Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master’s Thesis Nr. 83

Systems Group, Department of Computer Science, ETH Zurich

Multicore Virtualization over a Multikernel

by

Lukas Humbel

Supervised by

Stefan Kaestle, Adrian Schüpbach, Timothy Roscoe

5.5.2013

Informatik
Computer Science

Abstract

The recent trend towards more CPU cores has led to the design of the multikernel operating system architecture. Applications and operating systems, multikernel based or not, are adapted to take advantage of multiple CPUs. When executed under virtualization, these applications need multiple CPUs to achieve good performance. This thesis studies the design choices and the performance of an implementation of a multi core virtual machine monitor on top of a multikernel.

Whenever possible the design uses operating system facilities, for example the message passing system or the access control. Our solution is capable of running an unmodified Linux with comparable performance to that of KVM. We show that Linux requires low latency interprocessor interrupts to perform well. Additionally, we found a polling based messaging mechanism unsuitable for emulating those interrupts on the host side.

The performance evaluation revealed the slowdown caused by the need of multiple memory indirections when running under virtualization. Moreover, the effects of gang scheduling on performance are quantified when applied to virtual CPUs. This specific type of scheduling denotes the execution of all virtual CPUs at the same time. We have observed workloads that perform not only better or equal under gang scheduling, but worse too.

The effects of placing virtual CPUs on varying NUMA nodes have been investigated. On our test machine, a memory bound microbenchmark has shown considerable effects, while on other workloads they were non existent or minimal.

Contents

1 Introduction 4

2 Virtualization on x86 6
2.1 Virtual Machine Terminology . . . 6
2.2 Virtual Machine Overview . . . 6
2.3 Virtualizing Memory . . . 7
2.4 Interrupt Handling . . . 8
2.5 ACPI . . . 9
2.6 Multicore Virtualization . . . 9

3 Barrelfish 11
3.1 Process Management . . . 11
3.2 Capabilities . . . 12
3.3 Communication . . . 12
3.4 Old VMkit Architecture . . . 12

4 Literature Review 14
4.1 vCPU Scheduling . . . 14
4.1.1 VMware ESX . . . 16
4.1.2 Xen . . . 16
4.1.3 KVM . . . 17
4.2 Related Work . . . 17

5 Approach 19
5.1 VMM Design . . . 19
5.2 Memory . . . 20
5.3 Physical Interrupt Handling . . . 21
5.4 Interprocessor Communication . . . 21
5.5 Interrupt Controller . . . 21

6 Implementation 23
6.1 VMM / VM Interaction . . . 23
6.2 Virtual Hardware . . . 24
6.2.1 CPU . . . 24
6.2.2 APIC . . . 25
6.2.3 ACPI . . . 26
6.3 Implementation Issues . . . 27

7 Evaluation 30
7.1 Test Environment . . . 31
7.2 Benchmark Description . . . 31
7.3 Scenario 1: Performance in a non-contended setting . . . 34
7.3.1 Microbenchmark: World switch . . . 34
7.3.2 Microbenchmark: Forkbench . . . 35
7.3.3 Microbenchmark: Hackbench . . . 35

7.3.4 Microbenchmark: Rambench 36
7.3.5 Case study: Linux kernel compilation 36
7.4 Scenario 2: Effects of vCPU placement 37
7.4.1 Single core placement 37
7.4.2 Dual core placement 39
7.5 Scenario 3: Scheduling strategy 40
7.5.1 Method 40
7.5.2 Results 41

8 Conclusion **44**
8.1 Future Work 45

9 Appendix **47**
9.1 References 47
9.2 Raw Benchmark Results 49

1 Introduction

Many hardware virtualization solutions have been developed both for commercial use and in academics. They have proven useful in solving a broad range of tasks. Virtual machines are used to run multiple operating systems in parallel providing isolation from each other. Servers can be consolidated to increase utilisation of hardware resources, improve energy efficiency and save physical space. To enable migration, virtual machines can be stopped and resumed at any point in time.

The hardware landscape has changed significantly during the last decade. Nowadays, multi-core systems are standard in commodity hardware and the trend towards integrating more and more processing cores on a chip continues. Hence it follows that applications and operating systems are adapted to take advantage of multiple cores.

To cope with the increasing core count at the operating system level, one idea is to use the multikernel architecture. A multikernel operating system employs independent kernels on each CPU[1]. Instead of relying on shared memory, communication between cores is made explicit by using message passing.

To provide multi core optimised applications a familiar environment, it is necessary to provide multiple CPUs to a system under virtualization. There are already several virtualization systems on the x86 architecture that offer this feature. While they all provide a very similar interface to the guest, their implementation differs.

Since the guest operating system expects to run on bare hardware, it assumes to own the hardware completely. To sustain this illusion in a multi core VM[2], the scheduling of the emulated CPUs must be done carefully. How to schedule virtual CPUs fair and without degrading VM performance is still a matter of research.

The aim of this thesis is to find a suitable implementation of a multi core virtual machine running on a multikernel based operating system. We examine how to implement the virtual machine monitor in a secure and well performing fashion using the facilities provided by a multikernel system. These facilities include capabilities for rights management, messaging infrastructure and fine grained control over scheduling parameters. Due to time constraints, we were only able to use AMD's virtualization extensions. As Intel's virtualization extensions are similar, our design decisions made are applicable to Intel's implementation too. Although the design supports an arbitrary number of CPUs, we have limited our evaluation to two CPUs.

In order to measure performance, the virtual machine should be able to run an unmodified Linux. The performance of our implementation is compared to the Linux kernel based virtual machine and to native execution. Once the VM is started, we keep the allocation of virtual CPUs to physical CPUs static and we allow at most one virtual CPU to run on each physical CPU. Therefore,

[1] Central processing unit

[2] Virtual Machine

the effect on performance of such different allocations is evaluated. A number of workloads are used to identify non-uniform memory architecture (*NUMA*) effects.

To gain a better understanding of the implications of different scheduling strategies, different workloads are executed under gang scheduling, where all virtual CPUs run concurrently and under a forced asynchronous scheduling. We avoid scheduling optimisations in order to get reproducible worst case results. This method is useful regarding the study of the guest's behaviour. Due to the lack of obvious optimisations, it is not possible to rate real world scheduling techniques from our results.

Overview

An overview of the relevant subsystems for multiprocessing on x86 and the mode of operation of x86 virtualization gives Section 2. In Section 3, the subsystems of Barrelfish used by our virtual machine are explained. These include process management, messaging, capabilities and the old state of the virtualization solution, on which our implementation is based. Section 4 analyses problems that multi core virtual machines face. Currently implemented and proposed solutions to these problems are reviewed. Section 5 discusses the design decisions made, whereas in Section 6 the implementation on top of Barrelfish is examined. Section 7 evaluates whether our goals are achieved or not and presents the results of some experiments concerning core placement and scheduling strategy. Eventually, Section 8 concludes the work by reviewing our design decisions and performance results.

2 Virtualization on x86

2.1 Virtual Machine Terminology

Virtual machine is a commonly used term. In this thesis, virtual machine (*VM*) refers to a system which provides the software running under it a duplicate of real PC hardware. The *host* is the machine on which the virtual machine is running. Physical CPU (*pCPU*) refers to a CPU of the host.

The *guest* is the software running within the virtual machine and is usually a whole operating system. Virtual CPU (*vCPU*) refers to a (simulated) CPU of the guest.

The virtual machine monitor (*VMM*) is the controlling part of the virtual machine. It emulates the hardware, manages memory for the guest and communicates with the host operating system. It is often also referred to *Hypervisor*.

One distinguishes between a *type 1 VMM* [Gol73] which runs directly on the host hardware, while a *type 2 VMM* runs on a host operating system. Examples for a *type 1 VMM* are Xen and VMWare ESX. A popular example of a type 2 VMM is the Linux kernel-based Virtual Machine (*KVM*). There is some dispute if KVM is a pure type 2 hypervisor, because it requires a kernel module to be loaded in the host operating system. However, the kernel module is necessary if a VMM wants to use hardware extensions and KVM keeps as much as possible in the user space. A context switch from the VM to the VMM is called a `VMEXIT`. This happens for example when the guest wants to access the hardware. The opposite context switch is called a `VMENTER`. This in turn happens for instance when the VMM has emulated the requested hardware access and wants to continue the VM execution.

2.2 Virtual Machine Overview

There are different techniques how to implement a virtual machine (see [San09] for an elaborate discussion).

As the x86 architecture does not fulfil the virtualization requirements stated by Popek and Goldberg [PG74], it is not possible to run a whole operating system unprivileged like a normal user process. The requirements state that every instruction should either behave the same in all privilege levels, or it should always trap when running unprivileged. An example of a problematic x86 instruction is `POPF`. When running privileged, the operation can be used to set the interrupt enable flag. As it is undesirable to let unprivileged processes change the interrupt behaviour, the instruction ignores the interrupt flag when executed unprivileged. This is a violation of the virtualization requirements, as the instruction should either behave unaltered or trap into the operating system.

To enable virtualization anyway, there are several ways around. First, one could emulate the whole machine and not let anything run directly on the hardware. As emulation adds a lot of overhead, the usefulness of such a solution is limited.

To speed up emulation, the host could not directly emulate all operations but emit code that is doing so. This approach is called *binary translation*. Using this approach, VMware was able to virtualize x86 with acceptable performance and no help from the hardware.

Another idea is not to fully virtualize every aspect of the platform, but to modify the guest operating system to allow correct execution. This approach is called *paravirtualization*. The big drawback is the need for modifying the guest operating system limits the choice of available guest operating systems. In the case of commercial operating systems, the vendor must cooperate with the hypervisor manufacturer to include the necessary modifications.

As the hardware vendors realized the need for virtualization, they started to include hardware extensions which allow the guest to run unmodified while taking full advantage of the hardware. The extensions give the VMM fine grained control when to transfer the control back from the guest to the VMM.

We focus our work on the x86 architecture and on virtual machines which make use of these hardware extensions (either AMD SVM or Intel VT). The basic mode of operation for both extensions is based on "trap and emulate". The VMM sets up a block of memory which describes the state of the virtual CPU according to the CPU vendors specification. It then uses a special (privileged) instruction to start the VM, providing the memory region as an argument. The VM runs until an exit condition occurs, for example it may perform I/O or access an unmapped memory region. Then, the CPU restores the state it had before entering the VM and the control is transferred to the VMM. The VMM examines the exit condition to determine what action caused the exit and emulates the action the virtual CPU intended. After the emulation, it re-enters the VM by performing again a `VMENTER`.

2.3 Virtualizing Memory

The VMM must take care that the guest can only access the memory designated to it. The guest operating system expects to have direct access to the physical memory and the currently used page table. Clearly, this operation must be limited, otherwise the guest is able to access any physical memory region.

To do so, the x86 virtualization solutions offer two mechanisms. The first is so called *shadow paging*, which works without hardware assistance. The VMM intercepts all access to the page table register and translation lookaside buffer (*TLB*) related instructions. Once the VMM catches such an access, it inspects the page table and constructs a new page table which does not map guest virtual to guest physical addresses but maps directly guest virtual to host physical. As this operations are quite costly and introduce additional `VMEXIT`s, hardware manufacturers started to include improved MMUs which are able to perform this operation in hardware.

Today, most systems support these hardware extensions to allow so called *nested paging*. The nested paging mechanism implemented by AMD is also called Rapid Virtualization Indexing (*RVI*). Nested paging involves two page tables, the outer page table is set by the VMM and the inner page table is set by the guest operating system. To translate a *guest virtual address*, first the inner page table set by the guest operating system is traversed to find the *guest physical address*. The found address is translated further into a *host physical address* using the page table provided by the VMM. The hardware page table walker has to perform a lookup in the outer page table at every step of the walk of the inner page table. Hence the number of lookups is squared compared to a non virtualized lookup. Once a mapping is found, a direct guest virtual to host physical mapping is inserted into the TLB.

A guest and VMM context switch can happen without flushing the TLB by making use of different tags for each domain. To save memory, a VMM may not directly allocate and map all memory for the guest, but catch page faults from the guest and install a mapping on demand.

2.4 Interrupt Handling

As x86 CPUs provide only one interrupt line for general purpose interrupts, the interrupts from different sources must be multiplexed to this line. Before the appearance of systems with multiple CPUs, x86 systems usually used the Intel 8259 Programmable Interrupt Controller (*PIC*). A single chip is able to multiplex eight interrupt lines to one. To increase the number of interrupt inputs, two or more PICs can be cascaded. Typically two of them were used, providing 15 interrupt input lines. Once the CPU receives an interrupt, it reads the interrupt vector from the PIC using additional lines.

A drawback of this design that only one output is supported. Hence, in a system with multiple CPUs, only one would be able to receive interrupts. Also the limited number of interrupt lines made it hard to create flexible hardware extensions.

To overcome these limitations, the Advanced Programmable Interrupt Controller (*APIC*) has been designed. It replaces and extends the functionality of the *PIC*. The biggest improvement over the PIC is the ability to route interrupts to any core. This is important to distribute the interrupt load among cores and enables true symmetric multiprocessing (*SMP*). SMP is a type of multiprocessor architecture in which every core is able to perform the same set of tasks, such as accessing the same memory, using I/O and, as mentioned, receiving interrupts.

The APIC system consists of a core local part, the so called local APIC. The interface to the hardware consists of one or more I/O APICs. These APICs are connected over a bus to transmit interrupts. The operating systems can configure the I/O APIC to transfer interrupts to any local APIC. As an additional interrupt source, each local APIC contains a core local timer.

It is desirable that the operating system can trigger an interrupt on a remote core. To do so, the APIC supports interprocessor interrupts (*IPI*). The OS can command the APIC to send an IPI to any subset of cores. IPIs are also used to boot secondary CPUs. Hence, the APIC is a crucial component for supporting multiprocessing on x86.

Most systems still contain a PIC compatible interface to offer interoperability with legacy operating systems. The PIC can also be used together with the

APIC. To do so, the APIC is able to forward interrupt requests from a PIC to the CPU. When using this combined setting, only the first CPU is able to receive interrupts from hardware, but the secondary CPUs can still use the core local timer and IPIs.

2.5 ACPI

Modern systems use the Advanced Configuration and Power Interface (*ACPI*) to perform device discovery and energy management. The operating system access the information by reading multiple memory mapped tables.

Some tables contain complex data structures, such as a device tree, or executable methods. These are written in a human readable format, called the ACPI source language and are compiled and stored in byte code. The operating system uses an interpreter to extract the structures or execute the methods.

Of relevance for multiprocessing is the Multiple APIC Descriptor Table. Operating systems use this table to determine the number of installed CPUs and their physical identification numbers.

The first multi processor x86 systems were produced before ACPI was finalised. Hence, they offered the information to the operating system in different format. The systems used the Intel Multiprocessor Specification (*MPS*).

2.6 Multicore Virtualization

Extending this trap and emulate strategy to multiple cores is straightforward: For each additional CPU the VMM starts a new thread which performs trap and emulate until the virtual machine is stopped.

However, there are several design choices when it comes to virtual hardware. Either the emulation is permitted in each CPU execution thread. Then the virtual devices must be aware of that and use locking where appropriate. Or, one can use a single thread for all virtual hardware.

The virtual hardware itself may perform long running operations, such as disk I/O. To provide an asynchronous interface to the guest, either the virtual hardware must use asynchronous host APIs or start a new thread for such long running operations. In cases where the virtual hardware must perform long running computations, the use of asynchronous APIs is not possible.

As x86 is a shared memory system, each CPU should see the same memory. Hence for each address translation from guest physical to host physical the same mapping can be applied. In the case of nested paging, this implies the outer page table can be shared by all vCPUs.

Having additional cores running in isolation is not useful. A platform that supports SMP (or multiprocessing in general) must provide some infrastructure to allow CPU discovery, communication and notification. As discussed, x86 provides these mechanisms through ACPI, shared memory and IPIs provided by the APIC system.

Hence to support a x86 multi processor guest operating systems, the mentioned facilities have to be emulated. There is a number of commercial and free products that provide a multi core virtual machine, in this thesis we will examine VMWare ESX, Xen, KVM and our own solution VMKit.

3 Barrelfish

Barrelfish [BBD+09] is a research operating systems developed by the Systems Group at ETH Zurich, it uses a novel multikernel architecture. It is designed to cope with recent and future hardware trends. It is built on the assumptions that the number of cores per chip increases and that they will become heterogeneous. Heterogeneous in the sense that not all cores may use the same instruction set, the memory access latency is not uniform and that the caches do not need to be coherent and accessible by all cores.

To cope with these trends, the *multikernel* architecture was designed. In such an OS, every core hosts one kernel, called the *CPU driver*. Between these kernels state is not shared, but replicated. If state across CPUs must be modified, it is done over explicit messages. This way, no locks are needed inside the CPU driver.

Each CPU driver is accompanied by a *monitor* instance. The monitor processes, one on each core, are responsible for providing global shared state to other processes. All monitors run an agreement protocol to keep the state consistent. The monitor is also responsible for setting up messaging channels.

3.1 Process Management

The CPU driver is responsible for multiplexing hardware resources to user-space processes. It is also responsible for scheduling and dispatch of the user-space processes.

User-space processes are called *domains* in Barrelfish. A domain consists of a virtual address space and a set of *dispatchers*. A domain which wants to execute on a given core needs to create a dispatcher on the desired core. Once the dispatcher is made runnable, it will enter the run queue of the CPU driver and gets scheduled. To allow the CPU driver to execute user space domains, it must keep some information, such as the runnable state, about the domains it owns. This information is saved in the dispatcher control block (*DCB*).

Thread management is done entirely in user-space. When the dispatcher is invoked from the CPU driver, it will not simply continue at the last executed instruction, but it will call the user-space dispatcher. The dispatcher will decide on its own which thread to run.

As a consequence of this design, the CPU driver cannot decide to put a thread on another core. Hence threads are immovable assigned to a dispatcher, which in turn is fixed to a specific CPU.

Barrelfish provides a generic way how to handle callbacks. When a user space application sets up a messaging channel it can provide callbacks which should be called when a message arrives. When a message arrives, the callback is not called immediately but it is inserted in an event queue called *waitset*. As there may be multiple waitsets, the process specifies the one it likes to use at binding time. To allow the user process to handle these callbacks at well defined times, it must explicitly dispatch events from the waitset. A process can request an event from a waitset in a blocking manner.

The user-space dispatcher also contains support for timed callbacks, so called *deferred events*. The dispatcher maintains a list of all not yet triggered deferred events. Every time the dispatcher is invoked from the CPU driver, it checks if it is time to execute an event. If so, it puts the event in the queue for the specified waitset. When a user thread dispatches events from the waitset the next time, the handler will be called.

3.2 Capabilities

Barrelfish uses capabilities to manage access to resources. A capability is an object which is attached to a process and represents a right to use or the ownership of a resource. So it can be seen as the opposite of an ACL, which is attached to the resource and lists all processes allowed to use the resource. Capabilities are stored inside the CPU driver and may only be manipulated by privileged code. A process has no direct access to its capabilities, but uses references and syscalls to perform actions on its capabilities.

3.3 Communication

Barrelfish uses explicit rather than implicit sharing of data structures. Implicit sharing denotes the access of the same memory region from different processes. Explicit sharing denotes replicated copies of the structures and synchronizing them using messages. Barrelfish employs *Flounder*, a domain specific language for describing interfaces. Flounder uses the descriptions to generate message passing functions. For the client, using these methods is transparent. Internally, flounder selects one from multiple possible backends at binding time.

Messages are passed over different backends, depending whether the communication crosses cores. If it the communication stays on one core, Barrelfish uses LMP. LMP stores the message payload directly in the CPU's registers. If the communication goes cross-core, UMP is used. UMP stores the payload in memory. The receiver polls the memory to receive the message. To keep the bus traffic as low as possible, the payload size matches a cache line size.

Another backend available is the UMP-IPI backend. In addition to the mechanics of UMP, it sends an IPI to the destination core. On the destination core, the CPU driver checks which process is receiving the message and schedules the process immediately.

3.4 Old VMkit Architecture

Raffaele Sandrini [San09] wrote a single core VMM as his master thesis. It is capable of running Linux and its performance is comparable to that of KVM. However, the set of supported hardware is just enough to run Linux, it emulates an AMD K6 CPU. It contains a virtual serial console. For interrupt handling, it emulates a programmable interrupt controller (*PIC*).

It is split into two domains, one containing the VM, one containing the VMM, as depicted in Figure 1. When the CPU driver decides to schedule the

VM, it does not perform a normal dispatch but performs a `VMENTER`. Whenever an exit condition occurs, a message is passed to the VMM, which handles the `VMEXIT` and makes the VM domain runnable again.

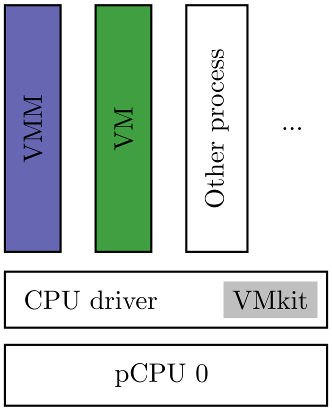

Figure 1: Old VMkit process layout. The kernel part of VMkit is involved when a domain is dispatched.

4 Literature Review

4.1 vCPU Scheduling

The threads of a multi core VM must be scheduled by either the host operating system (in case of a type 2 VMM) or by the hypervisor itself (in case of a type 1 VMM). Scheduling VM threads is different than scheduling usual processes. VM threads run an operating system which expects to run on bare hardware, so they make assumptions on how long certain operations will take. However, if we schedule the VM as we schedule normal processes, different vCPUs may be scheduled at different times, destroying the illusion of completely owning the hardware. When scheduling the vCPUs out of sync, the following problems (among others) may be observed.

- Guests use spinlocks to synchronise access to kernel data structures. This is tolerable, as long as the guest operating system can guarantee that the lock will only be hold for a short period. For instance, when the lock is only used to protect short, non preemptable kernel sections. When the VMM decides to preempt a vCPU which currently holds a lock, it may be impossible for other vCPUs to make progress. If the VMM takes no care of this problem and Linux is used as guest, this can have a major performance impact. We call this problem *Lock holder preemption*, as first identified by Uhlig *et al.* using L4Linux [ULSD04].
- Another thing observed in Linux is the use of an IPI call response protocol. The initiator sends an IPI and waits until a reaction of the receiver. The waiting is done by polling a memory location in a non preemptable kernel section. On real hardware, interrupts get handled almost immediately, hence it is tolerable to block execution for this short timespan. This protocol may also be invoked by user space applications which do not perform any explicit synchronisation.
- Theoretically it may be possible that an operating system uses a CPU as a watchdog to observe progress that other CPUs make. In a virtualized setting, this may lead to false alerts, if an observed CPU isn't scheduled at the same time as the watchdog CPU. VMware uses this example to justify their scheduling implementation [VMwb].

There are different ways to cope with this problem. One way to sustain the illusion of completely owning the hardware is to use *gang scheduling.* Gang scheduling (also known as *co-scheduling*), executes all vCPU threads belonging to one VM at the same time. It restores most of the assumptions the guest OS makes and solves all of the problems mentioned above, including the watchdog example. However, gang scheduling severely limits the flexibility of the scheduler and the schedulers itself must synchronise across CPUs, which can be costly. Gang scheduling may lead to fragmentation and therefore suboptimal CPU utilisation. Imagine a dual core host system running a dual core VM and a single threaded process. While the process is running, one CPU must idle as the gang scheduling forbids running only one vCPU.

A virtual machine may avoid the worst effects by not preempting a lock holder. However, it is hard to detect for a VMM when its guest holds a lock. Uhlig *et al.* noticed that the problematic locks occur almost exclusively in the kernel, hence they propose to preempt vCPUs only when running in user-space. Detecting if the preempted CPU is running in kernel or in user-space can be done by inspecting the privilege level of the CPU. Using this method they were able to achieve good performance when using Linux as a guest. However, as vCPUs are not preemptable at any point in time anymore, the scheduler may not be able to guarantee fairness.

Another possibility is to detect when a CPU needs to wait for a long time until it can acquire a lock. Then the VMM may schedule another vCPU which can make progress. This may either be done by a paravirtual extension of the guest or the VMM may monitor the activity of the CPU. Linux for instance supports paravirtualized spinlocks. AMD supports a triggering-threshold for many subsequent `PAUSE` instructions. Pause instructions are typically used inside the spin loop to relax the CPU. A problem is that for this approach the VMM has to know how many subsequent `PAUSE` instructions actually imply lock holder preemption. When the limit is set too low, many false positives are detected and unnecessary many context switches happen. Setting the threshold too high increases the time needed to detect unnecessary spinning and hence lowers utilization. However, at least when using Linux as guest, the approach works well. If one wants to implement priority inheritance to schedule the lock holder instead of the spinner, this technique is not useful, as it does not indicate which CPU is actually holding the lock.

The Linux kernel also offers ticket locks, ticket locks guarantee fairness by imposing a FIFO order on the lock acquiration. Once one process releases the ticket lock, there is at most one vCPU which is able to acquire the lock. Without using paravirtualization, the scheduler is not able to figure out which vCPU this is. Hence when scheduling vCPUs without this information, the chance of hitting the right vCPU gets smaller the more vCPUs the system has.

[Fri08] has shown, using two 16 vCPU VMs running under Xen, that the effect of spinning may account of up to 7.6% of guest execution time. Even worse, when using ticket spinlocks, up to 99.3% of guest time is spent waiting for the lock. Using paravirtualized spinlocks, they were able to reduce the overhead to under 3.6% and in the case of ticket locks to under 5.4%.

A similar approach may be taken to solve the IPI latency problem. When a core sends an IPI it can be interpreted by the VMM as a directed yield, allowing the receiving CPU to react quickly.

Preempting the vCPU only when in user-space and the spin detection solve only the lock holder preemption issue. They do not help in the watchdog example. However, it seems that lock issue is the main contributor to performance degradation and current operating system don't make use any watchdog like mechanism.

In the following survey, we examine how existing products handle this problem.

4.1.1 VMware ESX

VMware implements a proportional share-based algorithm [VMwb]. It assigns each VM a user specified share of CPU time. The user may also specifies hard lower limits (*reservation*) or hard upper limits (*limits*). The VMM will never assign fewer resources than specified and never more. If it cannot met the lower limits, it will not let the VM start. A vCPU has higher priority, if its used vCPU time is further away from the user specified proportional share.

ESX 2.0, the first version of VMwares VMM that supported SMP, uses a gang scheduling approach [VMwa]. The scheduler accounts skew for each vCPU. The skew of a vCPU increases, when a sibling vCPU makes progress while the original isn't doing so. If the skew of a single vCPU exceeds a given limit, typically a few milliseconds, the scheduler stops the entire VM. The VM will only be rescheduled if there are enough pCPUs available to start all vCPUs at the same time. This way, the skew can only become smaller.

As discussed, strict co scheduling may lead to CPU fragmentation. VMwares slightly modified approach does not avoid this problem completely.

To overcome this limitation, ESX 3.0 introduced *relaxed co-scheduling*. It still accounts each vCPU with skew, but requires only those vCPUs which are skewed to be scheduled. This ensures that no vCPU can lag behind the others too much. Relaxed co-scheduling reduces the chance of fragmentation, therefore improving CPU utilization.

4.1.2 Xen

Xen was initially built for paravirtualization, but it now supports unmodified guests using hardware extensions. Xen currently uses its credit scheduler by default. Credits arrive at a fixed period and each vCPU receives a user defined amount of credits. The scheduler checks frequently which vCPU is running and accounts the running time to it by taking away credits. vCPUs which have a positive credit balance are preferred by the scheduler. This is a proportional fair share CPU scheduler, similar to the one of ESX, although it has no support for gang scheduling.

Southern *et al.* [SHB09] have shown this may lead to performance degradation for synchronisation heavy workloads in a contended setting. As Xen originated from a paravirtualization hypervisor, it contains support for diverse paravirtualized optimizations. One of these is the mentioned paravirtualized spinlock supported by Linux.

Xen also implements a mechanism to prefer vCPUs which receive an interrupt. If the guest issues the `HLT` instruction, the vCPU is interpreted as blocked. Once an interrupt arrives and the concerning vCPU has credits left, the vCPU enters the boost mode and is inserted at the front of the run queue.

This guarantees that the CPU is executed at the beginning of the next time slice.

4.1.3 KVM

While Xen and VMware ESX are type 1 hypervisors, the Kernel-based Virtual Machine (*KVM*) is embedded into the Linux kernel and makes heavy use of its features. KVM denotes only the part running inside the Linux kernel. The kernel part is responsible for executing guest code by using hardware features. Device emulation and management is done in the userspace counterpart, QEMU. It does not provide its own scheduler but relies on Linux, which currently uses the completly fair scheduler (*CFS*). CFS does not support any kind of gang scheduling. KVM uses task groups to group together the vCPU threads of a VM. This way the fairness between VMs with differing number of CPUs is maintained.

KVM uses hardware extensions for spin detection.

4.2 Related Work

Tian *et al.* [TDY11] examined if the Xen credit scheduler meets its goals. They observed that the scheduler maintains fairness only when assigning each CPU the same weight. Xen migrates vCPUs along pCPUs, they have found that the decisions made by the scheduler when to move a vCPU does not degrade performance. They also found that Xen performs poorly (they claim due to lack of gang scheduling) in a contended setting when running synchronisation heavy workloads.

Weng *et al.* [WWLL09] proposed to differentiate the scheduling strategy according to the VM workload. They distinguish between throughput and synchronisation oriented VM. Gang scheduling is applied only to the VMs classified as synchronisation type. The classification to which type a VM belongs can be changed at run time. The change must be initiated by the system administrator. Their proposed scheduler handles both classes of VMs.

Bai *et al.* [BXL10] proposed something very similar to [WWLL09], except that they employ an algorithm which makes the decision whether the VM should be gang scheduled or not.

Sukwong *et al.* [SK11] proposed an alternative to gang scheduling, balance scheduling. They use KVM (and therefore Linux CFS) where each pCPU has a local runqueue. Balance scheduling ensures that two sibling vCPUs are never put in the same runqueue. Hence increasing the probability that sibling vCPUs are scheduled at the same time. They claim to achieve, depending on the workload, almost as good or better performance than gang scheduling.

Govindan *et al.* [GND$^+$07] proposed an alternate communication aware scheduler for Xen for multi-tier server applications.

Bae *et al.* [BXDL12] investigated how the placement of vCPUs on different pCPUs affects power, energy and performance.

Cherkasova *et al.* [CGV07] compared the three available schedulers of Xen in a uniprocessor setting.

5 Approach

5.1 VMM Design

A multikernel operating system avoids shared memory in the kernel. This principle may also be extended to the applications running on the OS. In contrast, a goal of the virtual machine implementation is to run shared memory guests, such as Linux. The VMM must be able to access its guest's memory, hence a large part of memory must be shared by the execution threads which represent a vCPU. Also, the state stored in emulated hardware is shared between the CPUs. The only exception being is the local APIC, of which one exists for each CPU. Because of this, we decided to implement the VMM as one domain running on two pCPUs and share all memory among the two dispatchers.

On the old VMkit, the separation of the VM and the VMM into two domains (see Figure 1) leads to increased complexity in the CPU driver. Also the context switch from the VM to the VMM involves two context switches. This can be reduced to one when the VM and VMM are running in the same domain. Hence, we decided to do exactly the following: Remove the separation and create only one domain, containing the VM and the VMM. Figure 2 illustrates the new design.

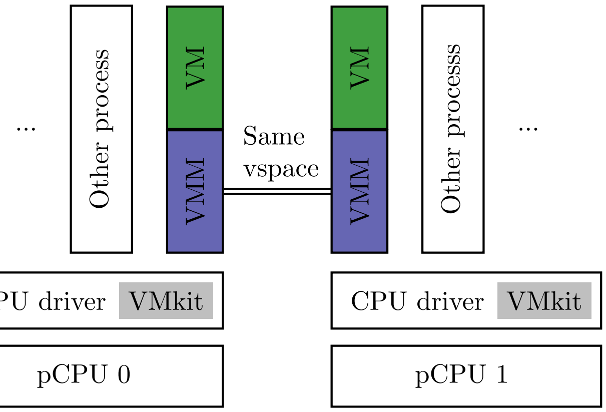

Figure 2: The new VMkit design. VMM and VM are in the same domain. The role of the kernel part is to handle requests from the VMM to enter the VM.

As the VMM makes use of hardware extensions, it must be able to invoke privileged instructions. To restrict access to these privileged instructions, we introduce a new capability which has two invokes.

One to activate the hardware extensions, `VMSETUP`. `VMSETUP` checks if the current CPU supports the required extensions and initialises them. It has to be called on every CPU before making use of `VMENTER`. If the hardware is not capable, an error code is returned. The other invoke actually enters the VM, `VMENTER`. Figure 3 illustrates the control flow.

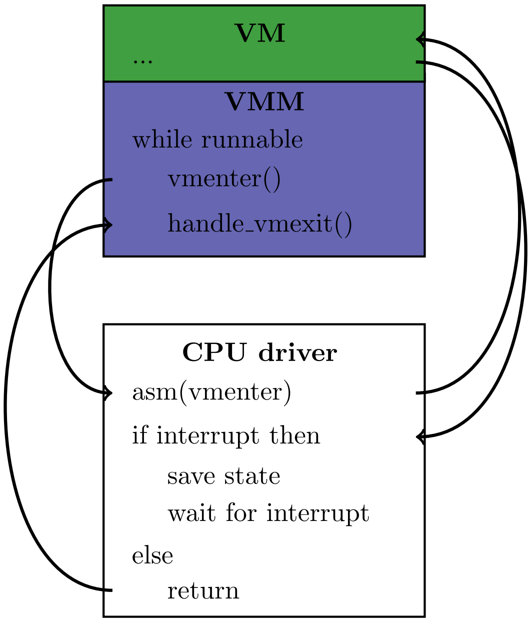

Figure 3: Non interrupted `VMENTER`/`VMEXIT` flow on one CPU.

Since the VMM runs one thread for each CPU and shares all memory, it must make sure that accesses to common data structures are synchronised. To do so, we introduce a coarse lock which is hold by the VMM during the handling of a `VMEXIT`. This may introduce a performance bottleneck. As there is no complex hardware emulated, we perform the emulation directly in the CPU thread. No implemented hardware requires long running operations, so this should not lead to performance problems. The only exception are being the timers, which use an asynchronous host API (see Section 3.1).

5.2 Memory

As nested paging makes the implementation of the VMM easier and faster, VMkit exclusively uses nested paging. The only drawback of this decision is that some older CPUs, which do not provide nested paging, are unable to run VMkit. We decided to make use of the tagged TLB, hence no TLB flush has to be performed when switching from the guest to the VMM or back. To further ease implementation, we use a fixed amount of memory provided to the guest. All memory is allocated and mapped in the outer page table before starting the guest.

5.3 Physical Interrupt Handling

When the VM is running and a non-virtual interrupt arrives at the host, the interrupt must be serviced by the host operating system. AMD's hardware extensions handle this case by generating a `VMEXIT`, the VMM must then take care that the interrupt gets handled. The VMM does so by resetting the interrupt flag. The CPU will then take the interrupt and jump to the interrupt service routine (*ISR*). The kernel part of the VMM must ensure that it saves its own state suitable for rescheduling. However, when the operating systems wants to schedule the VMM again, it does not know whether the VMM process was running the VM or running in normal VMM user space code. We decided not to add any kernel state to the process. This makes the VMM indistinguishable from any other process for the scheduler. However, this means that whenever the VMM gets rescheduled, it will first enter the VMM which in turn asks the OS to enter the VM. This may have a performance impact.

5.4 Interprocessor Communication

x86 does not support explicit message passing. So most data transfer is done using shared memory. However, unless the receiving core is constantly checking for changed memory, shared memory does not provide a way to signal another core. As explained in Section 2.4, x86 provides interprocessor interrupts to do so.

A VMM must take care of these interrupts and ensure that they are delivered with low latency. Low latency cannot be achieved by the VMM polling a memory location, because most of the time not the VMM is running but the VM. Hence we decided that the VMM triggers itself a physical IPI whenever a virtual IPI has to be delivered. This has the effect that the target core leaves the VM immediately and enters the VMM, which in turn can deliver the virtual IPI to the VM.

5.5 Interrupt Controller

As discussed in Section 2.4, to support multiprocessing on x86 a platform must offer an APIC.

Recent versions of the APIC (*x2APIC*) offer support for more CPUs and an optimized IPI interface. It also changes the interface from a memory mapped to a model specific register (*MSR*) mapped interface. The IPI interface is changed such that fewer accesses are needed to transmit those. This improvement may prove especially valuable in the context of virtualization, as every access causes an expensive `VMEXIT`. Also, a `VMEXIT` due to an illegal memory access is more expensive than a `VMEXIT` caused by a MSR access, because the page table has to be walked to detect the illegal memory access.

We decided, unless identified as performance drawback, not to use these possibilities, but to emulate a standard APIC. This is done by creating a hole in the memory map, such that an access to the APIC memory region causes a `VMEXIT`.

The operating system may choose where to map the APIC by setting a MSR of the CPU. This is done in a per core fashion. So in theory, an OS can decide to map the APIC at different locations for each CPU. This implies that the page table for the nested paging can not be shared across CPUs, as the holes may be at different locations. Fortunately, Linux does not remap the APIC, so sharing the page table is possible when running Linux as a guest.

The APIC can either replace the PIC when it is accompanied with one or more I/O APICs, or it can co-exist with one. When both exist, the PIC is wired to the local APIC of the first core. We decided not to implement an I/O APIC to reuse as much functionality as possible.

6 Implementation

6.1 VMM / VM Interaction

To merge the VM and the VMM into one domain the following steps have been performed. First, we added the VMM capability. At the moment, it can be created without any protection, using the same mechanism as the ID capability. To support untrusted applications, this must be changed, as owning the capability can be used to access any physical memory location. Apart from obtaining the capability, the security is maintained.

Then we implemented the two invocations of this capability `VMSETUP` and `VMENTER`. `VMSETUP` did not change, so we were able to re-use the old code. We transferred the specialised dispatch code to the `VMENTER` invocation. Whenever a `VMEXIT` occurs, we return from the invocation to the VMM. In the special case where the exitcode indicates the occurrence of a physical interrupt, we tried enabling the interrupts and let the CPU process the interrupt by calling the `hlt` instruction. Such that the operating system can handle the interrupt immediately. However, this lead to problems when the VMM gets rescheduled, as the state was not saved correctly. As a temporary solution, we did not treat the interrupt caused `VMEXIT` as a special case, but always returned to the VMM from the invocation. This way we were able to boot Linux.

Later on, as performance problems appeared, we changed this code to handle interrupts correctly. As when sending a message or performing a yield, we save the VMM's state in the DCB when calling the `VMENTER` invocation, enable interrupts and execute the `hlt` instruction. This leaves the domain in a state suitable for rescheduling. As this change made performance even worse, we noticed that the delay between interrupt detection and interrupt handling can be as high as $80ms$, which is the duration of Barrelfish's scheduling period. Clearly, we identified too many `VMEXIT`s as caused by physical interrupts. One case where this happens is the occurrence of a system management mode interrupt (*SMI*). We decided that the most generic solution is not to wait until an interrupt happens, but just to check it. This is done by enabling interrupts for a short time and then disabling it again. If no interrupt has happened the current code will not be preempted and we re-enter the VM without any context switches.

This implementation leads to no wasted time when waiting for an interrupt, while still allowing a simple return from the syscall for `VMEXIT`s which are handled by the VMM. To be able to save the state to the DCB, we must be able to access information, such as the instruction pointer, from the caller, which is not available to normal invocations. We added a special case in the capability invoke code which passes this information as arguments to the invoke handler.

The VMM itself needs some change to handle the new design and to make use of this new capability. Instead of a message dispatching endless loop it now enters an endless loop which enters the VM, handles the exit condition and re-enters the VM. Instead of creating a new domain, the VMM now just allocates frames for the virtual machine control block (*VMCB*), which saves the state of the virtualized CPU, the guest memory and the page table. To fill the

VMCB, we need to retrieve the physical base address of the page table. In the old design, this was not needed, as it was saved in the DCB of the VM domain. To do so, we added a identity invocation, similar to the existing one for generic frame capabilities, also for page table capabilities. As we want to use the page table across two cores and capabilities are core local, we cannot simply use a capability reference as parameter.

Finally, a lot of code can be removed. The DCB does not need to contain any information about the domain being a VM and the dispatch code does not need to know how to dispatch these specialised domain.

6.2 Virtual Hardware

6.2.1 CPU

The old VMkit emulated an AMD K6 CPU, which does not support neither multiprocessing nor an APIC. Hence the vCPU has been upgraded to emulate an AMD K7 processor. This is basically done by changing the information provided by the `cpuid` instruction and implementing the new CPU features. The VMM was extended to support the additional MSRs of the new CPU.

To prepare the VMM to support multiple CPUs, all CPU concerning information is factored out from the guest structure into a newly created CPU structure. This required a large number of changes, however most of them were trivial. The Linux real mode boot code of the secondary processor uses a different instruction to access the CR0 register than the boot code for the primary processor. Hence, support for the `lmsw` instruction was added.

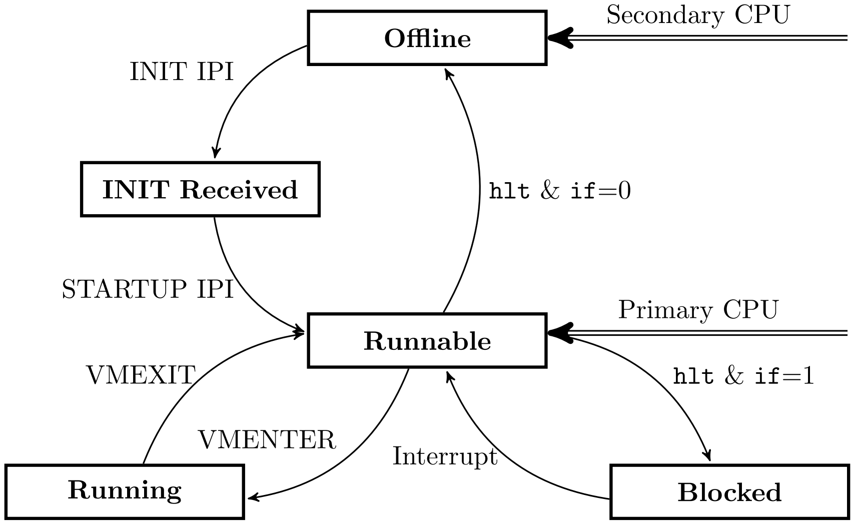

Figure 4: The states of the emulated CPU. `if` denotes the interrupt flag.

We took care to handle all CPU states correctly. The CPU state is reset whenever a transition to the INIT received state happens. Figure 4 illustrates all CPU states and transitions. `if` denotes the interrupt flag of the CPU. Interrupts can only be taken by the CPU when the flag is set. The `hlt` instruction is used to let the CPU sleep until an interrupt happens.

The CPU is interpreted as being offline when the `hlt` instruction is executed and interrupts are disabled. This careful implementation allows *CPU hot-plugging*. The guest operating system can shut down and reboot a CPU at any time.

6.2.2 APIC

As mentioned in Section 2.4, the Advanced Programmable Interrupt Controller (*APIC*) is crucial to support multiprocessing on x86. As discussed in Section 5.5, we have chosen to implement a standard APIC and intercept access to it by creating a hole in the guest's memory map. The invalid memory access `VMEXIT` handler checks if the faulting address is in the region of the APIC memory. If it is, the VMM will emulate the access to the APIC and continue the VM.

Apart from the interrupt routing ability similar to the one of the PIC, the APIC also provides a core local timer and the ability to send IPIs.

As we have chosen to let the PIC be available in the system and not to emulate any I/O APICs, nothing had to be done for interrupt routing, as the PIC is still used to deliver interrupts from the emulated hardware. However, in this co-existing of PIC and APIC, one has to make sure that not only interrupts from one controller are delivered. To ensure this, after the PIC receives an end-of-interrupt (*EOI*) command, we first deliver the interrupts from the local APIC and vice versa. On booting the first core, Linux calibrates the APIC timer against the legacy PIT timer. If the fairness between both interrupt controllers is not guaranteed, the calibration fails with some probability.

Figure 5 shows all entities which participate in interrupt handling during runtime.

To implement the local APIC timer, we started by using Barrelfish's timer service. As this leads to locks when running two CPUs, we replaced all used timers, including those used by the legacy PIT, with the now recommended deferred events (see Section 3.1). Whenever the timer fires, we set the corresponding bit in the interrupt ready register. If no interrupt is currently being processed, we deliver the interrupt to the CPU. If an interrupt is in service, we deliver the interrupt after operating system has issued an EOI action. As the APIC specification does not require a specific frequency to be used, we have chosen $100MHz$.

To handle IPIs, each local APIC is connected over a Flounder binding to an APIC bus instance. If one local APIC receives from the guest the command to transmit an IPI, a message is sent to the bus instance. The message consists of the contents of the interrupt control low (*ICRL*) and interrupt control high (*ICRH*) APIC register, together they encode the destination and the interrupt vector to activate. The bus then examines the registers and figures out which of

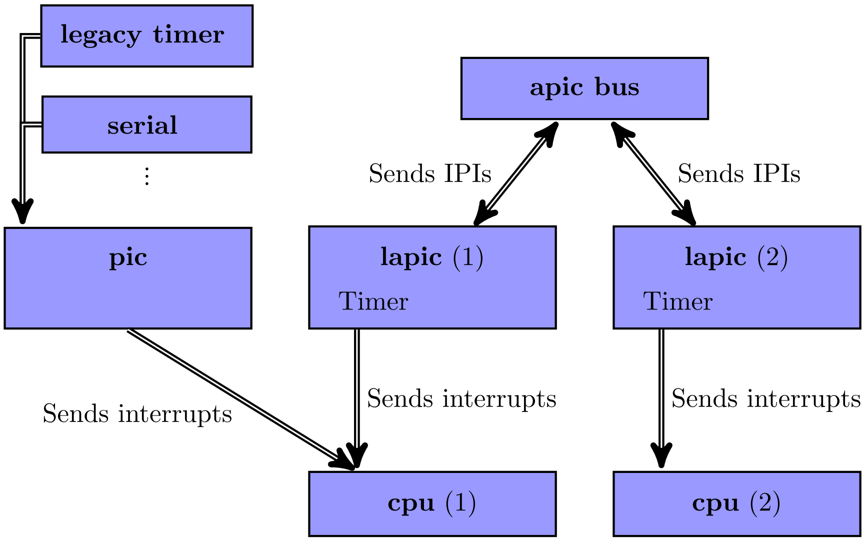

Figure 5: Object instances that participate in interrupt handling.

the CPUs should receive the IPI. The APIC supports different addressing modes, we added support for the ones Linux uses. First, there is the physical addressing mode. In this mode, an IPI is sent to exactly one core, the one encoded in the destination field. After boot, Linux switches to the logical destination mode. Each APIC contains a logical ID field which can be set to whatever the OS considers appropriate. When using the logical destination mode, an IPI is sent to a CPU if its logical ID and the destination register have at least one bit in common. Using this mode, it is possible to selectively broadcast IPIs.

A particularly subtle issue is where to exactly start the CPU when a `STARTUP` IPI is received. The APIC specification [APM] states the CPU should start execution at the address specified by the vector field. The vector field is an 8-bit value and provides the upper 8-bits of a 20-bit physical address. As the CPU starts in real mode, there are multiple combinations of the code segment register (CS) and the instruction pointer (IP) which map to the desired physical address.

It turns out, Linux assumes that the IP is set to zero when the execution starts. Otherwise it loads an invalid global descriptor table (GDT) and an invalid local descriptor table (LDT), which causes problems that only start to appear after the switch into protected mode is completed.

6.2.3 ACPI

As discussed in section 2.5, ACPI consists of multiple memory mapped tables. To locate the tables, the OS looks for the root system descriptor pointer ($RSDP$), by scanning some fixed BIOS memory locations for a signature. The

RSDP points to the root system descriptor table, which contains a list of all provided ACPI tables. The Differentiated System Description Table (*DSDT*) and the Multiple APIC Description Table (*MADT*) are used to figure out what CPUs are currently installed and what identification number they use. The DSDT contains information in the ACPI machine language (*AML*) byte code. We wrote a tree containing two CPUs and compiled it using the Intel ASL compiler. As ACPI provides a memory mapped interface and the tables mentioned above are read only, we do not have to unmap the memory region. We just fill the mapped memory with appropriate data.

We decided not to implement any form of MPS tables, as they are considered legacy. However under certain circumstances Linux will fall back to these tables. To get around this problem, Linux must be compiled without support for MPS.

The operating system may use information from the BIOS provided memory map to know where the ACPI tables are stored. We added the region in the memory map of the emulated BIOS. However, Linux fails to load the memory map. We traced the bug down to the real mode emulator, which is unable to cope with some optimised memory copy operations issued by Linux. As a first solution, we modified Linux to copy each byte separately instead of using `memcpy`. This allows Linux to correctly load the memory map. After completing our implementation, we figured out that Linux assumes valid default locations if no memory map is supplied.

However, supplying a memory map is highly desirable, otherwise Linux does not know how much memory is installed and defaults to 128 MiB. Hence, if Linux runs without the patch, a memory map must be passed over the kernel command line.

6.3 Implementation Issues

Serial Console

VMkit emulates a PC16550 compatible serial interface which is infinitely fast [San09]. This means, from the guest's perpsective, the outupt FIFO buffer is always empty. When activating two vCPUs, Linux somehow notices this fact and stops writing to the serial line until the serial line receives a keypress. As a workaround, which is necessary for automatic testing, the serial driver of Linux can be patched.

Missing MPS Tables

As Linux tries to parse the MPS tables and we did not implement them, Linux support for them must be turned off at compile time. When not doing so, only one CPU will be recognised. We are not sure why Linux does so and does not prefer the ACPI information.

Timer Issues

VMkit used the deprecated Barrelfish timer service to emulate virtual timers. The timers were replaced by the now recommended deferred events, which plug in into the user level dispatcher. As the kernel has no knowledge of running timers, the timer events can not achieve better accuracy than those of a kernel time slice. The time slice is currently $80ms$ in Barrelfish. But Linux sets the timer to fire each $10ms$. Hence, the provided interrupts occur eight times too slow, making the Linux guest time run eight times too slow. This implies, as Linux uses the APIC timer as time source, the time reported from the Linux cannot be trusted. Further, this may change the Linux performance, as preempting the running application can only happen each $80ms$ instead of $10ms$.

Lock Issues

One source of frustration was seemingly random occurring locks when running with two vCPUs. In the end, we identified two issues.

It is crucial to never touch the VMCB of a running VM. The case we did not notice for a long time happened when the second CPU used the serial console. The serial output code checked if it is necessary to send an interrupt to the CPU. If it is necessary, it sends an interrupt, which is routed through the PIC. As the PIC always sends interrupt to the first core, it was possible that the thread on the second core modified state on the first core, while the first core is in the VM.

After resolving this issue, we still observed some weird behaviour on the second core. It seemed like callback functions, such as Flounder binding callbacks, were executed at random times. We assumed they are only called when we actually dispatch events from the waitset. Under this assumption, we could not identify the problem. After we printed stack traces in each handler we were able to identify the problem: There is a thread running on the second core, the so called slave thread, which dispatches events from the default waitset in an endless loop. This thread is started after a domain is spanned to a secondary core. The purpose of this thread is to receive messages from the monitor to allow creation of threads on the secondary core.

We decided to use locks in all of our handlers, which solves the problem. The cleanest solution would be to modify Barrelfish's slave thread to not use the default waitset.

Performance Issues

After having reached a stable state, we tried to perform some benchmarks and encountered very bad performance. When using two vCPUs, a Linux compilation slowed down by a factor of four compared to execution with *one* vCPU. Once the system has booted, mainly the IPI subsystem of our new code was used. As we were aware that the implementation of the IPIs was not optimal, we measured the number and delays of IPIs sent. There were in fact many and some of them needed almost $80ms$, which is the interrupt period of the hardware timer, to reach their destination. This happened because Barrelfish's default messaging uses polling. And there is no way to poll while the VM is running.

A solution to this problem is to accompany the virtual IPI with a real IPI. When doing so, the real IPI will instantly evict the VM and transfer control to the VMM, which can handle the virtual IPI.

To do this, Barrelfish supports an alternate backend for Flounder: UMP-IPI. As the name suggests, it uses UMP to transmit the payload and signals the receiving core using an IPI. UMP-IPI includes support for x86, but it was broken. After resolving this issue by increasing the size of the notification buffer, it looks like the messages are transferred and indeed an interrupt is triggered on the receiving core.

Although now a real IPI is sent, the delay measured until reception is still high. We are not completely sure why, but as a workaround we tried using the default UMP backend and trigger IPIs separately. This way we were able to observe the desired behaviour: A `VMEXIT` caused by an interrupt accompanied with a virtual IPI Flounder message.

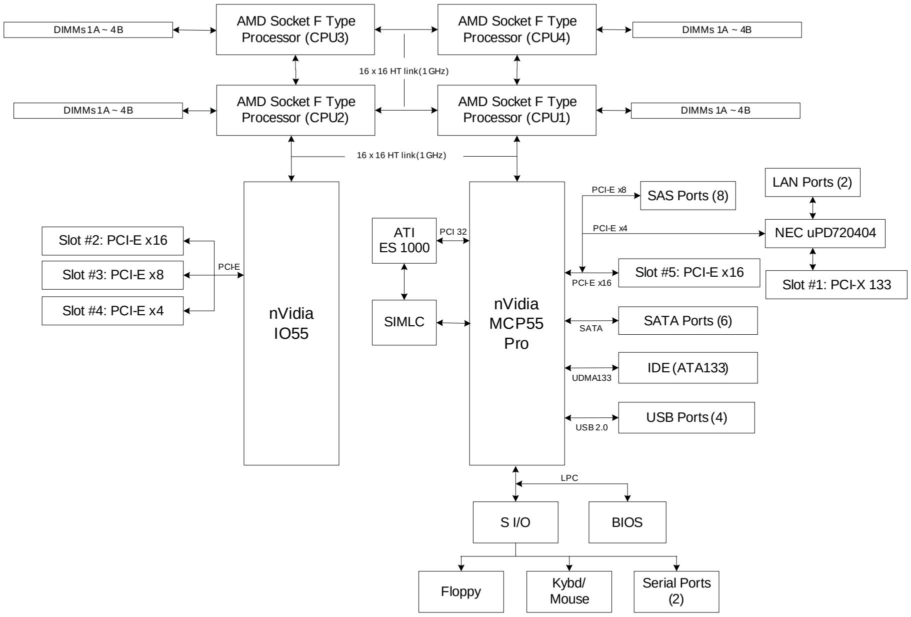

Figure 6: Block diagram of the mainboard.[3]

7 Evaluation

To evaluate the performance of the implementation, we used three different scenarios. The first consists of a one or two CPU VM running under different virtualization systems in a non-contended setting. The main purpose of these benchmarks is to specify overhead of virtualization in general and overhead of our solution compared to other virtualization systems. We decided to use KVM as the reference, because its architecture resembles the VMkit architecture. To get a baseline, we also executed the benchmarks running on native hardware.

In the second scenario, we vary the placement of vCPUs on pCPUs. The results should quantify what effects the NUMA architecture of the machine has on performance. For reference, some experiments were also executed natively.

The third scenario quantifies the effect of scheduling vCPUs concurrently. To get a reproducible worst case scenario, a synthetic background task is introduced which runs non-preemptable on Barrelfish and competes with the VMM for CPU time. The pattern of the time slice the background task receives is varied and effects on the benchmarks running in the VM are observed.

[3] Source: SuperMicro H8QM3-2 mainboard manual

7.1 Test Environment

All benchmarks are running on a 16-core machine. The motherboard has four sockets, each hosting a 2.5 GHz quad core AMD Opteron 8380 processor.

Each core contains 64 KiB data and 64 KiB instruction L1 cache plus 512 KiB shared L2 cache. The cores are connected to 6 MiB L3 cache which is shared among the same socket. As each socket has its own memory controller, the memory is split into four NUMA nodes. Each memory controller hosts 4 GiB of memory, totalling in 16 GiB of memory. To access a remote memory location, the CPUs make use of the inter-socket HyperTransport interconnect. As seen in Figure 6, the socket interconnect has a ring topology. For communication between cores on the same socket, a crossbar switch is used. Hence, the cores of a single package should have indistinguishable behaviour regarding memory access.

HyperTransport is not only used to access remote memory, but also to transport interrupts and handles I/O. To do so, two HyperTransport I/O controllers are on the mainboard. One connected to CPU 1 which handles almost all hardware and one connected to CPU 2 which adds more PCI Express slots.

The VM receives three GiB of memory. Whenever a benchmark is running natively, the amount of RAM Linux uses is limited using the `mem` kernel parameter. To limit the number of CPUs, we used the `maxcpu` Linux kernel command line argument.

When KVM is used, the host platform is an Ubuntu 12.04, running Kernel 3.2.0 and QEMU 1.0. CPUs are always pinned to one core, using the `vcpupin` option.

Since there is no reliable timing source in the VMkit provided VM, we use the time stamp counter of the CPU. The `rdtsc` instruction is executed from within the VM, while the VM is configured not to intercept the `rdtsc` instruction. When running native or KVM benchmarks, we use the `gettimeofday` function as a timing source. Using this timing source, we measure the wall time.

As VMkit does not provide any virtual disk, the benchmark system runs completly off the initial ramdisk. The userspace is based on Debian Squeeze 6 and uses a custom built Linux 2.6.32 kernel. Building a custom kernel allowed us to ensure that it does not make use of paravirtual optimisations. It also allows to turn off parsing of the MPS (see Section 6.3) tables, which confuses Linux if not present. To get comparable results when running natively or in KVM, the system also runs completly off the ramdisk.

7.2 Benchmark Description

Microbenchmark: World switch

The trap and emulate approach introduces an overhead compared to native execution, where privileged instructions are directly executed in hardware. The amount of virtualization overhead is heavily influenced by the time spent in the VMM, as with using hardware extensions, normal guest execution works with almost native speed. The time spent in the VMM is determined by the number of VMEXITs and how long it takes to process these. A lower time bound for processing a VMEXIT is the time it takes to go from the VM to the VMM and back. In this case, the VMM does not perform any action at all, except returning the control to the VM.

To perform this measurement, the vmmcall instruction is used. This is a special instruction that can be used for hypervisor calls from the guest.

Neither KVM nor VMkit perform any action except advancing the instruction pointer and returning into the VM.

Microbenchmark: Forkbench

Forkbench is a microbenchmark in which a parent process forks of a child and waits until its completion. The child does not perform any work but immediately exits. As this happens rather quick, one run of the benchmark consists of 1000 fork operations. To gather statistical data, 1000 of these runs are executed.

The primary motivation for including this benchmark was that we observed performance problems in our implementation, which were most likely related to the IPI latency. To verify progress, a microbenchmark that sends an IPI had to be found. While running the Linux compilation, we found that during process creation an IPI was triggered. Hence, we created this benchmark which does nothing else than create processes.

Calling fork has a number of consequences. The child process sees a copy of the memory from the parent process. Linux implements this copy operation efficient using copy-on-write. When fork is called, the page table of the process is copied and the pages must be remapped read only. This implies a need to flush the Translation Lookaside Buffer (*TLB*). As other threads using the same memory map may be running on another CPU, Linux must make sure that also these CPUs flush their TLBs.

Linux ensures this by sending an IPI to the affected CPUs and prepares a shared piece of memory, where each CPU writes into after it has completed the operation. The CPU which initiated the process polls the memory location until all CPUs have completed the flush. During this operation, preemption is disabled. This implies, the CPU executing the fork operation will not make any progress until all other CPU have reacted to its request.

After the TLB flush, the TLB must be refilled. On x86, this happens in hardware by walking the page table until a mapping is found. The cost for walking the page table is different in a virtualized setting, as there are two nested page tables involved. See Section 2.3.

Microbenchmark: Hackbench

Hackbench is small benchmark created to measure performance of the Linux scheduler. It is part of the rt-tests package[4]. It creates 10 communication networks. Each communication network consists of 20 senders and 20 receivers, all of them being processes. Sender and receiver are connected using a local

[4] http://git.kernel.org/cgit/linux/kernel/git/clrkwllms/rt-tests.git/

socket pair (created using `socketpair()`). After all child processes, sender and receiver, have started, all sender start sending messages to all receivers of the same group. Sending a message is done by a blocking write to the sender's end of the socketpair. The message size is 100 bytes and each sender sends 100 messages to each receiver. The sender sends the first message to all its receivers, it then continues with sending the next message without waiting on the receiver. The receiver process performs a blocking read on its receiving socket. Hence, during one run there are $10 \times 20 \times 100 \times 20 = 400'000$ messages exchanged. To gather statistical data, the process is repeated 250 times.

The buffer size for sockets[5] is 112 KiB. As we are interested in the behaviour where synchronization happens between cores, we let the benchmark run twice. Once we set the CPU affinity of all sender processes to CPU zero and all receiver to CPU one. In the other case, we do not set any CPU affinity and let the Linux scheduler decide where to put the processes.

Care was taken that only the duration of the actual sending and receiving of the messages is counted. Especially the rather costly fork operation is excluded from measurement time.

Microbenchmark: Rambench

The aim of this benchmark is to measure the memory bandwidth and latency. To do so, 512 MiB are allocated and a shuffled linked list is created in it. To reduce cache effects, each element has the size of a cache line. Then the time to traverse the list is measured. To advance to the next list cell, the previous one must be fetched. As the next location is random, the chance of it being in cache is small. Hence, most of the time, the CPU has to fetch the memory location from main memory. So the cost of traversing this list is dominated by the *latency* of the main memory.

A second test is performed, where the list is not shuffled, but linearly arranged. As the access pattern is highly regular, the cache can be used to satisfy many memory requests. At some point, the cache has to be filled anyway, so the dominant factor is the memory *bandwidth.*

When two CPUs are available, the benchmark is run twice, once pinned to the first CPU and once pinned to the second CPU.

Case study: Linux compilation

Software compilation is a widely accepted benchmark. It exercises the file system and the CPU. In our case, the file system resides on a ramdisk, reducing the number of `VMEXIT`s compared to a virtual hard disk. As a compile subject we have chosen Linux 3.7. To reduce the time needed for benchmarking, we configured Linux with `make allnoconfig` which sets all options to no. To make use of the additional cores, we set the jobs option of make to four to run multiple processes in parallel.

[5] read from `/proc/sys/net/core/wmem_default`

During compilation, mostly the compiler is running. Compared to the microbenchmarks, the process running time is higher. Also, the need of synchronisation is rather low. The controlling make process waits until a process finishes and starts a new one if possible. Some parts of the compilation consist of Bash scripts. Bash scripts usually works by combining the work of many smaller programs, hence many short running processes are created during these stage.

As discussed in 7.2, process creation requires the CPUs to synchronise.

7.3 Scenario 1: Performance in a non-contended setting

In this section we compare the performance of our implementation to native execution and to KVM, using single and dual core VMs. No other significant processes were running. All virtual CPUs were placed on the same NUMA node.

We found that VMkit performs as well as KVM in both, one and two CPU settings. The Forkbench on two CPUs even runs much slower on KVM than on VMkit. The Rambench illustrates the higher cost of TLB misses when running the VM. In the cases of randomised access, more TLB misses are generated and hence the overhead compared to native execution is much higher than in the sequential access case.

7.3.1 Microbenchmark: World switch

System	`vmmcall` latency [cycles]
VMkit (old)	7538 ± 95
VMkit (new)	4572 ± 159
KVM	2165 ± 573

Table 1: `vmmcall` microbenchmark.

As seen in Table 1, the latency is decreased with the new VMkit design.

The old VMkit design uses two domains, one for running the VM and one for runing the VMM. Hence, to process a single `VMEXIT`, the system must switch out of the VM into the kernel, where a message is sent to the VMM domain. To handle the message, a switch to the VMM domain is necessary. After handling the `VMEXIT` this process happens again in reverse order.

The redesigned VMkit uses a simpler approach. Once the VM exits, Barrelfish's kernel must simply return from the syscall that requested the VM operation. After handling the `VMEXIT`, the VM is re-entered by using the same syscall again. Thanks to the tagged TLB, no TLB flush and no page table change is necessary.

The new architecture requires less context switches, which explains the observed improvement in latency. The operation still takes more cycles than on KVM, which handles the call directly in the kernel.

7.3.2 Microbenchmark: Forkbench

To verify that the benchmark actually sends IPIs, we counted the number of IPIs in the VMM and checked the output of `/proc/interrupts`.

System	1 CPU [ms]	2 CPUs [ms]
Native	56.3 ± 0.2	92.2 ± 0.4
KVM	105.6 ± 0.2	530.7 ± 8.6
VMkit	112.3 ± 0.7	196.2 ± 0.9

Table 2: Forkbench microbenchmark.

Table 2 shows the results of the Forkbench. The first thing to note is that this benchmark takes actually *more* time when running on two cores. In the single core case, there is no need to notify the other cores when the address mapping has changed, so no IPIs are sent and no synchronization is necessary.

The program runs much slower in a virtualized environment, even using only one CPU. As the benchmark creates a lot of processes and flushes the TLB all the time, a significant part of the execution time is spent in the hardware traversal of the page tables. Both KVM and VMkit use nested paging, which increases the cost of a TLB miss.

Once two CPUs are used, the operating systems has to synchronise the two cores when creating a new process. On native hardware, the synchronisation adds approximately $36ms$ of overhead. VMkit gets slower by $84ms$. The additional overhead comes either from the virtualized IPI, which has a higher latency, or from the fact that the page tables must be traversed from both cores.

KVM struggles with this benchmark when running two CPUs. We have not investigated the cause for this slowdown. Candidates are a suboptimal implementation of IPIs or possibly running background tasks.

7.3.3 Microbenchmark: Hackbench

System	1 CPU Hackbench [s]	2 CPU Hackbench [s]	2 CPU Hackbench with affinity [s]
Native	0.60 ± 0.02	0.38 ± 0.01	0.63 ± 0.01
KVM	0.77 ± 0.02	0.51 ± 0.03	0.69 ± 0.04
VMkit	0.63 ± 0.06	0.69 ± 0.09	1.03 ± 0.11

Table 3: Hackbench under different systems.

The results of the Hackbench are shown in Table 3. When going from one CPU to two without explicitly setting the CPU affinity, native and KVM executions are able to profit from the second CPU. In contrast, the execution on VMkit does not speed up.

When setting the affinity explicitly, all systems slow down. In this case, the scheduler is not able to balance the processes equally among the two cores, leading to higher execution times. Also the information that a communication socket is readable must be transferred cross core. This introduces some synchronisation overhead.

7.3.4 Microbenchmark: Rambench

System	Sequential		Randomised	
	[s]	Relative	[s]	Relative
Native	0.24 ± 0.0001	1.00	1.26 ± 0.0023	1.00
KVM	0.27 ± 0.0001	1.13	1.83 ± 0.0015	1.45
VMkit	0.26 ± 0.0002	1.08	1.82 ± 0.0017	1.44

Table 4: Rambench running under different systems using one CPU. Relative factors are based on native execution.

Table 4 shows the results. As the placement benchmarks (see Section 7.4) reveal that the core numbering under Barrelfish is somehow wrong, we picked the results where VMkit is running on core 4. Also two CPU results are omitted, because they are exactly the same as when running on one CPU.

As no `VMEXIT` should be triggered during the execution, both virtualization systems introduce the same overhead. While in the sequential case, the overhead is around $\frac{0.26}{0.24} \approx 1.08$, in the randomised case it is higher $\frac{1.82}{1.26} \approx 1.44$. The randomised benchmark causes more TLB misses and hence has to walk the page table much more often. As explained in Section 2.3, the cost of a TLB miss is higher when running a virtualized system.

7.3.5 Case study: Linux kernel compilation

System	1 CPU		2 CPU	
	[s]	Relative	[s]	Relative
Native	138	1.00	70	1.00
KVM	166	1.20	84	1.20
VMkit	157	1.14	81	1.16

Table 5: Linux compilation benchmark. Relative factors are based on native execution.

Table 5 shows the results of the kernel compilation. The benchmark does not perform much synchronisation, so it can take advantage of the additional CPU. However, some parts of building Linux involve Bash scripts, which create quite a lot of processes. As mentioned earlier, process creation requires the guest kernel to synchronise.

The compilation is quite memory intensive, we suspect this is the reason why KVM performs worse than VMkit. VMkit allocates and maps all memory before boot, while KVM behaves like a normal Linux process. The host Linux catches the pagefault, installs a mapping on demand and continues the VM.

7.4 Scenario 2: Effects of vCPU placement

As the test machine offers four NUMA nodes, we wanted to quantify the performance impact when the allocation of vCPUs to pCPUs is varied along the NUMA nodes. Memory was allocated always from NUMA node zero. To see effects of a single core VM when placed on different nodes, we let all benchmarks with a single core VM run on each core.

We found that the Rambench, which specifically exercises the main memory, shows significant effects when executed on a remote NUMA node. Also we observed some inconsistencies concerning the NUMA node numbering and associated memory regions under Barrelfish. The ring topology of the interconnect that shows up in the Rambench is not visible on the other benchmarks. We do not have a full explanation for the effects observed.

7.4.1 Single core placement

Placement (NUMA node)	Linux compile [s]	Rambench sequential [s]	Rambench random [s]
0	158.67	0.31 ± 0.0002	1.93 ± 0.0019
1	156.77	0.26 ± 0.0002	1.81 ± 0.0017
2	160.45	0.37 ± 0.0002	2.30 ± 0.0023
3	159.12	0.33 ± 0.0002	1.96 ± 0.0019

Table 6: Placement benchmark when running with one CPU under VMkit.

Table 6 shows the results, which were grouped by NUMA node. When looking at the results, something with the numbering of the NUMA nodes seems wrong. As the memory is from node zero, we expected the best performance when running on a CPU from node zero. To verify the numbering, we let the Rambench run on native Linux. To force an environment as similar as possible to the first experiment, the Linux NUMA allocator was used to get memory from node zero and the CPU affinity was set to the desired CPU.

As Table 7 shows, contrary to the results from VMkit, the results from native execution are as expected. We verified that VMkit on Barrelfish is really using memory from node zero by printing the physical guest memory base address and size and compare it to the NUMA information. The NUMA information from Barrelfish was cross validated to the information Linux provides[6]. Barrelfish

[6] The information from Linux was gathered by specifying acpi=debug as kernel command line parameter, then ACPI dumps are printed on boot.

Placement (NUMA node)	Rambench sequential [s]	Rambench random [s]
0	0.24 ± 0.0001	1.34 ± 0.0013
1	0.29 ± 0.0001	1.44 ± 0.0015
2	0.30 ± 0.0001	1.46 ± 0.0015
3	0.34 ± 0.0001	1.78 ± 0.0019

Table 7: Rambench when running with one CPU native. Placement is set via CPU affinity and the NUMA allocator was used to get memory from node zero.

uses the APIC id by default to print the CPU number. The APIC id may be modified by the operating system. To ensure this did not happen, we verified the CPU number executing the VM is correct by using the information provided by the `cpuid` instruction, which cannot be changed. None of these checks revealed the cause of the problem.

Due to the ring topology of the socket interconnect (see Section 7.1), we expect three categories of results. The fastest execution should be observed when the running on the CPU where the memory is directly connected. The second fastest results should be observed when the memory is one hop away. Finally on one node the memory is two hops away, there the slowest execution should be encountered. The native execution under Linux (see Table 7) shows the expected behaviour. Also the execution in VMkit shows this behaviour when we ignore the wrong numbering of NUMA nodes: Node 0 and 2 are two hops away, node 1 is directly connected to the memory and node 3 is two hops away.

In line with the variations of the Rambench is a slight variation of the Linux compilation times. The effect is much smaller though, as the Linux compilation is not completely memory bound.

Placement (NUMA node)	Forkbench [ms]	Hackbench [s]
0	108.9 ± 2.22	0.70 ± 0.07
1	108.6 ± 0.88	0.72 ± 0.07
2	108.5 ± 1.00	0.75 ± 0.03
3	113.3 ± 2.84	0.80 ± 0.08

Table 8: (Continued) Single core placement.

As seen in Table 8, Forkbench and Hackbench slow down slightly when executed on NUMA node 3. We are not sure how to intepret this, as possible NUMA effects should also appear on other nodes.

As we have seen, the Forkbench forces quite a lot of page table walks. As the page tables also reside on memory from NUMA node zero, one might expect a slowdown when running on a core which is not directly connected to the memory. However, as the working set is small and the memory is exclusively used by this core, the caches should be able to hide the effect.

Also as the Hackbench has a small working set, we do not expect any variation. Due to the high variance of the benchmark, no final conclusion can be drawn from the observed numbers.

7.4.2 Dual core placement

The next step is to activate two CPUs and let them run on different NUMA nodes. We expect performance to degrade when not running all of the vCPUs on the same NUMA node, due to increased IPI latency and increased memory access latency. We kept the first core fixed and varied the second one. As the single core placement scenario has shown that core 4 is the fastest, core 4 was chosen as the fixed CPU. To ensure, that inverting the placement of the cores does not have any effect, we executed one run also with inverted placement.

Placement (NUMA node)		Linux compile	Forkbench	Hackbench with affinity
CPU 0	CPU 1	[s]	[ms]	[s]
0	1	81.32	196.76 ± 1.6	1.04 ± 0.10
1	0	81.28	198.27 ± 2.3	1.06 ± 0.10
1	1	81.63	199.10 ± 2.4	1.02 ± 0.08
1	2	82.01	212.44 ± 4.4	1.29 ± 0.20
1	3	81.74	213.27 ± 1.6	1.24 ± 0.16

Table 9: Dual core placement.

Table 9 shows the results. The Hackbench without CPU affinity set does not show any relevant effect and is not listed here. Also the Rambench executions are not listed, as they show exactly the same behaviour as under the single core VM.

The Hackbench and Forkbench both show slightly worse perfomance when executed on NUMA nodes 1-2 and 1-3. It is not clear why the execution on nodes 1-0 does not slow down as well. Maybe the chip internal interconnect does not provide the same characteristics for the HyperTransport access, hence NUMA node zero can be reached faster from core 4. Or, there is another issue with node numbering, but since we run the experiment on all cores (see Section 9.2 for details) and aggregated the results here, we should have spotted such a discrepancy.

The Forkbench slowdown might appear because of the increased IPI latency. Also the same page table has to be traversed from two different sockets, hence there might be some penalty due to remote memory access. On fork, the page table is modified by Linux for copy-on-write, hence the cache may not fully be able to hide these effects.

It is likely that during the Hackbench with affinity, the kernel frequently needs to access shared data structures. Such as when a sender sends a message, the remote (potentially blocked) receiver process must be inserted into the remote run-queue. Also the messages which are written on one core and read from the other must be transferred over the interconnect. Hence, the latency of accessing these shared data structures increases when executed on two different NUMA nodes and slows down the execution.

As seen in Table 9, inverting the placement does not have any effect.

7.5 Scenario 3: Scheduling strategy

As discussed in the literature review section, performance may degrade more than expected due to asynchronous vCPU scheduling. To get an impression of the effect we performed benchmarks, where the time slice of the vCPUs are shifted by various amounts.

We found the results of the Linux kernel compilation and the Forkbench in line with our expectations. The Linux kernel compilation is CPU bound and does not perform much synchronisation, hence it shows equal performance when time slices are offset. The Forkbench in contrast, requires synchronisation all the time and is therefore highly sensible to offset time slices. A surprise was the behaviour of the Hackbench with affinity set. We expected worse behaviour under offset time slices due to increased communication latency between the senders and receivers. But the experiments revealed *better* performance under offset time slices.

7.5.1 Method

To ensure that the CPUs are not scheduled out of our control, we modified Barrelfish's timer interrupt handler. The timer interrupt is responsible for preempting the currently running domain. The handler of this interrupt increments the kernel time, calls the scheduler to let it make a scheduling decision and executes the selected domain. When the running process yields or another (non timer) interrupt arrives, the scheduler is also invoked.

Hence to make absolutely sure that the vCPU does not get scheduled, we let the interrupt handler spin for a given amount of time until it is allowed to execute a domain. During this spinning, the CPU remains in a non-preemptable state and nothing unexpected, such as a yield, can happen which would lead to scheduling of the vCPU. This way, we hope to get reproducible, worst-case results. A drawback of this approach is, that it does not necessarily matches a real world scenario. In a real world scenario, the scheduler could upon reception of a virtual IPI designated to the VM, decide to schedule the VM with priority and let it handle the interrupt.

On each timer interrupt, the kernel alternates between two delays. Either in the *synchronous* case, the two cores running the VM choose the same delay at the same time or in the *asynchronous* case, they always choose offset delays. Figure 7 illustrates the different delays used in this section.

The hardware timer fires every $80ms$, hence the spinning is activated each 80ms. We have chosen two different delay pairs $15ms/30ms$ and $5ms/40ms$.

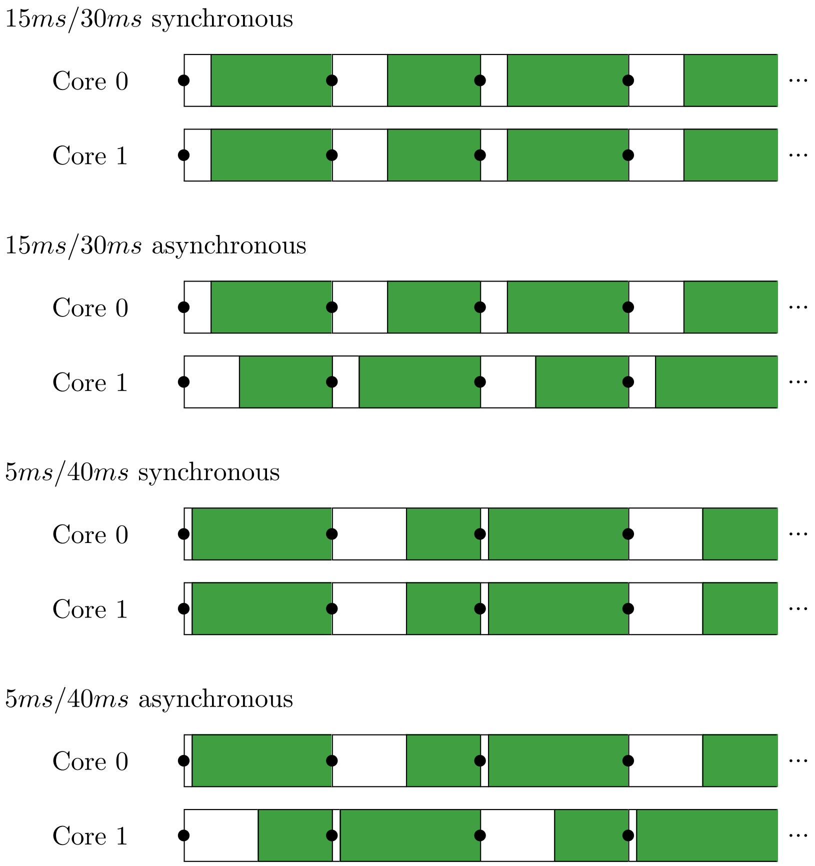
15ms/30ms synchronous
Core 0
Core 1
15ms/30ms asynchronous
Core 0
Core 1
5ms/40ms synchronous
Core 0
Core 1
5ms/40ms asynchronous
Core 0
Core 1

Figure 7: The different scheduling patterns. Black dots indicate timer interrupts. White areas indicate spinning inside the kernel, green areas the VM.

The numbers were chosen so that the amount of concurrent VM execution differs, while keeping the total time each CPU gets constant. During all benchmarks, each vCPU of the VM runs during $1 - \frac{15ms+30ms}{2\cdot 80ms} = 1 - \frac{5ms+40ms}{2\cdot 80ms} \approx 71.8\%$ of total time. Hence, the expected slowdown compared to the non-contended setting is $\frac{1}{0.718} \approx 1.39$.

7.5.2 Results

The results are shown in Table 10 and 11. As seen in Table 10, the Rambench does not show any variations when running under offset time slices. This is expected, as the Rambench is strictly single threaded.

The Hackbench without affinity shows no variations, but the slow down factor of ≈ 1.26 is smaller than the expected factor of 1.39. We do not have an explanation for the small slow down. The absence of varying results under

Pattern	Rambench sequential [s]	Rambench random [s]	Hackbench without affinity [s]	Rel.
$15ms/30ms$ sync.	0.44 ± 0.0127	2.64 ± 0.0111	0.87 ± 0.10	1.28
$15ms/30ms$ async.	0.43 ± 0.0119	2.67 ± 0.0112	0.86 ± 0.10	1.26
$5ms/40ms$ sync.	0.43 ± 0.0180	2.65 ± 0.0193	0.86 ± 0.11	1.26
$5ms/40ms$ async.	0.43 ± 0.0182	2.65 ± 0.0186	0.88 ± 0.09	1.29

Table 10: Contended setting benchmark.

Pattern	Linux compile [s]	Rel.	Forkbench [ms]	Rel.	Hackbench with affinity [s]	Rel.
$15ms/30ms$ sync.	112.11	1.38	273.7 ± 13	1.39	1.30 ± 0.12	1.26
$15ms/30ms$ async.	112.62	1.39	320.7 ± 10	1.63	1.19 ± 0.13	1.15
$5ms/40ms$ sync.	112.07	1.38	272.7 ± 18	1.39	1.35 ± 0.13	1.30
$5ms/40ms$ async.	113.66	1.40	397.8 ± 13	2.03	1.19 ± 0.13	1.15

Table 11: (Continued) Contended setting benchmark. Relative factors are based on non-contended VMkit execution.

different time slice allocations indicates that the Linux scheduler is always able to schedule one of the communication processes and that these processes can indeed make progress.

The Linux compilation slows down by a factor of ≈ 1.39 as expected. As the benchmark is likely to be CPU bound and the need for synchronisation is minimal, the moving of the time slices does not have any effect.

In the two synchronous cases, Forkbench slows down by a factor of ≈ 1.39 as predicted. However, as explained, a fork operation can only complete as soon as every CPU has responded. Hence the Forkbench is only able to make progress when both CPUs are running. To verify this assumption, we calculate the slowdown factor by using only the concurrent running time.

In the asynchronous case with $15ms$ and $30ms$ delay, the fraction of time where both CPUs are running is $1 - \frac{30ms}{80ms} \approx 62.5\%$. Hence the execution time slows down by a factor of $\frac{1}{0.625} \approx 1.60$.

Correspondingly in the case with $5ms$ and $40ms$ delay, the fraction is $1 - \frac{40ms}{80ms} \approx 50\%$ and the execution slows down by a factor of $\frac{1}{0.5} \approx 2$.

The calculated factors match the measured ones. Figure 8 also gives an indication that Forkbench exclusively makes progress while both CPUs are running.

Contrary to the Forkbench, the execution time of the Hackbench with affinity actually *decreases* when the CPUs are being scheduled asynchronous. A program which has a large sequential fraction should show such behaviour, as only one execution thread makes progress at a given point in time. It is not clear where the Hackbench contains a significant sequential part. As it does not

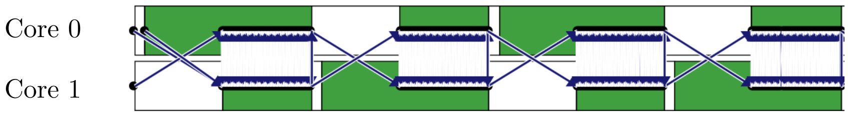

Figure 8: Forkbench running with asynchronous timeslice. Green is the VM. Arrows indicate exchanged IPI messages.

have any explicit locks in the benchmark code, the reason must be somewhere in the kernel or library functions. One suspect is the FIFO queue by which the two sockets are connected. If the queue is implemented naively, only one reader or writer may modify the queue. In general Linux is implemented quite efficient and the number of queues is pretty large. Hence we do not think this is the bottleneck.

Another suspect are the scheduling queues. During the benchmark, processes frequently block on read and have to be taken of the run-queue. Also, the writer processes write tiny amounts of data all the time; on every write the (potentially) blocked reader has to be inserted into the run-queue. Access to these scheduler queues must most likely be synchronised.

However, it does not get faster from the asynchronous $15ms/30ms$ to the asynchronous $5ms/40ms$. This indicates that only specific amount of work can be completed without the other core responding. We suspect that after a short time, all FIFO buffers are full and the writer processes also start to block. The same could happen when the sender have sent all their messages and terminate. Then the system must wait on the other CPU, which runs the receivers, until it can finish.

8 Conclusion

Our aim was to find a suitable implementation of a VMM that supports multiple vCPUs on a multikernel.

We have done so by extending VMkit, the VMM of Barrelfish, to provide two CPUs to a guest operating system. According to our goals, we paid attention to implement VMkit efficient. In the end, our implementation reached performance comparable to other virtualization systems. Our solution is able to run an unmodified Linux by making use of the hardware extensions available on modern x86 CPUs. The guest Linux uses the newly implemented hardware. The APIC, for instance, is used to deliver IPIs and also as timing source. For CPU discovery, ACPI is used.

The design of the VMM has proven to work, although in retrospect, we would probably try to implement the VMM as two separate domains, sharing only the address space of the guest. From our point of view, this seems more compatible with Barrelfish's approach of explicit sharing than memory sharing. Using this two domain approach, we would also avoid some unexpected behaviour of Barrelfish that only occurs with one domain spanning multiple cores. With hindsight, the decision to leave the PIC in the system was most likely wrong. We think that making the PIC and the APIC work together correctly took more time than just implementing an I/O APIC and discarding the PIC. In order to prevent non privileged processes from compromising the system, a coarse grained VMM capability is introduced.

We identify that a polling based messaging protocol is unsuitable for rapid transmission of virtual IPIs. Low latency virtual IPIs have shown to be essential to achieve good performance. To accomplish a satisfactory delivery latency, sending an accompanying physical IPI is required.

To evaluate performance, a number of microbenchmarks are introduced and some of them revealed interesting effects. The increased cost of TLB misses, due to nested paging, could be observed multiple times. KVM shows a considerable slowdown when running the Forkbench on a two CPU virtual machine.

Furthermore, we looked at how core placement on different NUMA nodes affects performance. As expected, a microbenchmark exercising RAM bandwidth and latency shows significant slowdown. However, all other benchmarks did not show such large effects. So far, we found no exhaustive explanation for those smaller effects. No significant impact caused by increased IPI latency between different NUMA nodes could be observed. In general, the test machine seems to do a good job in hiding NUMA effects. As virtualization adds another layer of complexity, we do not think that this is a good method to study NUMA behaviour.

The effect of gang scheduling has been investigated. Benchmarks, that synchronise often, suffer from performance degradation, while independent workloads are not slowing down when scheduled asynchronously. A certain class of workloads, however, shows better performance when the vCPUs are scheduled asynchronously. Surprisingly, one of our microbenchmarks, the Hackbench with affinity set, belongs to this class. We have not fully investigated what exactly causes this behaviour, but we suspect some kernel level synchronisation introduces a sequential part. The classification if a program performs better or worse under gang scheduling is hard. One does not only need to understand the behaviour of the application, but also how different operations in the Linux kernel are implemented.

8.1 Future Work

The extended VMkit is usable to reproduce some effects of different scheduling strategies. The lack of supported hardware, especially I/O devices, makes it impossible to run more complex benchmarks. Adding more virtual hardware devices would allow us to run more interesting benchmarks, for example a web server. With this support, the *latency* of replying to a request could be measured. I/O limited benchmarks exercise the scheduler in a different way than the benchmarks we used. The workloads we used present an always runnable VMM process to the scheduler.

Our benchmark results have shown some effects we do not fully understand yet. Especially some questions concerning the placement of vCPUs on different NUMA nodes remain open. It is disputable to use a VM to answer such questions, as it introduces a lot of complexity and indirections, which can easily shadow NUMA effects.

As mentioned before there are other design alternatives for a VMM on top of a multikernel. It would be interesting to restructure the VMM into two (or more) domains that communicate exclusively through messages. This could result in insights how to implement a VMM on a non shared memory system.

In order to enable migration of running Barrelfish domains, one might put specific subsystems of Barrelfish inside a VM. To do so, messaging channels must be bridged into the VM and after migration one has to pay attention to recreate the bindings correctly.

As the classification, whether a program slows down or speeds up under gang scheduling, is hard, one might try to alter the guest operating system to make its behaviour more predictable. For example a process that calls fork still blocks until all concerning TLBs are flushed (not to do so would be incorrect). The hope is, that this fork operation will not block the whole operating system.

Maybe using a Linux kernel with the real-time patch[7] improves the situation, since the patched kernel remains preemptable in more situations. The usage of a multikernel based operating system as guest may also help. The communication is made explicit and the operating system is aware of message latency, that could solve some of these issues as a side effect. However, this does not help to solve any user space synchronisation issues.

Literature suggests that many effects of scheduling are not observable with two cores or that these effects get more distinct. VMkit is already designed to support more than two CPUs, nevertheless it has never been tested and most likely some little modifications will be necessary.

[7] `https://rt.wiki.kernel.org/`

Paravirtualization is known to improve performance. When the guest makes use of paravirtualization, it would be interesting to verify if KVM shows the same bad performance on the Forkbench as in our evaluation.

9 Appendix

9.1 References

[APM] AMD64 Architecture Programmer's Manual, Volume 2: System programming.

[BBD$^+$09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[BXDL12] Chang S Bae, Lei Xia, Peter Dinda, and John Lange. Dynamic adaptive virtual core mapping to improve power, energy, and performance in multi-socket multicores. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 247–258. ACM, 2012.

[BXL10] Yuebin Bai, Cong Xu, and Zhi Li. Task-aware based co-scheduling for virtual machine system. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 181–188. ACM, 2010.

[CGV07] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *Performance Evaluation Review*, 35(2):42, 2007.

[Fri08] Thomas Friebel. Preventing guests from spinning around, June 2008. Xen Summit Boston 2008.

[GND$^+$07] Sriram Govindan, Arjun R Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136. ACM, 2007.

[Gol73] Robert P Goldberg. Architectural principles for virtual computer systems. Technical report, DTIC Document, 1973.

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[San09] Raffaele Sandrini. VMkit: A lightweight hypervisor library for Barrelfish, 2009.

[SHB09] Gabriel Southern, David Hwang, and R Barnes. SMP virtualization performance evaluation, 2009.

[SK11] Orathai Sukwong and Hyong S Kim. Is co-scheduling too expensive for SMP VMs? In *Proceedings of the sixth conference on Computer systems*, pages 257–272. ACM, 2011.

[TDY11] Jia Tian, Yuyang Du, and Hongliang Yu. Characterizing SMP Virtual Machine Scheduling in Virtualization Environment. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 402–408. IEEE, 2011.

[ULSD04] 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, 2004.

[VMwa] VMware. Co-scheduling SMP VMs in VMware ESX Server. Technical report.

[VMwb] VMware. VMware® vSphere: The CPU Scheduler in VMware ESX® 4.1. Technical report.

[WWLL09] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120. ACM, 2009.

9.2 Raw Benchmark Results

CPU	Linux compile	Rambench Sequential	Rambench Randomised
	[s]	[s]	[s]
0	158.58	0.31 ± 0.0002	1.95 ± 0.0019
1	159.11	0.31 ± 0.0002	1.93 ± 0.0018
2	158.03	0.31 ± 0.0003	1.92 ± 0.0020
3	158.95	0.32 ± 0.0001	1.92 ± 0.0018
4	156.35	0.26 ± 0.0002	1.82 ± 0.0017
5	156.62	0.26 ± 0.0002	1.82 ± 0.0018
6	156.89	0.26 ± 0.0002	1.81 ± 0.0016
7	157.22	0.26 ± 0.0001	1.80 ± 0.0016
8	160.60	0.37 ± 0.0001	2.31 ± 0.0023
9	160.28	0.37 ± 0.0002	2.30 ± 0.0023
10	160.30	0.37 ± 0.0003	2.31 ± 0.0021
11	160.61	0.37 ± 0.0002	2.30 ± 0.0025
12	158.74	0.32 ± 0.0002	1.96 ± 0.0018
13	159.52	0.32 ± 0.0002	1.97 ± 0.0021
14	158.98	0.33 ± 0.0003	1.96 ± 0.0018
15	159.27	0.33 ± 0.0002	1.96 ± 0.0020

CPU	Forkbench	Hackbench
	[ms]	[s]
0	107.1448 ± 0.7942	0.72 ± 0.0474
1	109.0601 ± 0.8732	0.73 ± 0.0476
2	106.9935 ± 0.7525	0.72 ± 0.0479
3	112.3174 ± 0.7074	0.63 ± 0.0577
4	108.6995 ± 0.8337	0.74 ± 0.0177
5	108.5936 ± 0.7415	0.76 ± 0.0581
6	109.0956 ± 0.7719	0.76 ± 0.0572
7	107.9625 ± 0.7516	0.64 ± 0.0622
8	107.7536 ± 0.8398	0.75 ± 0.0299
9	109.0212 ± 0.8361	0.74 ± 0.0259
10	108.6087 ± 0.8426	0.74 ± 0.0267
11	108.3094 ± 0.8397	0.75 ± 0.0297
12	115.2143 ± 0.7538	0.84 ± 0.0720
13	114.7233 ± 0.7418	0.74 ± 0.0761
14	108.4072 ± 0.8211	0.84 ± 0.0690
15	114.0728 ± 0.7417	0.78 ± 0.0527

Table 12: Single core placement on all available cores using VMkit. The shading indicates the NUMA node.