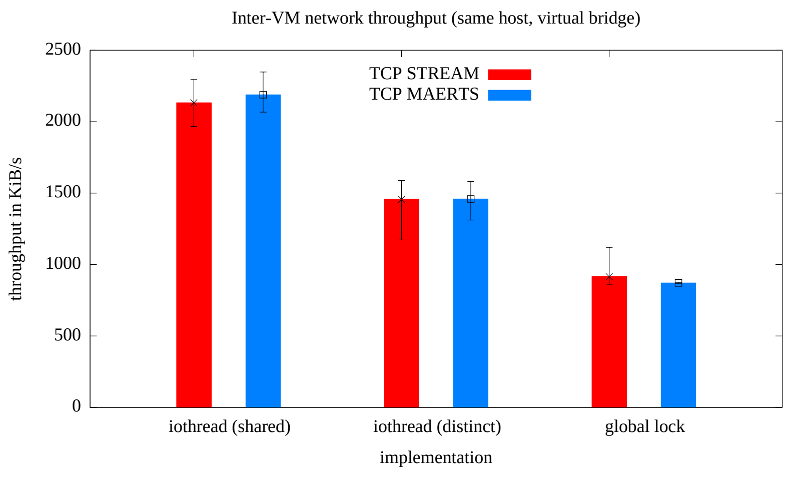


Figure 5.7: Inter-VM network throughput comparison for shared and distinct I/O cores

Another aspect of multiple VMs is physical CPU sharing between vCPUs. If multiple virtual cores are co-located on one physical core, scheduling becomes very challenging and is still being actively researched, as mentioned in Section 2.2. The effect of one underlying problem commonly referred to as *lock-holder preemption* can be seen in a simple experiment: A Linux kernel compile is executed in two VMs simultaneously. In the first version, both VMs get three exclusive physical cores to execute on, in the second they share the same six ones. From a pure mathematical point of view, the results should be the same, but in practice the second version shows ~4-5% overhead.

Because the time frame of this thesis did not allow for further investigation and optimization of these two aspects, they are just mentioned here for completeness, but could be considered in future work on virtual SMP in NOVA.

## 5.5 Summary

The conducted experiments were able to confirm the expected performance issues when scaling the global lock or the full I/O thread approach to a higher number of cores. Although the I/O thread performs better than the global lock, the serializing characteristics still impede maximum scalability. As predicted using a simple scalability model based on *Amdahl's Law*, the hybrid approach was able to significantly improve performance. However, the modeled experiment using a pass-through NIC clearly showed that even though posted writes help to a certain degree, highly parallel accesses to device models can not be handled by a serializing I/O thread without losing scalability.

# 6 Future Work

While working on this project, I identified several opportunities to improve virtual SMP in the NOVA virtualization architecture. The reason for why I did not incorporate them in this thesis is the limited scope with regard to both time and software components. As I focused on the SMP infrastructure within the user-level VMM, optimization involving hypervisor modifications were not feasible.

In the following sections I will explain ideas which could be worked on in future projects in order to improve certain aspects of multiprocessor support for virtual machines. Both scalability and real-time aspects are covered.

## 6.1 Improving Synchronization of Virtual CPUs

The first vector of improvement is the synchronization of virtual CPU handling itself. In Section 5.3, the performance model clearly indicated that the interrupt infrastructure is one of the major scalability inhibitors when synchronized inefficiently. But also parallel I/O has to be handled with as little serialization as possible to achieve maximum throughput, as shown in Section 5.1.2 on the example of 10 GBit network. These are therefore very promising components to inspect for optimization.

The local APIC together with the vCPU model is a central part of a multiprocessor system due to frequent and concurrent access to its functionality. As already mentioned in Section 4.4.2, there exist hardware mechanisms to reduce the number of VM exits caused by APIC accesses. To leverage them already promised significant improvement in other solutions like KVM [24]. Implementing support in both the hypervisor and the VMM could be advantageous not only for performance, but also for code complexity, because part of the APIC emulation and its synchronization could be dismissed.

Another interesting field of research is *Transactional Memory* [16], also arriving as hardware extensions with recent processor generations. By grouping critical code sections into transactions, hardware would execute them speculatively and only apply synchronization in case of a conflict. A variety of software and hardware flavors have been researched [20], but it has to be evaluated how the inherent overhead compares to conventional locking schemes. From a programming point of view, code would become much more readable and less error-prone, as critical sections only have to be identified and placed in a transaction rather than spreading locks among conflicting components and ensuring correctness and deadlock-freedom in the design.

With regard to the I/O thread, future projects could investigate a stronger focus on the hybrid approach with more fine-grained locking in performance-critical device models. Especially parallel networking suffers from throughput degradation due to the serializing property of the I/O thread as shown in Figure 5.5. Using the scalability model described in Figure 3.7, workloads adhering to *Amdahl's Law* could be investigated with respect to performance gain in the hybrid approach.

However, not only parallelism, but also congestion can be a problem with the I/O thread. When the dedicated I/O core is fully utilized, serious delay and slowdown could be the result. Implementing a fall-back to the global lock could help in this kind of situation, if the I/O core is being shared with threads of other VMs or the runtime environment. Although it would re-introduce the previous serialization behavior, it could avoid vCPUs waiting for work unrelated to the VM. If it were possible to identify completely isolated subsystems in the device emulation, it would be an interesting concept to distribute work among multiple I/O threads. If no such subsystems exist, these threads have to be synchronized again, e.g. by the use of fine-grained locking or transactional memory.

## 6.2 Scheduling Virtual CPUs of Multiple VMs

In the previous section, I already mentioned sharing of cores among different components like the VMM and services of the runtime environment. But in a virtualization system used for consolidating multiple machines into one large server, even a larger number of VMs running concurrently is possible. Ideally, virtual machines would share physical cores in such a way that both overall and individual performance are maximized. As explained in Section 2.2.3, predominantly preempting lock holders poses a challenge in doing so.

There exist two techniques tackling the aforementioned problem:

- **Co-Scheduling**[1] [25]: All vCPUs of a VM are only scheduled together.
- **Pause-Loop Exiting**: vCPUs waiting for a lock to be released get preempted in favor of a different VM.

Both ideas have their advantages and disadvantages as well as different target usages. Co-scheduling can be used to establish guaranteed execution time rather than the current best-effort implementation. However, with VMs executing strongly asymmetric workloads (i.e., not utilizing all of the vCPUs evenly), the achievable global utilization is not optimal. On the other side of the spectrum, the system can be configured to schedule virtual CPUs faithfully and taking care of the preemption problem directly. Pause-loop exiting aims at detecting threads executing busy-waiting loops and preempting them, leaving the core free for useful computation. This approach requires a higher level of scheduling intelligence and is not suitable for reasoning about guarantees, but achieves a potentially higher global utilization.

[1] Also commonly referred to as *Gang-Scheduling*.

Another aspect of scheduling in a consolidated system is the currently fixed assignment of virtual to physical CPUs. In order to install intelligent mechanisms to improve performance of utilization, it might become necessary to implement a dynamic assignment supporting vCPU migration. This can be particularly useful in *Non-Uniform Memory Access* (NUMA) environments where access latencies to main memory are not uniform across all CPU sockets [33]. Virtual CPUs could then be migrated closer to the data they are using.

Lastly, the number of vCPUs per VM could be made dynamically reconfigurable to adapt to the current global state of the system. The technique to enable this is called *CPU hot plugging* and could be useful for performance, energy and even migration aspects, when virtual multi-core machines could be migrated between machines with a differing amount of physical CPUs.

# 7 Conclusion And Outlook

This thesis described the journey from a proof-of-concept implementation of virtual multiprocessor support towards a scalable, efficient and yet maintainable solution. In the implemented software components, I took into account that certain virtualized devices have different underlying mechanisms than hardware and therefore need special treatment in the case of parallel access, as described in Section 2.1.3. Furthermore, I incorporated the trade-offs and challenges when implementing concurrently accessed device models. As explained in Section 4.4, race conditions are possible in device models. To prevent them while maintaining the best possible performance requires a deep understanding of the behavior of real hardware and careful protection against data corruption.

With the focus being set to scalability and maintainability and the scope strictly limited to the user-level VMM, the result of this project provides a largely self-contained component that enables safe access to device models. Assessment and following evaluation of the impact of serializing parallel accesses clearly showed the need for sophisticated synchronization mechanisms in highly contended components in order to achieve best performance, which is why mechanisms for directed customization of critical subsystems were integrated. Using the example of the vCPU and interrupt controller subsystem, I implemented the hybrid approach involving minimal serialization. Combining the encapsulating properties of the I/O thread and the performance gain of fine-grained locking, the hybrid I/O thread is the solution providing the highest scalability while only modifying device models that are performance-critical. Future extensions to the VMM (e.g., other device models) can be easily integrated into the structure and be configured to bypass the I/O thread on demand, when thread safety is provided by the device model itself. Compared to the full I/O thread with serious performance issues and a pure fine-grained locking scheme where the entire codebase has to be reviewed and potentially modified, it proved to be the superior solution.

To assess future opportunities, I also investigated use cases and ideas beyond the current state of the software environment (i.e., the NOVA microhypervisor and its runtime environment). By conducting a modeled experiment with a high-speed network connection, I was able to show that also in the case of the virtual network, fine-grained locking will be a promising technique to provide fast, parallel network functionality. To mitigate performance degradation due to virtualization overhead, also the use of recent hardware extensions could relieve the load on currently highly contended VMM code paths.

The software solution implemented in this project can be an enabler for researching a large variety of aspects concerning virtual SMP systems. In Chapter 6, I laid out opportunities for both improving and leveraging the enhancements of my contribution. NOVA could evolve into a flexible solution with the ability to dynamically distribute compute power among virtual machines beyond the scope of single-processor VMs. Tackling the scheduling challenges described in Section 2.2.3 could allow for highly utilized, secure and robust server architectures powered by microkernels, virtualization, hot-plugging and migration [9].

# Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[2] Advanced Micro Devices, Inc. Amd64 virtualization codenamed “pacifica” technology, secure virtual machine architecture reference manual. Technical report, May 2005.

[3] Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. `http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`, 1967.

[4] Daniel Bartholomew. QEMU: a multihost, multitarget emulator. *Linux J.*, 2006(145), May 2006.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] Hewlett-Packard Company. netperf network benchmark. `http://www.netperf.org/netperf/`, 2005.

[7] Intel Corporation. 8259A PROGRAMMABLE INTERRUPT CONTROLLER. `http://pdos.csail.mit.edu/6.828/2005/readings/hardware/8259A.pdf`, 1988.

[8] Deutsche Telekom AG. Telekom’s high-security cell phone obtains Federal Office for Information Security (BSI) approval. `http://www.telekom.com/media/enterprise-solutions/200664`, September 2013.

[9] Jacek Galowicz. Live migration of virtual machines between heterogeneous host systems. Master’s thesis, RWTH Aachen University, 2013.

[10] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[11] Irfan Habib. Virtualization with KVM. *Linux J.*, 2008(166), February 2008.

[12] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2006.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2011.

[14] Intel Corporation. The 8086 User’s manual, October 1979.

[15] Intel Corporation. Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture. Technical report, April 2005.

[16] Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions, 2012.

[17] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's ManualCombined Volumes 3A, 3B, and 3C: System Programming Guide. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`, 2013.

[18] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Linux Symposum 2007, 2007.

[19] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992.

[20] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.

[21] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[22] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[23] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, Vol. 6:4–15, February 2002.

[24] Jun Nakajima. KVM Forum 2012: Enabling Optimized Interrupt/APIC Virtualization in KVM. `http://www.linux-kvm.org/wiki/images/7/70/2012-forum-nakajima_apicv.pdf`, November 2012.

[25] J.K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[26] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[27] Stefan Hajnoczi. QEMU Internals: Overall architecture and threading model. http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html, March 2011.

[28] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of EuroSys 2010*, Paris, France, April 2010.

[29] Jie Tao, Karl Furlinger, Lizhe Wang, and Holger Marten. A performance study of virtual machines on multicore architectures. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 0:89–96, 2012.

[30] Jia Tian, Yuyang Du, and Hongliang Yu. Characterizing smp virtual machine scheduling in virtualization environment. In *Proceedings of the 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, ITHINGSCPSCOM '11, pages 402–408, Washington, DC, USA, 2011. IEEE Computer Society.

[31] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 6–7 2004.

[32] VMware, Inc. Timekeeping in VMware Virtual Machines. http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf, 2011.

[33] VMware, Inc. The CPU Scheduler in VMware vSphere® 5.1. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf, February 2013.

[34] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *In Proc. of 15th PACT*, 2006.

[35] Phillip B. Gibbons Xiaoning Ding and Michael A. Kozuch. A hidden cost of virtualization when scaling multicore applications. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, HotCloud 2013, San Jose, California, USA, June 2013. Usenix.