Diplomarbeit

Efficient Virtualization on Hardware with Limited Virtualization Support

Jan Christoph Nordholz

2. Mai 2011

Technische Universität Berlin
Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Professur Security in Telecommunications

Betreuender Hochschullehrer: Prof. Dr. Jean-Pierre Seifert
Betreuender Mitarbeiter: Dipl.-Inf. Michael Peter

Erklärung

Die selbständige und eigenhändige Anfertigung versichert an Eides statt

Berlin, den 2. Mai 2011

Jan Christoph Nordholz

Abstract

After decades of a rather shadowy existence in the mainfraime niche, virtualization has found widespread adoption in the commodity market in the recent years. The underlying reason for this resurgence is that virtualization allows for a rate of innovation that traditional operating systems have failed to deliver. The range of applications is quite varied and extends from system management over security to aiding development and debugging. At the same time the processing power of embedded devices has increased rapidly. While devices of the last generation came with a very restricted amount of resources, contemporary products easily equal the power that desktop computers had about five years ago, both in terms of CPUs and RAM. This newfound power opens up the possibility to use virtualization even on embedded devices, where its virtues are not less desired.

Before the advent of hardware support for virtualization OS kernels had to be adapted to run under the control of another OS, either in advance or just in time. However both technologies suffered from significant performance loss due to duplicate bookkeeping of task structures, virtual memory mappings etc. The introduction of basic hardware support enabled developers to obviate the work of guest kernel adoption, but the management of virtual memory on behalf of the guest remained a cause for considerable slowdown.

While full virtualization support has become readily available on commodity desktop platforms like Intel's x86, most embedded processors are still either completely incapable of virtualization or only support basic register virtualization. As OS kernels often have to be tailored to the specific SOC they are intended to run on, it seems promising to enhance a basic virtualization setup by making additional modifications to the guest OS in order to assist the host in the management of virtual memory.

This thesis therefore develops and evaluates several software implementations for memory virtualization for use on processors without corresponding hardware support. After creating a basic vTLB implementation, guest-to-host interaction is iteratively extended to create more optimized solutions, e. g. address space caching, support for global (context-switch-persistent) pages and optimizations involving the handling of the hardware TLB. Finally the different steps are evaluated using a set of micro- and macrobenchmarks and compared against a popular open-source virtualization software and against native execution of the guest OS.

Kurzbeschreibung

Nachdem die Virtualisierungstechnik Jahrzehnte lang ein Schattendasein in der Marktnische der Mainframes geführt hat, hat sie in den letzten Jahren umfassend Anwendung im Massensegment gefunden. Dieses Wiederaufleben ist den traditionellen Betriebssystemen geschuldet, die keine innovativen Lösungen für eine Reihe von Problemstellungen liefern konnten. Diese werden nun durch Virtualisierung abgedeckt und reichen von Systemmanagement über Sicherheit bis zu Erleichterungen bei Entwicklungs- und Debugging-Tätigkeiten. Zur selben Zeit hat sich die Leistungsfähigkeit von Embedded Devices massiv verbessert. Während Geräte der letzten Generation noch über sehr eingeschränkte Ressourcen verfügten, kommen aktuelle Geräte leicht der Leistung von Desktop-PCs von vor nur wenigen Jahren gleich, sowohl in bezug auf Prozessoren als auch auf Arbeitsspeicher. Diese neue Leistungsfähigkeit eröffnet auch den Embedded-Markt für Virtualisierungslösungen, wo diese nicht weniger gesucht sind.

Vor dem Aufkommen von Hardwareunterstützung für Virtualisierung war es nötig, Betriebssystemkerne anzupassen, um sie unter der Kontrolle eines anderen Kerns ausführen zu können — entweder im Vorfeld oder direkt zur Laufzeit. Beide Techniken arbeiteten jedoch mit reduzierter Effizienz, da viele Verwaltungsstrukturen doppelt geführt werden mussten, z.B. für Tasks und Speicherverwaltungstabellen. Die Einführung der Hardwareunterstützung erlaubte es Entwicklern, die Anpassungsarbeit des Gastkerns zu umgehen, das Verwalten der doppelten Speicherstrukturen blieb aber ein Hauptgrund für den erheblichen Geschwindigkeitsnachteil.

Während vollständige Virtualisierungsunterstützung inzwischen für gewöhnliche Desktop-Plattformen wie Intel x86 allgemein verfügbar ist, werden die meisten Prozessoren für Embedded Devices immer noch entweder gänzlich ohne oder nur mit grundlegender Unterstützung ausgeliefert. Da Betriebssystemkerne aber ohnehin meist an das spezielle Chip-Design, auf dem sie arbeiten sollen, angepasst werden müssen, erscheint es vielversprechend, ein Setup mit grundlegender Virtualisierung dadurch zu verbessern, indem es dem Gast-Kern durch zusätzliche Anpassungen gestattet wird, dem Host bei der Verwaltung der Speichertabellen zu assistieren.

In dieser Diplomarbeit werden daher Software-Implementierungen zur Speichervirtualisierung für Prozessoren ohne entsprechende Unterstützung entwickelt und evaluiert. Nach der Erstellung einer grundlegenden vTLB-Implementierung wird die Interaktion zwischen Gast und Host sukzessive erweitert, um immer optimiertere Lösungen zu entwickeln, so z. B. das Caching von Adressräumen, Unterstützung für globale (d. h. Context-Switch-persistente) Seiten und Optimierungen im Umgang mit dem Hardware-TLB. Abschließend werden die verschiedenen Ausbaustufen mit einer Reihe von Mikro- und Makrobenchmarks evaluiert und mit einer gängigen Open-Source-Virtualisierungslösung und nativer Ausführung des Gastsystems verglichen.

Acknowledgments

First of all, I would like to thank all members of the chair for their support — in particular Prof. Seifert for supervising my thesis, Michael Peter for his valuable suggestions concerning my measurements and helpful advice on getting it all into written form, him and Janis Danisevskis for the opportunity of exchanging our results and combining them into a conference submission and Steffen Liebergeld for the insights he gave me into the design principles of his VMM Karma.

I would also like to thank my whole family for making all this possible and of course Martina who provided invaluable moral support and encouraged me to get this finally wrapped up.

Contents

List of Figures

List of Tables

1 Introduction

Computers have undergone several significant evolutionary steps since their invention, reflecting the growing demands imposed by the environment they were deployed in. Early systems were built under the assumption that all applications were trustworthy and therefore protecting them against each other was unnecessary. The development of multi-user systems made it necessary to at least separate the processes controlled by each user from those of other users, imposing a first restriction on the premise of trustworthiness. And today, in the era of dynamically downloaded content, users can usually not even guarantee that the byte code their browser plugin is about to execute does not contain any malicious instructions, so even a single user's processes need some level of separation from each other. Contemporary off-the-shelf operating systems separate processes only at the user-ID level, so once an attacker has managed to break into a user's application he is free to access and modify the process state of that user's other processes.

Likewise, there is no separation between the subsystems of monolithic operating system kernels. Modern OSes contain hundreds of device drivers besides the mandatory core modules for memory management, permission handling etc. This poses two threats. If one of the drivers is affected by a bug, its failure jeopardizes the stability of the whole system, because the kernel cannot forcibly regain control from the malfunctioning driver. On the flip side, if the driver bug is exploitable, the security of the whole system can be compromised by breaking into the faulty driver, as there are no further restrictions between the driver and the kernel core.

With both safety and security endangered by the monolithic design, a logical step would be to attempt a modular approach in order to reduce the amount of code where a break-in would be fatal to the overall system security. In a monolithic setup, applications have no choice but to trust the whole kernel with all of its components — in other words, their *Trusted Computing Base* or *TCB* is undesirably large. With a more modular design, applications could choose more freely which components are trusted and which are not.

Assuming that the large monolithic kernels are untrustworthy, a moderate solution would be to employ the possibilities of virtualization to move certain security-relevant services out of the control of the untrusted OS, like Chen and Noble have demonstrated in [CN01]. Naturally this only works for services which are not inherently tied to the specific OS instance they are usually run on, but there are a few suitable use cases, e. g. event logging (in contrast to transmitting log messages over the network, the logging process can inspect the virtual machine directly) or even intrusion detection/prevention by creating a sandbox clone of the actual virtual machine and testing the effects of untrusted incoming data there.

A more drastic, but equally viable and well-known strategy is to reduce the operating system to the bare minimum, which provides only absolutely necessary features, and to move everything else into separate unprivileged user processes. This significantly reduces the attack surface of the OS (which is then usually called a *microkernel*) and guarantees that the crash of (or the intrusion into) a single user-space device driver has no influence on other drivers or the core and only impacts those applications which have chosen to trust this particular driver. Such a modular system design is also (due to its flexibility) perfectly suited for platforms where only a small subset of the possible functionality is desired.

However this reduction comes at a cost. Porting even simple applications to a microkernel is a complex task due to the fundamental differences in system design. Established application programming interfaces (APIs) like POSIX and the Windows family are usually easy to implement because the underlying monolithic kernel exports a similar set of system calls. Providing a POSIX-compliant interface requires not more than wrapping one (or sometimes several) calls to the operating system into a single library function.

The interface exposed by a microkernel is not only much smaller, it is also very much unlike a plain list of system calls. Besides administrative functions like process creation and destruction and the (de-)allocation of memory, the only way to interact with the system is to engage in communication with other user-space components.

For example, traditionally "simple" system calls like `sleep()` require a clock source and assistance of the scheduler, which both could well be implemented as user-space modules. If backward compatibility is desired, a third layer has to be inserted between the desired applications and the microkernel. This layer must be able to use the interface exposed by the microkernel and the functionality provided by other user-space processes to itself provide the ABI (e. g. POSIX) these applications were compiled for. A first attempt to create such a layer would be a port of a contemporary full-fledged OS to a microkernel, and such implementations indeed exist (L4Linux, MkLinux).

It is noteworthy that these ported or "Rehosted" operating systems suffer from the weaknesses they have inherited from their native form (improper separation of processes, monolithic core etc.), as Roscoe et al. rightly point out in [REH07]. While these should be rectified in the first place, this approach can yet work around some of the problems. While subverting a module of the rehosted OS still renders all its modules unstable and/or compromised, the security breach ends at the boundary of the rehosted OS: if the TCB of the underlying microkernel-based system is indeed working properly, the compromised rehosted OS cannot gain more rights than already granted to it, so other virtual machines and the applications running therein are safe.

On platforms without special hardware support for virtualization techniques the concept of OS Rehosting is the only option to create an intermediate compatibility layer. These platforms often only support two privilege levels[1] (aptly called "unprivileged" and "privileged"), so the rehosted OS has to be demoted to an unprivileged process with its own address space, as the host operating system must be — for obvious security and separation reasons — the only privileged component. Thus a syscall in the guest

[1] x86 is no exception here - although there are basically four levels code can be executed at (ring 0-3), other structures, e.g., page tables, differentiate only between "system" (ring 0) and "user" (ring 1-3).

now requires two task switches (from process to guest kernel and back, which both in turn each require calling into the microkernel, thus entering and leaving privileged mode) instead of switching in and out of privileged mode once. Copying data between user process and guest kernel poses yet more, but different problems, as do restrictions imposed by the host kernel itself like a disadvantageous API, limitations on the amount of available virtual addresses etc. Finally the design model itself mandates adapting a monolithic OS kernel to a specific microkernel and keeping the adaptation applicable for new versions of both the microkernel and the guest kernel.

If, on the other hand, there is hardware support for virtualization, a guest kernel can be provided with a flawless illusion of the actual hardware, rendering modifications to it unnecessary. This support usually implies the existence of additional privilege levels to discern host OS (the microkernel) and guest OS and thus proper separation of all layers. If the guest OS is made aware of its virtualized state and allowed to explicitly request actions by the host OS, the delays which are undoubtedly incurred by providing that illusion can be reduced, possibly through only very limited modifications to the guest — a technique known as paravirtualization.

Basic virtualization support usually only provides *register virtualization*, i. e. the processor provides special instructions to enter and leave virtualized execution and automatically loads its registers from a memory-resident state-cache structure upon entry and stores the register values back upon its return to the host OS. Support for the virtualization of additional structures, e. g. memory-resident structures as often used for the implementation of virtual memory, is much more complex to implement and validate, so it is often added separately.

This is the reason why memory management has for a long time been one of the performance bottlenecks of virtual machines on the x86 platform. The lack of support for virtualized memory-resident structures meant that even though the architecture now possessed the basic capability to execute and switch between several different operating systems on the same hardware, there still was only a single set of address translation tables. Guest systems were not allowed to handle their own part of the translation process, so the host OS had to intervene whenever a guest wanted to change its memory mappings.

Recently Intel and AMD have introduced enhancements to the x86 virtualization support which includes the desired two-tier memory management support, called Extended Page Tables (EPT) and Nested Page Tables (NPT), respectively. However this has not reached all parts of the market yet and probably never will, especially not the embedded market.

This thesis presents different approaches for platforms with only a single level of address translation, compares their suitability for different workloads and evaluates their performance against another prominent virtualization product and against native execution. The implementation leverages existing components such as Fiasco.OC and the virtual machine monitor Karma (see [Lie10]).

Chapter 2 recapitulates the technological foundations this thesis builds upon. Chapter 6 explores the designs and research results of related projects. My own design approach is explained in chapter 3, whereas Chapter 4 contains a detailed description of my implementation. Chapter 5 presents the results of the evaluation of these implemen-

tations, and Chapter 7 contains my conclusions and ideas for possible future research topics.

2 Background

To facilitate the understanding of my research efforts and their results, I would like to present the technology my work builds upon. This chapter therefore briefly revisits the concepts of microkernels, virtualization in general and on the Intel x86 platform in particular and gives an in-depth survey of the x86 implementation of virtual memory. Readers who are familiar with these topics may safely skip to Chapter 6.

2.1 Microkernels

All details aside, the role of an operating system is basically that of a service provider. It manages the hardware resources of the computer, grants or denies access and often provides that access through some sort of abstraction layer. This management pertains to all components of the hardware: peripheral devices, memory, storage space and even CPU time. Governing decisions are usually based on a set of policies which serve a wide range of goals. The earliest computational devices only ran a single job at a time, so all that was required from the operating system was a timely response. The next evolutionary step, the multitasking OS, made it necessary to multiplex the hardware resources between several processes, which introduced the policy of *fairness* (every process must yield the CPU after a certain time span has passed). Yet another generation later, multiple users were able to concurrently work on the same system, and the need for security was born (data owned by a user must not be visible to any other user, and even memory allocated to a specific process must not be accessible by any other process unless specifically granted). While all this is adequately covered by current monolithic OS kernels, the rather young threat of executing foreign code without validation or user intervention (e. g. through multimedia plugins, JavaScript engines) is still unanswered, as these operating systems do not provide appropriate separation measures.

In order to meet the demands for security, the notion of a "privilege level" was introduced and memory areas (regardless of their contents, whether code or data) could be marked as privileged, unprivileged or inaccessible. This enabled the OS to exert control over its processes by giving each one only access to the part of memory which was granted to it.

However contemporary "fat" monolithic kernels implement all their functionality at the highest privilege level. A distinction between those parts that strictly require this privilege level and those that do not is not made. This gives a malicious user plenty of subsystems to attack where a successful break-in results in complete control over the system — in other words, the Trusted Computing Base is pretty large.

When these problems became more and more apparent, researchers began to ask which components could be moved to lower privilege levels and which were the essential

core components which had to stay at the highest level. Per Brinch Hansen described such a minimal set in [Han70], which (despite its unfamiliar notion of external devices as processes in their own right) identified only three core elements: process management (creation and destruction), inter-process communication (in this specific case study asynchronous IPC using kernel buffers) and a hierarchical resource model which is only insofar part of the kernel as it has to ensure that resources can only be granted from father process to child process.

A few years later, the paradigm of this field of research was formulated by Jerome Saltzer in [Sal74]:

> The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

Its consistent application to the drivers and other subsystems of an OS kernel has led to the birth of *microkernels*: The highest privilege level contains only essential functionality (which in large parts happens to coincide with Hansen's findings) while everything else is implemented as unprivileged server processes. A large percentage of contemporary monolithic kernels is made up of device driver code. Device abstraction services are completely expendable in terms of the principle of least privilege: they do not require special privileges to operate as long as access to the devices themselves can be delegated. The parts that remain in the kernel are support for address spaces, threads of execution, scheduling and inter-process communication. This set is irreducible (although it is undoubtedly not the only irreducible set): all these components are required for sensible operation, i. e. if one of them was removed from the set, operation would be either outright impossible or at least uninteresting (e. g. a single process unable to fork a second one).

The first notable microkernel project was *Mach*, whose design principle was to keep the system API as similar to UNIX as possible while still following the microkernel paradigm. This led to the choice of the UNIX pipe as the central tool for inter-process communication, which (as an asynchronous communication primitive) implied a lot of copying and buffering. The original Mach microkernel was rather slow due to such choices.

After several first-generation microkernels had experimented with different solutions for IPC, address space management etc. without finding an obvious bottleneck, Jochen Liedtke conjectured the IPC to be the cause of the slowdown. He set out to prove his assumption that microkernels were not inherently slow and that applications could run with similar speed as on a monolithic kernel if only the design was chosen wisely. He reduced the IPC mechanism to a minimum by removing all sanity checks and even passing the message in registers if possible, to avoid the copying overhead. The resulting second-generation microkernel was called L3 and yielded very promising performance results. A reimplementation of the concepts in x86 assembly language together with a further reduction of the feature set resulted in yet another performance gain and the birth of the L4 microkernel. Liedtke presented his theories and results in several publications (see [Lie95], [HHL+97]).

As the design principle of a microkernel was now proven to be competitive, Liedtke reimplemented L4 in C++, which meant a first step to platform independence. The resulting microkernel L4::Hazelnut had no significant performance drawbacks. Liedtke and his team continued their platform independence efforts, which finally culminated in a platform-independent specification of the L4 API (see [Gro11]) and another reimplementation coined L4Ka::Pistachio.

At the same time, several other university groups started creating their own fork of L4. A team at TU Dresden implemented their own L4-conforming microkernel and focused on making the kernel as preemptible as possible. With their microkernel L4/Fiasco they were able to build an L4-based system with hard real-time characteristics. Work on this branch continued, and the successor Fiasco.OC (which this thesis builds on) contains support for basic virtualization, multi-core systems and manages permissions as capabilities, thus becoming one of the first microkernels of the third generation.

Another team at Sydney has ported L4 to a variety of architectures, e. g. MIPS, Alpha and ARM. It has also created seL4, a fork of L4 which has been formally verified to match the specification.

2.2 Virtualization

The idea of having multiple operating systems execute on the same hardware hearkens back to the 1970s. IBM released its mainframe operating system *VM* in 1972 which already was capable of running an arbitrary number of "guest" operating systems under the supervision of a control program. A few years later Popek and Goldberg wrote their groundbreaking paper [PG73] on the theoretical prerequisites for an architecture to be fully virtualizable. They formulated three characteristics which must be provided by a *virtual machine monitor* or *VMM*:

- Equivalence: the VMM has to expose an interface which is essentially equivalent to the hardware the virtual machine expects to run on
- Performance: the virtual machine should run with only minor speed decrease
- Control: the VMM has complete control over the resources delegated to each virtual machine

They continued by formally identifying *privileged* (instructions that are only executed in privileged mode, but cause a system trap when attempted in user mode) and *sensitive* instructions (those which change the privilege level or alter the resource set of the machine, and those which have different semantics depending on the current privilege level) and postulated the theorem that an architecture was fully virtualizable if the set of sensitive instructions was a subset of the privileged instructions.

In the same year Goldberg also defined in his thesis (see [Gol73] two distinct types of VMMs: Type I ("bare metal") VMMs run directly on the hardware, fulfilling the duties of both the host OS and the VMM. Type II ("hosted") VMMs run on top of a conventional host OS and therefore communicate only indirectly with the actual hardware. None of these designs is inherently better than the other, but their use cases differ: a Type I

model does not have to rely on a host OS to manage its virtual machines, so the TCB is much smaller, while a Type II might be more suitable for OS development environments where such strict security is not required.

While both terms for the VM control software, VMM and hypervisor, remained in use throughout the years, the distinction between the two was never generally decided upon. I have decided to follow the definitions laid out in recent papers and reserve the term "hypervisor" for the component which enforces the security policy (isolation, fairness of resource distribution) and "VMM" for the component which mediates access to hardware devices. Following this reasoning, a hypervisor obviously requires top privileges to perform its duties, while the VMM can also be a user process. — The term "hypercall" for the special instruction which the guest may use to intentionally return control to the host is still used, though, even if in most cases it is the VMM whose assistance is desired by the guest.

Unfortunately, while the desire for virtualization leapt from mainframes to commodity hardware around the turn of the century, this requirement was not fulfilled for the Intel x86 (or IA32) platform for a long time, as Adams and Agesen have shown in [AA06]. The prominent counterexample if the x86 `popf` instruction, which is sensitive because it alters system state (in fact, the interrupt delivery mode flag) but not privileged because it can also be executed in user mode, in which case it simply does not modify the system flags. A VMM is thus unable to properly simulate the interrupt flag to virtual machines.

After almost a decade of adapting guest operating systems to the unwieldy state of virtualization possibilities on the x86 architecture, Intel introduced virtualization support in some of its later Pentium 4 processors in November 2005 and codenamed it *VT-x*. AMD followed in May 2006 with equivalent support for their Athlon64 cores and codenamed their support *SVM* (secure virtual machine), later marketed as *AMD-V* (AMD virtualization). Both technologies are roughly equivalent and make the architecture virtualizable according to Popek's and Goldberg's criteria: they introduce additional privilege levels, provide the possibility to trap all sensitive instructions and define storage structures which carry the state of a virtual machine. This state is read from and written to during special *world switch* instructions which switch execution between guest and host operating system. The stored state information includes not only the contents of the general-purpose registers, but also shadow data which cannot be observed directly, e. g. the presence of an *interrupt shadow* (the delay of interrupt delivery after certain instructions) or the actual characteristics of the active memory segments. Both technologies furthermore allow the host to intercept exceptions (whose delivery is sensitive because it can switch the VM from user to privileged mode) and even to comfortably inject events into the VM. The latter is not required by the virtualization criteria because the procedure of taking an external event is part of the hardware interface and could also be emulated by the VMM, creating an equivalent illusion.

To increase the efficiency of the TLB in combination with virtual machines, the VT-x and SVM virtualization extensions introduced an additional concept: the *address space IDs* or *ASIDs*. While "classical" TLBs are indexed with virtual addresses alone and context switches purge most contents from the TLB, the new TLB concept uses a combination of ASID and virtual address as lookup key and does not remove any entries upon switching between virtual machines and the host. Thus multiple virtual

machines (and the underlying host) are able to leverage the caching capabilities of the TLB simultaneously. Context switches in the host of course retain their flush semantics.

All this makes it possible to perform classical *trap-and-emulate* virtualization on x86 hardware. However, both VT-x and SVM in themselves lack support for a performance-critical aspect of virtualization: management of virtual memory.

2.3 Virtual memory on the x86 architecture

Intel's x86 platform, like most contemporary platforms, manages its virtual memory spaces through a hierarchical data structure which maps units of physical memory (so-called "pages") into virtual memory at a specific address.

In its initial state, an x86 CPU operates in Real Mode, a legacy execution mode which does not support paging and whose instruction and operand size is 16 bits. Memory areas can be given different access levels by creating disjoint "segments", i. e. contiguous chunks of memory which start at a specified 20 bit address. Operating systems which want to use more memory have to switch the CPU to Protected Mode, which widens the instruction and operand size to 32 bits. In this mode, four gigabyte of memory can be referenced. Segmentation is still supported, but segment addresses are also limited to 32 bits, so this offers no benefit. On the other hand, Protected Mode does support paging, which allows far more fine-grained and versatile control over the accessibility of physical memory at different privilege levels. As the segment descriptors are already contained in the basic register virtualization and additionally are practically irrelevant in contemporary operating systems[1], I will pay no further attention to this mechanism and focus on paging instead.

In 32 bit mode[2], the page table structure consists in its basic form of two levels. The upper level is called the *page directory*. Its base address is stored in control register 3, and it has room for 1024 entries, each one responsible for resolving two megabytes of virtual memory. These entries usually contain pointers to another table of 1024 entries (the lower level, called a *page table*), whose entries map single four-kilobyte pages to physical addresses. Figure 2.1 on page 13 illustrates this translation process. If the CPU supports "page size extensions", enabling them allows the upper level to contain leaf nodes itself; these so-called *superpages* directly map two megabytes of virtual memory to a physical address. Another extension allows pages of either size to be flagged as *global* which makes them persistent across context switches — how exactly this persistence is achieved will be discussed later on. A third and final extension called "Physical Address Extension (PAE)" adds a third level to the tree and extends the range of addressable

[1] As segmentation offers no addressing benefits, all segments are usually created flat, i. e. they start at address `0x0` and extend to the highest possible address. On x86 the code segment also contains the current privilege level, so a little switching is still involved, but access control to memory is exclusively done through paging.

[2] The page tables look different on 64 bit processors. Including these in my considerations does not yield any new insights nor does it pose additional problems which have to be overcome. The actual layout of the tables is a mere implementation detail. This thesis therefore limits itself to 32 bit mode only.

memory to 36 bits (or 64 gigabyte). However PAE was not employed during my thesis, so I will not discuss it in my further analysis.

The address resolution process described above is performed in hardware and cannot be circumvented once paging has been enabled, i. e. all memory accesses must be performed through pointers to virtual addresses which have a valid entry in the currently active page table (as referenced by CR3). Successful virtual-physical address translations are cached in the Translation Lookaside Buffer (TLB), and the paging structures are only consulted when the TLB does not already contain a matching entry. If a virtual address has no entry in the page table, a page fault occurs: execution is interrupted and control flow is diverted to the page fault handler as defined in the Interrupt Descriptor Table (IDT). This handler is usually part of the operating system. It is then responsible for either creating the appropriate entry (so that execution can continue where it was interrupted) or destroying the process which provoked the fault.

Since TLB entries have precedence over the page tables, the TLB has to be invalidated if its contents are no longer a subset of the entries in the page table, i. e. if a page table entry (PTE) is either modified or removed. If the operating system wishes to transfer control between two processes which execute in different address spaces, it has to change the effective page table and therefore writes the pointer to the new address space into CR3. This operation automatically flushes the TLB. If only a single page table entry has to be altered (which happens mostly due to copy-on-write semantics, where a shared read-only page has to be copied and remapped as writable for each process which issues a write request), the TLB can be selectively invalidated by using the *Invalidate Page* (mnemonic *INVLPG*) instruction.

As all addresses contained in the page table entries are aligned at a 4KB boundary, the low 12 bits of each entry are available for additional pieces of information. The paging mechanism uses these to store a set of flags which convey additional properties of the page. Some of these are set by the operating system and denote permission details, e. g. whether a page may only be read from or also written to, or whether the page should be exempt from TLB flushes caused by address space switches (the *Global* flag). The latter is commonly used for pages occupied by the OS itself to save unnecessary retranslations, as these pages are usually mapped into every process at identical addresses.

Other flags are set by the hardware page table walker. These indicate to the OS whether the page has been read from at least once (the *Accessed* flag) or written to (the *Dirty* flag). The OS may use these (together called the *Accessed/Dirty Assists*) in deciding whether a page which represents a part of an on-disk file has to be written back to disk.

The following list summarizes the control registers whose contents influence the behaviour of the paging mechanism on x86.

- Control register 0 (CR0)
  This register is a bit field and contains the activation bits for both "Protected Mode" and "Paging".

- Control register 3 (CR3)
  This register contains the pointer to the base of the page table structure.

- Control register 4 (CR4)
  This register is a bit field and contains the bits for "Page Size Extensions", "Global Pages" and "Physical Address Extension".

2.4 Paging under Virtualization

In the previous section we have seen the basic principles of paging on the x86 architecture, with Figure 2.1 showing the translation process for an exemplary virtual address. The page directory is located at the physical address contained in CR3, and the translation continues down the page table structure until the final page is found. It has also been shown that the x86 architecture (augmented with VT-x or SVM) is fully virtualizable. I will now demonstrate how virtual paging can be implemented using plain trap-and-emulate semantics.

As the paging structures control access to memory, it is obvious that the page tables governed by the guest cannot be used as effective page tables for security reasons alone. Additionally, the tables created by the guest translate only between the guest's own notions of virtual and physical memory, as the guest cannot know where its memory is located in real memory. Instead the VMM must trap all guest instructions and events which affect the address translation process, using the information provided by the guest page tables to construct a policy-conforming and properly translated copy, commonly called a *virtual TLB* or *vTLB* (see Figure 2.2 on page 14 for a schematic overview). The vTLB has to fulfill the following properties:

- Containment: valid entries in the vTLB may only reference parts of physical memory which have been delegated to the corresponding VM.
- Freshness: the vTLB must not contain an entry for a virtual address if the guest page table contains one at the same virtual address, unless the hardware TLB might still contain a cached entry for that address.
- Equivalence: each valid vTLB entry must be a correct translation of the corresponding guest page table entry, unless the hardware TLB might still contain a cached entry for that virtual address.
  Additionally, if an entry has been marked as Accessed or Dirty in the vTLB, the flag must also be set in the corresponding guest page table entry.

It is now easy to demonstrate that it is sufficient to intercept the following events:

- page faults occurring in the guest
- write operations to the control registers which affect paging (as shown above)
- the *INVLPG* instruction

The first intercept ensures the liveness of the system: as the guest page fault handler cannot add entries to the effective page tables, each page fault requires additional work by the VMM, or the page fault would be rethrown indefinitely. The VMM code

responsible for handling guest page faults can also easily ensure the Containment and Equivalence properties[3]. It is however usually necessary that the guest handler runs before the VMM, because otherwise the VMM has no entry it could translate. Unfortunately, the virtualization intercepts are designed to replace the intercepted event. Thus a single address commonly faults twice, with the first fault being reinjected into the guest and the second one being used to update the vTLB.

The other two listed intercept classes pertain to invalidation scenarios. If the guest alters or removes one or several entries from its page tables, it can only rely on these changes to become effective when the TLB has been cleansed of cached translations for these addresses. To achieve this, the guest can only do two things: write to a control register, thus triggering a complete TLB flush, or invalidating a single address by executing the *INVLPG* instruction. Both ways are intercepted by the VMM which then updates both vTLB and hardware TLB.

While this "manual" guest-physical to host-physical translation process provides the desired illusion of faithful virtualization, it does so at a severe performance loss: page faults are effectively duplicated and shadow structures of possibly considerable size have to be allocated and kept up to date.

The hardware-based solution to this problem is to provide two translation structures, one responsible for the translation of guest-virtual to guest-physical addresses (thus being functionally equivalent to page tables in a native environment) and the second one responsible for the translation of guest-physical to host-physical addresses, effectively making the VMM duties as described above (including the management of the Accessed and Dirty Assist bits) a part of the hardware lookup process.

Intel and AMD introduced exactly such virtualization extensions a few years after their basic virtualization support. Both technologies (*Extended Page Tables (EPT)* and *Nested Page Tables (NPT)*, respectively) use hierarchically structured page tables for both levels of the translation.

It is interesting to note that due to this choice, the page table walk for a single virtual address is proportional to the number of layers of both page tables. At each step of the lookup process in the guest page table, the resulting intermediate address has to be translated to a host-physical address using the outer (or "nested", as AMD irritatingly calls it) page table. Figuratively speaking, the lookup in this scenario is two-dimensional — see [BSSM08] for a more elaborate discussion of the process and possible optimizations. Figure 2.3 on page 15 depicts this translation process, with the second dimension represented by small boxes labeled "NPT".

The memory available to the guest is comprehensively defined by the entries in the nested page tables. Unless the microkernel manages these tables lazily, each nested page fault (a special kind of page fault which is raised when a lookup in the second dimension fails) represents misbehavior of the guest and can swiftly be treated as such.

[3] A very simple implementation for the Equivalence property concerning the Assist bits would be to set these immediately when creating the vTLB entry. There are of course more sophisticated solutions — those will be discussed later.

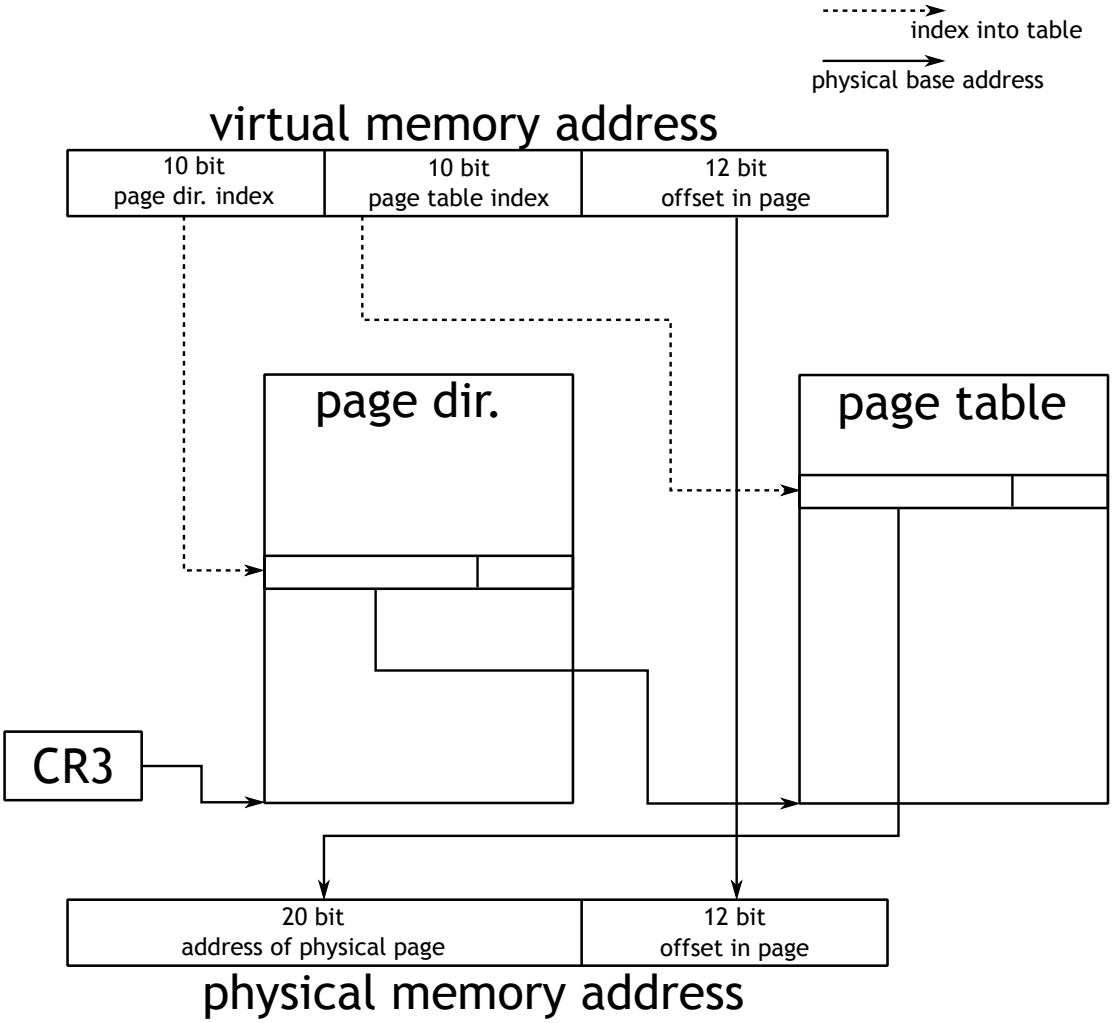

Figure 2.1: Page Table Structure — native mode (no virtualization)

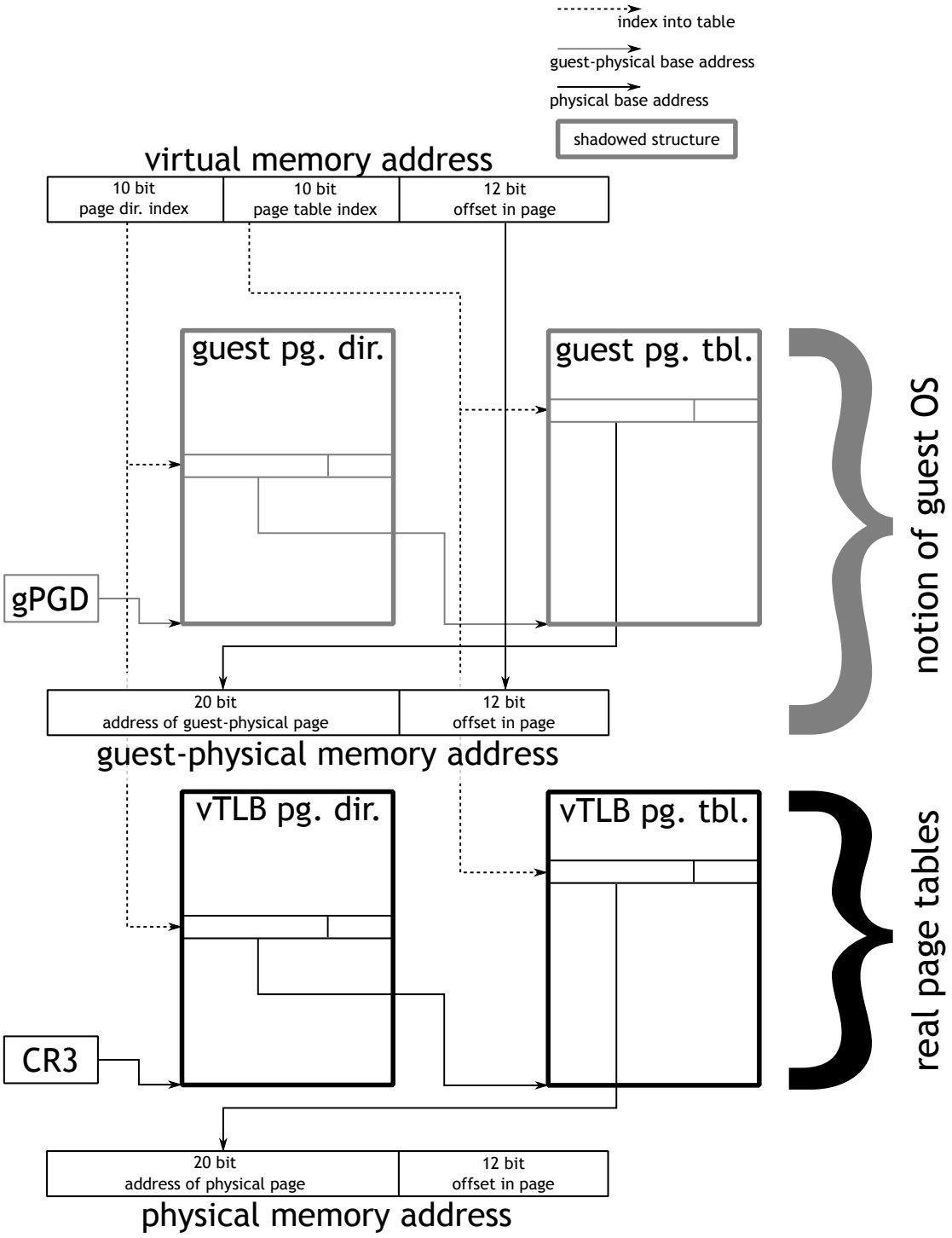

Figure 2.2: Page Table Structure — shadowed guest structures during vTLB operation

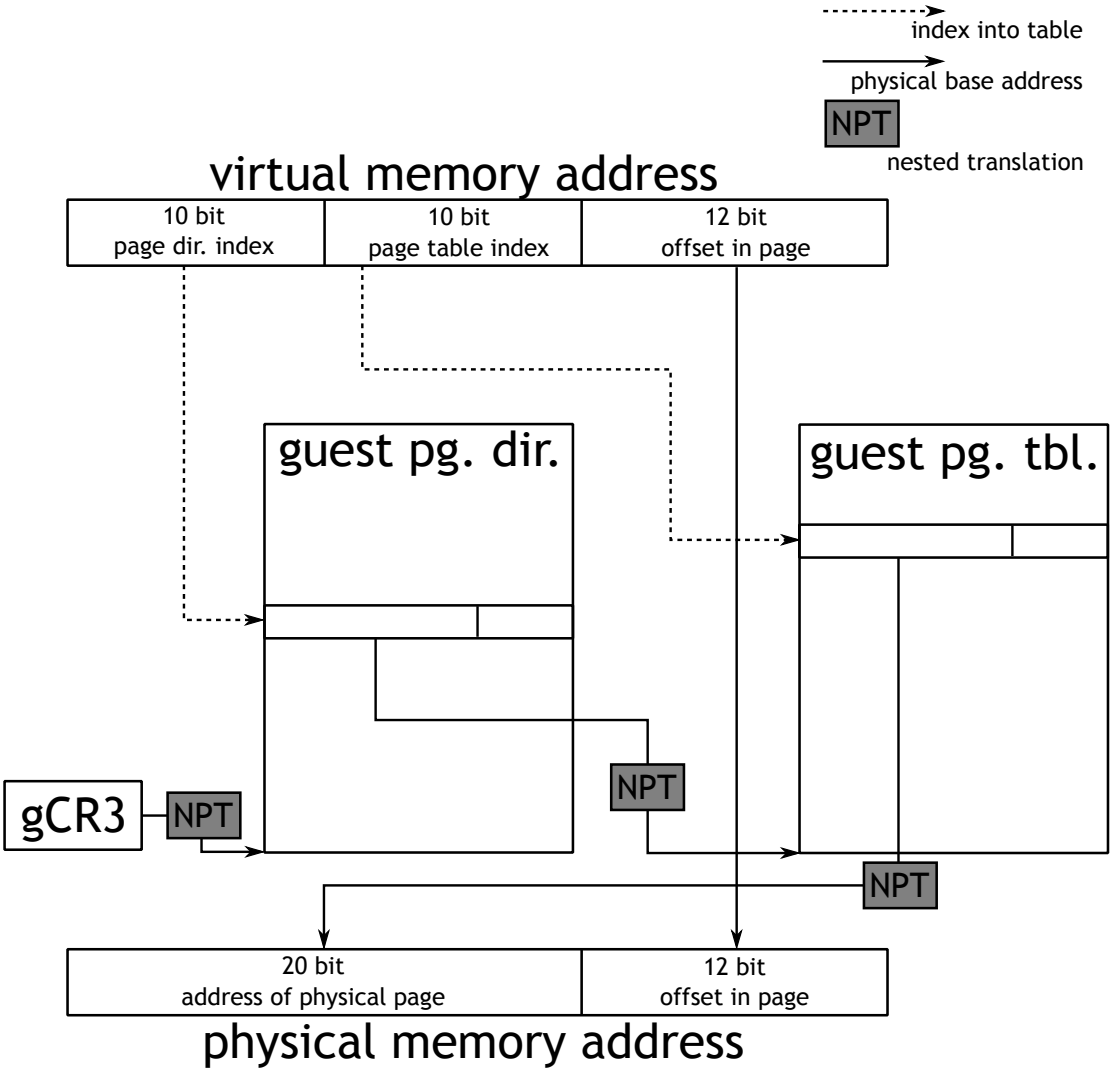

Figure 2.3: Page Table Structure — nested paging (hardware MMU virtualization)

3 Design

The basic setup for my thesis can be succinctly by its three components. The first layer, running directly on x86 hardware with support for register virtualization, is the Fiasco.OC microkernel. On top of it runs the virtual machine monitor Karma, which in turn governs a virtual machine running Linux. The ultimate and single goal is to achieve as much performance for applications running inside the virtual machine as possible.

During initialization, Karma creates a kernel `Vm` object and obtains a capability to it. While this object provides room for the VM address space (as its class is a subclass of Task), Karma has to allocate storage for the virtual machine's register state and control information block itself.

The design of the virtualization API of Fiasco.OC allows this VM address space to be used both with Nested Paging turned on or off; with NPT enabled it represents the nested page tables, with NPT disabled it represents the vTLB.

Karma's setup code allocates the "main memory" of the virtual machine and maps it to address `0x0` of the VM task. If NPT is enabled, this is all the VMM has to do (if no further device-specific mappings are required). If NPT is disabled however, this means that the whole of main memory has been anticipatorily faulted into the vTLB. Fortunately this is even beneficial, as it prevents page faults from occurring in the early bootup phase up to the point where the guest OS activates Paging mode and creates its first set of page tables[1].

From that point on, the vTLB implementation must create and destroy page table entries according to the needs of the virtual machine — as efficiently as possible. I will now describe the evolutionary steps of my design.

3.1 vTLB in user space

The vTLB implementation has to perform an address translation in software which entails the following steps:

- Host-virtual to host-physical:
  This is easily available by traversing the effective page tables of the VMM task itself.

- Guest-physical to host-virtual:
  This is the most important one. Memory delegations from the VMM to the VM must be memorized somewhere to provide this reverse lookup. Fiasco.OC's central

[1] More accurately, it *believes* to activate Paging. Without NPT, the guest always runs with a vTLB, it is impossible to switch it off. Toggling `CR0.PG` (Paging Enabled) in the guest simply has no effect.

mapping database is capable of this if the delegations have been created with the `sys_map` syscall.

- Guest-virtual to guest-physical:
  This part of the translation is contained in the guest page tables, which the VMM has to walk while constantly resolving each intermediate address to a host-virtual address (cf. the previous item). If the guest has not yet entered Paging Mode, this part of the translation is an identity mapping and can be omitted.

While there are comfortable API calls to perform the first translation and there is little room for optimization for the third translation itself, finding a fast solution for the second one is the only interesting part.

The L4 mapping database (mapdb) would be the perfect choice if it was not for its size and complexity. The database is an enormous tree structure and stores all existing memory mappings between any two L4 tasks. Each page of physical memory can be traced from the leaf-node L4 task which uses it all the way back to the σ_0 process which acts as the central memory manager. These lookups are very powerful, but they provide much more data than what is actually necessary for the translation at hand. I have therefore chosen to use a simple array and a linear lookup function, as the number of memory delegations besides the main VM memory area usually stay below 10.

Installing the additional interceptions necessary for vTLB operation can be done by simply activating the desired intercepts in the VMCB control structures and adding more cases to the list of SVM exit codes Karma has to check.

Operations on the L4 task which serves as the VM address space can be performed through the `Task->sys_map` and `Task->sys_unmap` syscalls, which involves giving the VM control over the memory region and creating a corresponding entry into the mapdb. Clearing the entire task is not available as a special operation, but the `Task->sys_unmap` syscall accepts a base address and a power-of-two size as arguments, so even clearing a whole address space takes a single call.

One optimization is already possible at this stage. Operating systems usually make use of the *global* flag to mark pages occupied by themselves as valid in all address spaces. If the OS runs natively, this causes these entries to remain in the TLB even across changes to CR3. This behaviour is easy to mimic in the vTLB implementation.

3.2 vTLB inside the microkernel

As page faults are one of the most frequent reasons for the VM to pass control to the outside world, handling these in user space causes significant delays. Also the vTLB implementation only represents a mechanism which enforces the memory access policies set forth in the list of mappings the VMM has granted to the VM. Therefore a comparative implementation of the same features as part of Fiasco.OC seemed reasonable and promising.

When VM execution is intercepted, Fiasco.OC resumes its execution after the `vmresume` instruction. As the surrounding function is part of the Vm object implementation, the VM address space is immediately available and can be modified without

any lookup costs. The exit code conditions for the cases listed above are then checked, and if one of them matches and appropriate action was taken, then the VM is immediately re-entered without exiting to the VMM. All other exit reasons are passed on to the VMM so access to hardware devices can still be emulated/mediated there.

Creating new mappings for the VM can now be implemented less heavyweight by avoiding the mapdb and calling the abstraction functions responsible for page table manipulation directly. Apart from that however, the page fault handler itself cannot be improved further, as its core functionality is simple and irreducible: walk the guest page tables, determine whether the desired entry is present and finally either insert the entry into the vTLB or forward the fault to the VM.

However during a context switch all page table entries in the vTLB are deleted, which means that the same memory address causes page faults again and again.

3.3 vTLB Caching

Preserving TLB entries across context switches seems to be the key to further reduce the virtualization overhead, but doing so is not straightforward because the vTLB implementation has to follow the constraints and guarantees of the hardware. The x86 architecture specification declares (cf. [Int]) that the TLB is emptied when the control register containing the base address of the page table (CR3) is written to, so an OS can rely on the fact that it does not contain stale entries. More to the point, an OS could switch from context A to context B, remove page table entries from the (now inactive) context A without special invalidation instructions and finally switch back to A: the hardware specification ensures that there are no entries in the TLB that were created before the context switch.

Therefore a conforming vTLB implementation cannot simply store the vTLB for a specific context and restore it later when the OS switches back to it — there is no guarantee that both still contain the same entries. One possible solution would be to validate the whole structure during restoration, but that would give up any performance benefit, because checking it against the guest page tables is as slow as simply reconstructing it from them[2]. The only other possibility is to modify the guest to alert the VMM of any changes to inactive contexts.

In order to keep the solution simple and the changes minimal, I exploited the multiprocessor features of the Linux kernel. Multiprocessor systems have to deal with the same problem: if a particular CPU is about to make changes to a paging structure which is not (nor not exclusively) its own active context, it either has to make sure that the context is globally inactive (no CPU uses it as active context) or it has to notify the affected CPUs of the change by sending an inter-processor-interrupt (IPI), an action which has been termed *Remote TLB Invalidation* or aptly *TLB Shootdown*. The functions responsible for this are located in `arch/x86/mm/tlb.c`.

The desired result is that the VMM is notified exactly once for every removal of a page table entry. If the context is active on the notifying CPU, other CPUs still possibly

[2] It still might be desirable to store and check the guest page tables, however the use cases are pretty rare, e. g. debugging the TLB handling code of the guest.

have to be notified of the removal, but we do not need a hypercall in this case because the local TLB has to be made aware of the change as well, and the necessary *INVLPG* instruction is intercepted anyway.

One final problem concerns memory reuse: the guest might destroy a page table structure and create a new one at the same address later on. If the VMM is not aware of this, it could probably restore stale entries from the cache. Thus context destruction has to be supplemented with a hypercall as well.

The impact of these changes (although minimal in terms of code delta) is hard to estimate. It could be unnecessarily expensive, because the guest cannot know whether the modified context is cached in the vTLB implementation at all and thus part of the hypercalls are superfluous. The exact percentage is likely going to depend both on the average number of runnable processes (i.e. active contexts) and the number of vTLB cache slots.

3.4 vTLB Update Batching

Using the aforementioned design, every genuine page fault (i. e. one that is caused by a missing entry in the guest page table, not just in the vTLB) usually causes two interruptions in VM execution. Figure 3.1 depicts which events occur and where execution is transferred to in each step:

1. A page fault in the guest occurs. As the VM intercepts these faults, a VMEXIT event is generated and control is transferred to the hypervisor.
2. The hypervisor examines the event and forwards it to the vTLB code.
3. The page fault intercept handler inspects the guest page table. As the desired entry is not there, it prepares the page fault for reinjection into the VM and calls upon the microkernel to resume the VM.
4. The microkernel issues the VMRESUME instruction. The injected page fault is taken immediately, and execution is diverted to the page fault handler of the guest kernel.
5. The guest kernel inserts the desired entry into the guest page table (first write operation, indicated by "#1"), and execution is finally returned to the guest application which caused the fault.

As the vTLB is still lacking the appropriate entry, retrying the interrupted instruction of the guest application causes another page fault in the guest:

1. The same page fault occurs again. Control is transferred to the hypervisor...
2. ...and on to the vTLB implementation.
3. The page fault intercept handler walks the guest page table, finds the desired entry, checks its validity (permission bits vs. page fault error code), does the necessary guest-physical to host-physical translation and eventually updates the vTLB ("#2" in the diagram). It then asks the microkernel to resume VM execution.

4. The microkernel issues VMRESUME, and the guest application can finally continue.

The double VMEXIT is undesirable because VM entries and exits are very expensive and can only be made cheaper by leveraging additional CPU features[3] which are even less widely available than Nested Paging. If however the page fault handler of the guest OS could be changed so that modifications to the guest page tables are pushed into the vTLB through a hypercall, the host would not even have to intercept guest page faults.

This additional optimization approach looks very desirable, but there are a few subtleties. If the host no longer intercepts guest page faults, the guest must be prepared to handle cases which cannot happen on physical hardware:

- The guest page fault handler must be able to gracefully handle the case that the guest page table already contains the desired entry and simply call out to the host to update the vTLB without making any changes to its own page tables. This might happen if a vTLB cache has been evicted because there were more active contexts in the guest than vTLB cache slots.
- The memory pages containing the GDT, the IDT and the page fault handler itself must be present at all times. Otherwise delivery of guest page faults will fail and a guest double fault is generated which it is difficult to recover from. The guest prevents this from happening by setting the Global Flag on these pages — this must be honoured by the vTLB.

On the other hand, the number of hypercalls can then be reduced even further by coalescing page table changes made by the guest OS on behalf of guest user land (e.g. caused by glibc `mlock`) and flushing these changes immediately before the guest exits to user land again.

3.5 Coupled ASIDs

CPU support for the TLB features related to address spaces can be diverse even between systems with an equal level of basic virtualization support. Therefore the current virtualization implementation of Fiasco.OC relies only on those ASID-related features which are guaranteed to be present in conjunction with SVM support: a non-zero number of selectable ASIDs for virtual machines and the VMCB control command to flush all ASIDs from the hardware TLB.[4]

The strategy chosen by Fiasco.OC simply allocates a new ASID for the VM if the VMM (whether in user land or as a part of the microkernel) indicates that it has made changes to the vTLB which are TLB-sensitive (i. e. which would require a TLB flush in a non-virtualized setup), thus imitating the original behaviour as closely as possible.

[3] To be precise, by marking certain areas of the VMCB as unmodified, allowing the CPU to use cached values instead of reading from memory — a feature known to the Linux kernel as *VMCBCLEAN* and presented in the AMD manuals as "VMCB State Caching", see [AMD].

[4] Additional features include flushing only a single ASID from the TLB or even — if desired — only its non-global entries.

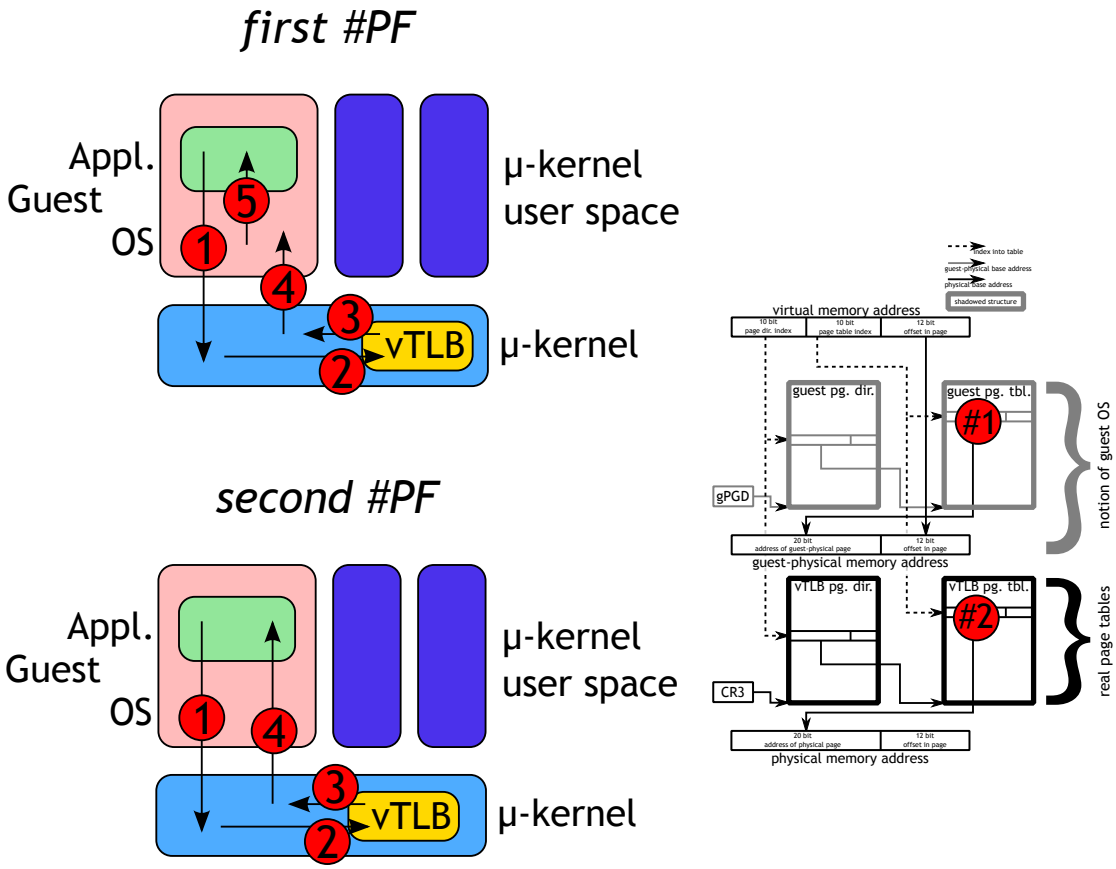

Figure 3.1: Usual sequence of events upon guest page fault

When all ASIDs have been used up, the ASID flush command is issued and the cycle starts again at the lowest ASID.

On the other hand a tagged TLB offers a far more sophisticated approach. The vTLB cache slots can easily be coupled with a specific ASID, so that switching back and forth between any two contexts would reuse the existing ASID (and thus make the original TLB contents available again) whereas each switch would mandate a TLB flush in a non-virtualized environment. The resulting performance should be close to that of two hardware TLBs, because the entries from both processes are retained in the TLB while switching back and forth between them. The advantage might be diminished by TLB collisions depending on whether the ASID is part of the TLB indexing scheme[5], but should be measurable nonetheless.

[5] More verbosely: depending on whether, for example, the entries for virtual address `0x08048000` (the base address for 32-bit ELF binaries' text section) in ASID 3 and 4 would evict each other from the TLB.

4 Implementation

The software components my thesis uses as evaluation vehicles, i. e. the Fiasco.OC microkernel and the Karma VMM, are both written in C++ and have taken a very similar approach at creating an object-oriented abstraction of virtualization. As the VM class of Fiasco.OC is vendor-agnostic, there are specific specialisations for AMD SVM and Intel VT-x. For the scope of my thesis it was sufficient to add my vTLB data structures to the generic VM class, thereby extending its functionality as an address space, and adding code to the specialized VM_SVM class implementation.

The approach taken by the user land VMM Karma differs only in encapsulating the memory management (nested paging or vTLB) into a separate class. Main VMEXIT handling is done by the L4_VM class, and memory-related operations (e. g. page faults, reactions to control register writes) are delegated to the L4_MEM class.

Especially Fiasco.OC has been a "moving target", as it is still under heavy development itself. The patches created for this thesis had to be ported several times to more recent versions of the microkernel.

Karma was only modified to stay compatible with the syscall interface exposed by Fiasco.OC. This interface was significantly changed from a straight call-into-VM/return-from-VM system call to a continuation-style calling convention (VMM calls upon microkernel to resume the VM; upon VMEXIT the VMM is resumed at a designated VMEXIT handler function instead).

4.1 vTLB in user space

At the outset Karma had only support for AMD processors with Nested Paging. The first step was therefore to implement the vTLB functionality in the VMM. The required additional intercept conditions and their handlers have been added as new cases to the main event loop's `switch` statement, as each of these is indicated by a unique VMEXIT code.

Of these cases, the page fault handler additionally required a page table walk algorithm. Karma had no need for this before, and the microkernel does not expose its internal page table abstractions to user land, so one was created from scratch.

Handling a page fault causes comparatively little work in terms of code logic: after a few address space translation operations, the page fault is either injected into the guest or an entry is added to the vTLB. If the page directory for this virtual memory area did not exist beforehand, it has to be allocated, but this occurs only for a smaller fraction of the faults, because applications, libraries etc. are usually mapped in only a few, tight clusters into virtual memory.

Afterwards the Accessed and Dirty bits have to be migrated back to the guest page table. There are different possible strategies to accomplish this. The simple solution

is to immediately flag all pages as accessed and writable pages as dirty as soon as the corresponding vTLB entry is created, the rationale being that no further access to that page will be intercepted. This implies that the guest OS probably flushes more pages back to disk than it strictly would have had to, if pages were mapped writable but never actually written to. A more conservative approach maps pages which are faulted in by a "read" page fault as readable, regardless of the actual protection of the guest page table entry. This counters the effect of superfluous flushing mentioned above, but provokes additional (vTLB-only) page faults if the page is subsequently really written to.

Microbenchmarks have demonstrated that these cases (read page fault for a page which is writable in the guest page table) only happens in the early bootup phase. Thus the discussion is moot and of little value to my performance evaluation. I have therefore chosen to go with the simple method (immediate dirtying).

Coming back to the effort estimates for the new interception cases, invalidating a single page in the vTLB (as a reaction to the *INVLPG* instruction) causes even less work than a page fault, because there is no need for any translations or memory management.

On the other hand, modifications to the paging-related bits of a control register require the whole vTLB to be reset. This entails freeing all allocated page directories and zeroing the used pages, which (depending on the number and distribution of entries) consumes far more time than handling a page fault.

This implementation form suffered besides the huge performance issues also from an API problem. Originally, the `Task::sys_map` syscall was designed to delegate memory from one task to another by creating new user-mode page table entries in the destination task at a desired virtual address. Applying this to virtual machines poses two problems. Firstly regular L4 tasks may only map pages into a subset of their virtual memory, because Fiasco.OC occupies a reserved range. This restriction cannot be imposed on virtual machines because it violates Popek's and Goldberg's Equivalence criterion. Secondly the syscall is unable to create supervisor-mode page table entries, because such entries would be pointless for regular L4 tasks.

While the first restriction was waived for `Vm` address spaces, the second problem has still not been remedied. This shortcoming prevents a user land VMM from properly separating guest OS and guest applications, because all guest pages are mapped equal (i. e. with user access enabled). Thus guest applications can write to any memory area the guest OS has write access to, enabling malicious guest applications to easily gain control over the VM.

Due to the constrained map/unmap interface the user space vTLB is an inherently unwieldy and slow design approach. It is included in this thesis only to give a complete overview of the design possibilities and to provide a baseline for the measurements. Therefore it has not been improved by the various optimization techniques discussed in Chapter 3, not even by the "global guest page tables" optimization which would have been technically possible to implement even at user level.

4.2 vTLB inside the microkernel

For the next iteration the intercept handler code for these conditions (write to CR3, write to CR4, *INVLPG*, page fault) were taken from Karma and added to the set of intercept handlers in the core VM driver of Fiasco.OC. While already saving the privilege transitions between microkernel and VMM, this also offered opportunities to optimize the way the handlers itself were implemented. As these were now a part of the microkernel, page table operations could be done directly on the in-kernel structures, avoiding expensive but unnecessary subsystems like e. g. the costly mapping database.

The page table walker was copied almost verbatim from the implementation used in Karma, adapted only to the different type nomenclature in Fiasco.OC. The kernel abstractions for page tables were now basically available, but they did not lend themselves to the task as they are only capable of performing one-dimensional page walks. Also the "global guest page table" feature was added, which was particularly easy because Linux has only non-global pages below `0xc0000000` and only global pages above.

However, handling all guest page faults inside the kernel is not straightforward as the management of device-specific memory mappings is still done by the VMM. The address translation process during the guest page table walk needs complete knowledge of the VMM address space and which of its memory regions have been delegated to the VM.

Several solutions were possible to notify the vTLB code of memory delegations from the VMM to the VM. The most obvious (and chosen) one was to intercept these delegations right as they happened, i. e. hijacking the `Vm::sys_map` syscalls and copying their arguments into a separate slim mapping database which could then be used to resolve all guest-physical addresses which were not part of the guest's main memory.

4.3 vTLB Caching

In order to implement the caching mechanism I created an array whose size could be determined at compile-time. Allowing the cache to grow and shrink dynamically would be easy to implement, but it was not necessary for my measurements. Each element of the array contains the following elements:

- the physical base address of the guest page table, i.e. the value of CR3 when this context is active — this serves as the unique identifier
- a Mem_space_vm object which contains the actual page table structure
- a timestamp that is updated each time the context is activated
- a few miscellaneous members to carry per-vTLB statistics, flag bitfields etc.

The effective Vm::mem_space pointer is then simply switched between the array elements. The timestamp enabled me to implement simple LRU-based eviction for cases of cache contention instead of having to choose a random context.

4.4 vTLB Update Batching

The double round trip to the hypervisor for a page fault (see Section 3.4) can be replaced with one hypercall by updating the vTLB during the guest page table update process itself. Earlier authors (see [AA06]) have done so through a technique called "tracing", wherein the memory containing the guest page table was tagged read-only in the vTLB so that each modification attempt caused another page fault. This also removed the necessity to intercept guest page faults and did not depend on any modifications to the guest OS, but it still relied on the interception mechanism and was thus constrained to a 1:1 ratio, i.e. each page table update caused exactly one intercept.

I have chosen a different approach which relies on paravirtualization. The guest is allowed to accumulate changes to the page table of the current context in an internal buffer without being intercepted by the VMM. These changes are pushed to the VMM as soon as one of the following happens:

a) the guest OS is about to pass control back to guest user land

b) a page fault occurs for an address which is already in the update queue

c) the buffer overflows

The first choice is motivated by the observation that once Linux is running, the vast majority of page faults are related to user space memory; the pages which belong to kernel space are marked as "global" and are therefore quickly propagated to all vTLB caches. Thus it is safe to assume that the addition of new vTLB entries can safely be delayed up to the point when the guest is about to resume execution in user land, coalescing multiple vTLB updates into a single hypercall. Usually there is still going to be a 1:1 relation (as most page table updates are performed by the page fault handler, which can only react to a single address at a time), but some syscalls might substantially benefit from this design, e. g. `madvise()` and `mlock()`.

The second item covers the cases where a user space page is faulted in by kernel code; e. g. this might happen if a `read()` syscall is issued whose buffer has been allocated, but not yet used. In this scenario the kernel itself causes the page fault, the page table is updated and the change queued — but as no exit to user land follows, the update is not submitted to the VMM and the address faults again.

The last item is included to demonstrate the benefit of dealing with a lot of page faults through one hypercall. We assume that if the guest kernel adds such a large number of PTEs without exiting to user land these pages are actually needed by the process, so it is worthwhile to add them to the vTLB.

While adding the corresponding receiver code to Fiasco.OC was pretty easy (and constituted of just another `case` statement in the giant switch), the necessary work on the Linux side for this goal included changes in all exit paths to user land (contained in `arch/x86/kernel/entry_32.S`), the #PF handler (`arch/x86/mm/fault.c`), several macros and inline functions in `arch/x86/include/asm` which are responsible for setting and clearing single PTEs and of course implementation of the batch submission function (located in `arch/x86/mm/tlb.c`).

It is interesting to note in this context how the Linux kernel actually invalidates the hardware TLB. For a single invalidation, the *INVLPG* instruction is chosen if it is available. For anything more, a full flush is performed (by rewriting control register 3 with its own contents). If the VMM handles these CR3 rewrites by doing a real vTLB flush, the Linux PTE macros require much less changes, especially the removal functions like `native_pte_clear()` can be left out. Instrumenting the whole list of native PTE modification functions allowed me to make CR3 writes a complete no-op if the value wouldn't change.

4.5 Coupled ASIDs

Changing the ASID management to properly reflect context switches in the guest OS is pretty minimal, as all necessary pieces of information are already available in the microkernel. The vTLB cache structure is augmented by another member which contains the ASID the VM was running under when the corresponding address space was last active. If the vTLB is changed in a way which requires flushing the ASID from the TLB, the member is set to zero (which is an invalid value for a VM because the host runs at ASID zero). If this address space becomes active later on, its ASID is reused if it is nonzero — otherwise a new one is allocated as usual.

When all ASIDs are exhausted, the only backward-compatible choice is to flush the whole TLB and invalidate all ASIDs at once. This requires iterating over the vTLB cache and setting all ASIDs to zero as well, because the ASID generator assumes that once a new ASID generation is started, all old ASIDs fall into disuse immediately.

This poses a small overhead for workloads with a high process creation/destruction rate, but this is usually offset by the advantage of fast switching between communicating processes (e. g. through UNIX sockets or pipes) without consuming ASIDs and losing TLB entries.

5 Evaluation

All experiments were conducted on an AMD Athlon 64 X2 Dual Core 5200+. The machine has 4 gigabytes of RAM, but for my tests I limited the amount of accessible memory to 256 megabytes.

The basic vTLB implementation and its optimized variants developed during this thesis are compared in the following sections to native Linux and to a KVM-based setup using a couple of benchmarks. The labels in the figures correspond to the following development steps:

- *vTLB:Karma*
  vTLB implementation integrated into the user land VMM, as described in Sections 3.1 and 4.1.
- *vTLB:Fiasco*
  vTLB implementation integrated into the Fiasco.OC microkernel, without any optimizations (see Sections 3.2 and 4.2).
- *+global*
  The above implementation with support for guest global pages.
- *+cache*
  The above plus a vTLB cache with room for 8 address spaces, as explained in Sections 3.3 and 4.3.

  Changes to the guest kernel: hypercalls for address space destructions and TLB shootdowns.
- *+no#PF*
  All of the above optimizations plus support for disabling the page fault intercept, which requires among other things PTE Batching. See Sections 3.4 and 4.4.

  Changes to the guest kernel: hypercalls for all PTE modifications, PTE Batching support, modified page fault handler.
- *+asid*
  All of the above plus the logical connection between VM ASIDs and vTLB cache slots (cf. Sections 3.5 and 4.5).

  Changes to the guest kernel: the above.
- *+invlpga*
  All of the above, but invalidate VM PTEs by issuing *INVLPGA* instead of allocating a new ASID.

  Changes to the guest kernel: the above.

These are compared against:

- *LinuxNative*
  A native vanilla 32-bit Linux kernel (version 2.6.35) running directly on the test hardware.
- *KVM*
  A native vanilla 32-bit KVM-enabled Linux kernel (version 2.6.35) running a Qemu instance as sole user space process, which in turn runs another copy of the same Linux kernel as virtual machine. The VM has direct access to the hard disk.

5.1 Microbenchmark "forkwait"

The `forkwait` benchmark has been employed numerous times to evaluate virtualization implementations. It is very simple and focuses on process creation and destruction, which both are expensive operations in a virtualized environment.

Creating a process involves copying its address space, and operating systems usually avoid immediately copying all writable pages and instead flag the page table entries as read-only, postponing the copy process to when the write is actually attempted (copy-on-write). As the writable entries might already exist in the TLB, these have to be invalidated.

Destroying a process means destroying and freeing its page tables, which is of no special importance for the vTLB management code, but depending on the sophistication of the vTLB paravirtualization the operations performed during the destruction might be interpreted as vTLB updates.

The core of the benchmark is a simple loop:

```
while (--N) {
  int f = fork();
  if (f < 0) { printf("Fork() error"); exit(1); }
  if (f == 0) exit(0);
  (void)wait(NULL);
};
```

The parent process continuously creates child processes, waiting for each one to terminate before spawning the next one. The output metric is number of processes created in a specific amount of time or, to be more in line with the rest of the benchmarks, time spent to create and destroy a specific amount of processes.

Figure 5.1 on page 38 shows the benchmark results for $N = 40000$. The initial version runs more than ten times longer than native Linux, but the main share of the delay is caused by the repeated page faults for guest kernel pages. Simply adding the optimization for guest global pages brings the number of intercepted page faults down to a fifth and the runtime to less than a third.

Caching of vTLBs has understandably only limited effect on the benchmark, as process creation and destruction cannot benefit from it: Of the 120000 intercepted writes to CR3, about 40000 are read-write cycles (i. e. forced TLB flushes, one for each child

process), so the remaining 80000 are the switches from the parent to the child and back. Thus each child context is only used a single time, rendering vTLB caching useless for them. Only the parent is able to benefit from retaining its pages in the cache, but its working set is rather limited. It still yields a drop from 347 to 313% native execution time, though.

The removal of the page fault handler gives another massive boost down to 258% due to the sheer number of PTE additions and removals involved. As the vTLB allocates a new ASID for the VM if at least one PTE is removed during a hypercall, coalescing several removals into one update also saves on the number of consumed ASIDs, which is almost halved.

The coupling of ASIDs with vTLB cache slots allows the pages of the parent to even remain in the hardware TLB and further reduces the number of consumed ASIDs. The latter presents a speedup in itself, too, as the tagged TLB has to be completely flushed when an ASID generation has been used up. This is likely the reason why invalidating all removed PTEs using the *INVLPGA* instruction is even faster than continuously allocating ASIDs, even though executing all those invalidation instructions is much slower than switching to another ASID.

The final ASID count is 667 generations, which adds up to 42688 ASIDs. With ASID 0 reserved for the host and another one continuously allocated by the parent process, this count is only marginally above the number of created processes. Rapid population and destruction of address spaces is however still the crucial point in efficient vTLB management, and the lazy fault-in strategy employed by the Linux kernel even precludes the vTLB from anticipatorily copying entries from the guest page tables which have not yet caused a fault.

The KVM implementation is more efficient than the basic implementation, but it is outperformed simply by honouring the global bit of the guest PTEs.

5.2 Microbenchmark "touchmem"

The `touchmem` benchmark is a custom tool which was designed to allow fine-grained control over the exact number of page faults and the number of different pages which provoked these faults. This benchmark thus provides an excellent opportunity to evaluate the speed of the vTLB page fault intercept code in its different stages, but it does not take any multitasking effects into account. The results must therefore be interpreted with a grain of salt, as measurement artifacts due to scheduling differences may appear.

The core loop of the benchmark maps a 4 MB file into memory, accesses a chosen subset of its pages (by doing some calculations on the contents), and releases the mapping again:

```
p = mmap(NULL, 4194304, PROT_READ, MAP_PRIVATE, x, 0);
// [...]
for (i = 0; i < 1024; i++) {
  int off = i % (1024 / frac);
  MD5_Update(&C, ((char *)p)+(off << 12), calcbytes);
}
```

With its default settings (5000 iterations, all 1024 pages touched) the program therefore generates 512000 separate page faults. Figure 5.2 on page 39 shows the results for 5000 iterations with a varying number of pages touched per iteration. The implementation steps perform as expected for a single-process benchmark whose runtime is almost exclusively spent handling page faults for user space pages: the global-flag and vTLB Cache optimizations have very little effect. Batching PTE updates and disabling the page fault intercept yield the usual speedup, and eliminating the TLB flushes caused by frequent ASID reallocation brings the performance to about 33% native. The *INVLPGA* instruction cannot improve the results due to the number of page table entries. Invalidating all these with individual instructions can only hurt performance, as the numbers show.

As a second mode of operation, the benchmark locks the pages it is going to touch into memory using the POSIX `mlock()` syscall. This change improves the runtime of the earlier solutions (up to simple vTLB Caching) because page faults are intercepted only once due to the prepopulation of the guest page table. Later solutions benefit even more due to PTE Batching, as the number of round trips to the hypervisor are even more drastically reduced.

Native Linux, on the other hand, while far superior to every virtual solution, fares comparatively badly with the `mlock()` mode. While it cannot gain anything from prepopulated page tables themselves, it should experience far less page faults, too — a possible explanation might be bookkeeping work of the `mlock()` syscall, e. g. modifying the internal MMU structures for the desired pages to make sure they are not swapped out.

KVM, as we have already seen in the `forkwait` benchmark, is inferior to the vTLB implementation in terms of page fault handling. In this case it is even slower than the basic one.

5.3 Microbenchmark "sockxmit"

The `sockxmit` benchmark is another custom tool which pursues the opposite goal to the `touchmem` benchmark. It completely neglects the latencies induced by PTE addition and removal and focuses instead on the efficiency advantages gained by making entries resident in the vTLB and even the hardware TLB.

The design idea of `sockxmit` is to create pairs of processes which rapidly exchange messages between each other through some local communication channel such as a pipe or a pair of UNIX sockets. To eliminate the impact of TLB collisions, the executable code of each process is moved to a different virtual address before entering the message loop.

Figure 5.3 on page 41 demonstrates the results for the default benchmark configuration (one pair of processes, 1048576 reads / writes). The outcome for the early implementation steps without vTLB caching are as expected out of the question. Replacing the page fault intercept with the PTE update hypercall mechanism has in turn no effect at all on the runtime, which also fits into the picture. But once ASIDs are bound to their guest address spaces, the performance makes yet another leap forward.

However, using *INVLPGA* to replace the remaining ASID reallocations is, contrary to the `forkwait` example, detrimental to the overall performance. This can be explained by a close look at the numbers. While the number of consumed ASIDs (and thus forced TLB flushes) was reduced by factor 15 in the `forkwait` example, there are almost no ASID changes left to cut down on in the `sockxmit` case. Thus the overhead of issuing single *INVLPGA* instructions for each changed PTE prevails over the few saved ASID changes.

Generally speaking, though, all the aforementioned optimization steps cannot come even remotely close to native performance. With coupled ASIDs enabled there should only be two distinctive effects left between virtualized and native execution: the advantage of keeping the entries of multiple processes in the hardware TLB under virtualization (the "tagged TLB") against the disadvantage of a hypervisor round-trip for each task switch (as the intercept for write operations to CR3 is mandatory). The numbers make it crystal clear that virtualization is still too expensive: the vTLB implementation is only half as fast (i. e. it spends almost half of its time doing VM bookkeeping and performing world switches), and that despite the tagged TLB.

KVM cannot keep up with native performance either. Compared to its results in the other microbenchmarks, context switching still seems to be one of its stronger areas.

5.4 Macrobenchmark: Linux kernel compilation

While the results of the microbenchmarks suggest that certain operations are horribly slow for vTLB-based implementations, the question of implications for "general-purpose" applications is still unanswered.

In order to give a glimpse of how my vTLB performs at commonplace use-cases, I have decided to employ the de-facto standard example for such a task: the compilation of the Linux kernel itself. As most effects can already be seen on a smaller scale , I have chosen to build only the `fs` subtree for the complete spectrum and to reserve the full build (which might take hours on lower optimization levels) for a few interesting versions.

Looking at the subtree builds (see Figure 5.4 on page 42), the basic implementation experiences a 46% slowdown compared to native, i. e. it runs at 68.4% performance, which almost exactly matches other contemporary vTLB implementations (see Section 6.3). Adding the optimization for global pages and activating the vTLB cache increases performance to about 80%, but the number of intercepted page faults is still vastly higher than the number of reinjected page faults, which indicates that the vTLB is persistently losing entries which were present at some time (as these entries are not reinjected, the guest page table must already contain the entry, which in turn means that an earlier fault has prompted its creation), probably due to CR3 read-write cycles.

It comes as no surprise therefore that the introduction of PTE Batching (which includes disabling the page fault intercept and the flush semantics of CR3 read-write) cause a performance leap up to 93.5%. The ASID-related improvements have only minor impact, especially the assets and drawbacks of using *INVLPGA* instead of ASID allo-

cation seem to almost cancel each other out. However, with all optimizations activated the net performance reaches a satisfying 96%.

Doing the promised full-scale run among the three main contestants (native Linux, KVM and the fully optimized vTLB, see Figure 5.6) sees the result diminished to some extent, as the vTLB only reaches about 83% native performance. This is likely due to the creation of object files of continuously increasing size, which amplifies the impact of PTE handling. KVM is even worse though with a final value of about 80%.

As an aside, running the compilation with several jobs in parallel is an excellent way to observe the vTLB cache at work. Figure 5.5 demonstrates how the number of vTLB cache evictions increase as the number of running processes surpass the number of cache slots. Due to the pipelining of the compilation process (the creation of a single executable or object file involves several gcc backends which are started in parallel and communicate through pipes) even with four jobs the evictions start to increase the build time.

5.5 lmbench

The open-source benchmark suite *lmbench* is already rather outdated, but it is a useful tool to validate the numbers measured during the previous tests.

The main point of interest in the suite is the *lat_ctx* test which measures the latency of context switches for varying numbers of processes and varying memory footprint size per process. The test works by letting a single byte travel through each measurement process in turn (thereby guaranteeing the scheduling order), which accesses all of its allocated memory before passing the byte on. The results are shown in Table 5.1 on page 40.

The latencies under native Linux depend mainly on the size of the hardware TLB, and the numbers increase sharply once the product of processes and memory pages per process has exceeded the size of the TLB. The latencies under vTLB operation are already initially higher due to the intercepted write to CR3, but increase even more as both the size of the hardware TLB and the number of vTLB cache slots is surpassed. Especially noteworthy here is the last run, as we can see the two borders being crossed separately: For $size = 64k$, the hardware TLB is already filled by $n = 4$ processes, so $n = 8$ shows increased latencies due to TLB evictions. The vTLB is still capable to hold $n = 8$ processes though, but gives in at $n = 16$, causing the latency to shoot up from 15 to 120 microseconds.

KVM on the other hand seems to contain a vTLB caching mechanism which is similar to my vTLB implementation, as its context switches take approximately the same time to complete. Furthermore KVM seems to have more cache slots, because the context switch latencies only show a light yet steady increase with the number of processes. There are however a few extraordinarily high latencies, which might indicate a KVM cache eviction. The benchmark is not particularly accurate though, so these could also be caused by other effects, e. g. scheduling artifacts.

5.6 Summary

The benchmark results have shown that while it is easily possible to beat other vTLB implementations like those of NOVA and KVM with only minimal improvements to the basic strategy and no changes to the guest kernel, no vTLB implementation can do the impossible and achieve 100% native performance. The microbenchmarks have demonstrated that the availability of hardware support still means a speedup factor of about 3 for TLB-critical operations like e. g. context switch and process creation, but the kernel compilation example has suggested that the impact on real-world scenarios is not as significant. Depending on the actual type of workload in the virtual machine, a vTLB solution does actually offer sufficient performance.

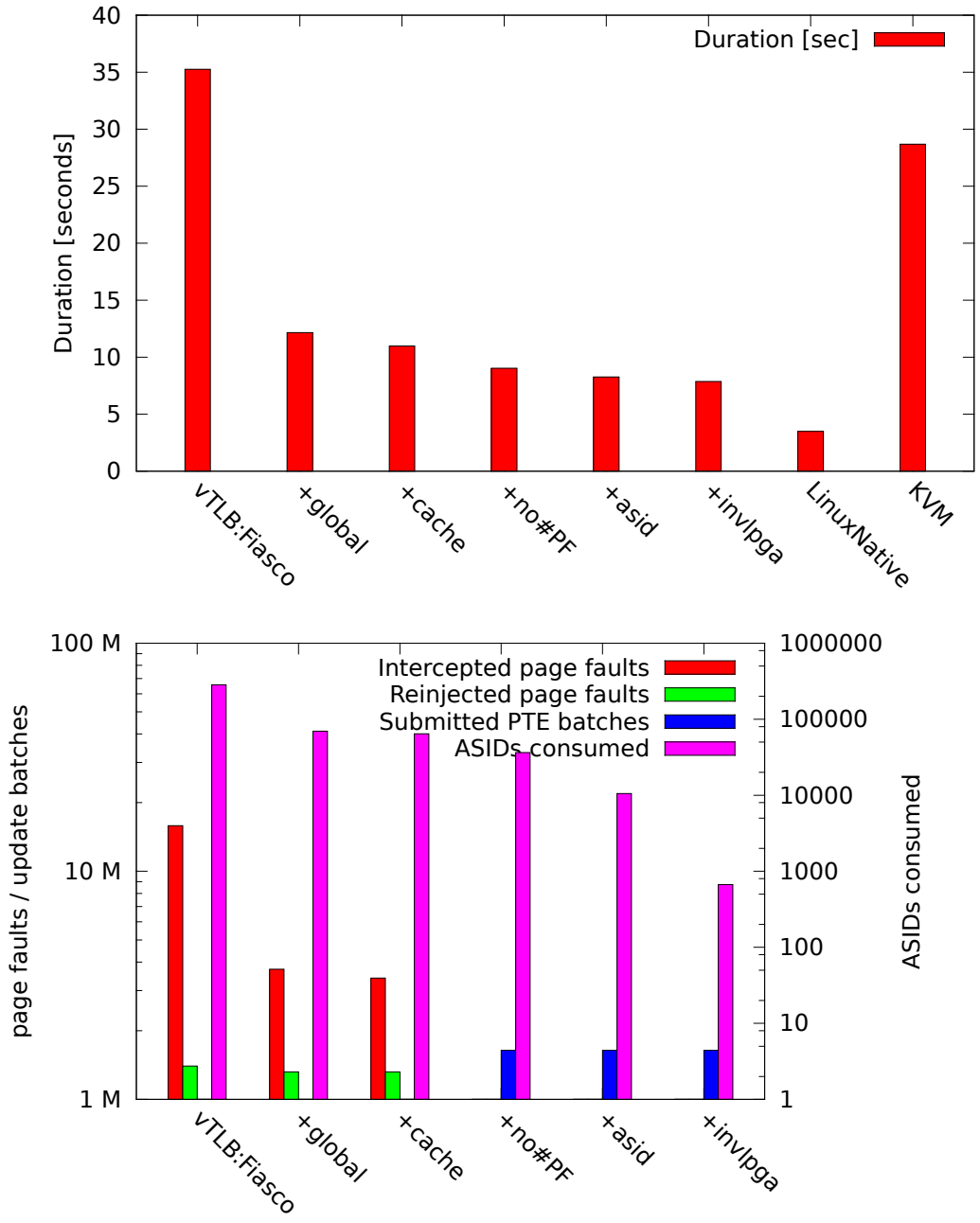

Figure 5.1: Statistics for the "forkwait" benchmark

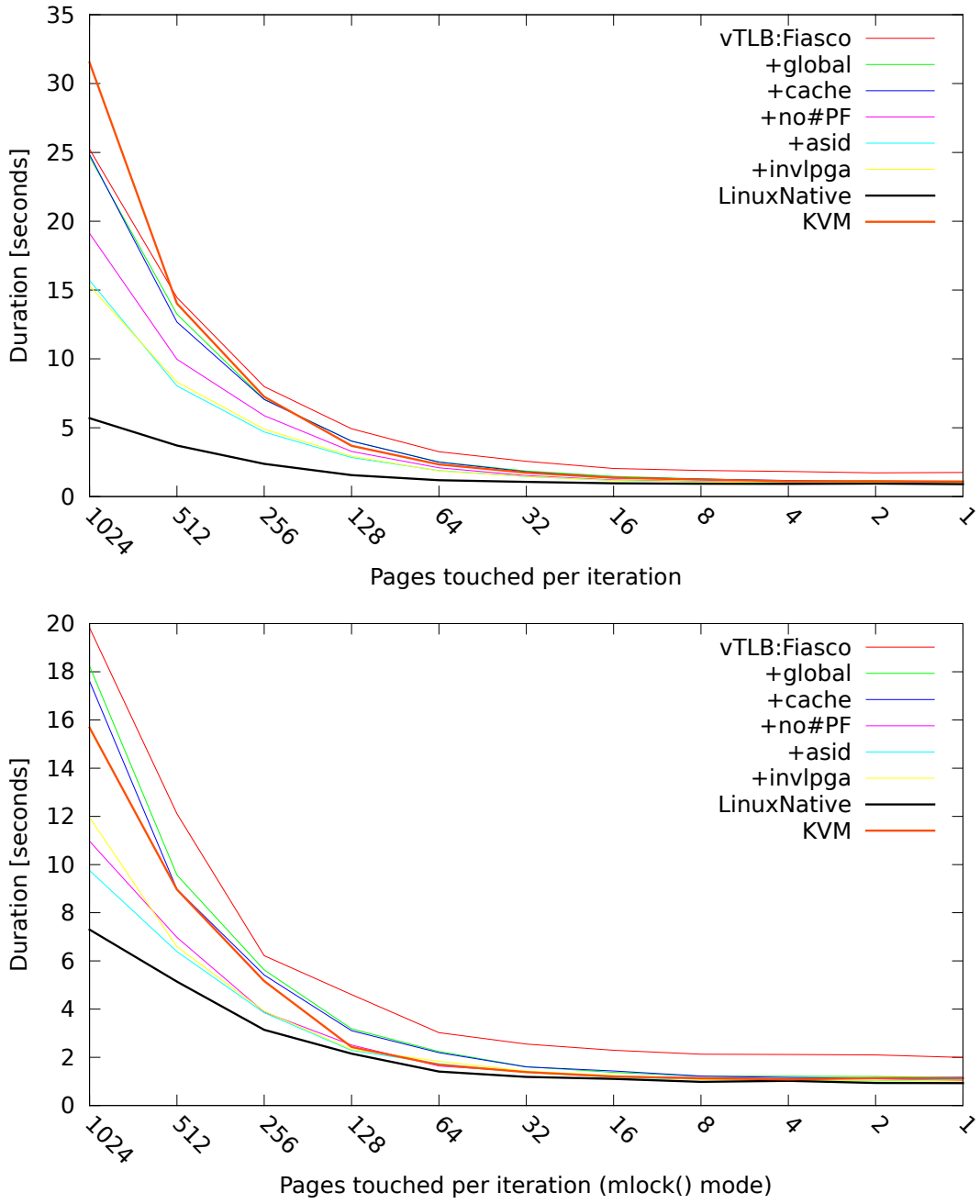

Figure 5.2: Statistics for the "touchmem" benchmark

n	native	vTLB	KVM	native	vTLB	KVM
	$size = 0k$			$size = 16k$		
2	1.15	5.24	5.30	1.20	5.31	5.57
4	1.11	4.32	6.69	1.55	5.06	6.03
8	1.26	4.30	5.38	1.83	5.28	6.22
16	1.35	50.54	5.59	2.24	77.82	6.93
24	1.59	61.91	5.72	4.61	80.33	9.57
32	1.78	61.84	5.98	6.65	80.32	11.64
64	2.57	61.40	6.55	7.69	80.76	12.80
96	3.08	61.94	7.44	7.52	80.49	117.35
	$size = 4k$			$size = 32k$		
2	1.00	5.33	5.39	1.45	4.64	5.83
4	0.97	4.25	5.57	2.45	5.96	8.43
8	1.26	4.70	5.72	2.94	6.53	7.63
16	1.69	68.47	5.95	8.89	92.23	13.59
24	1.97	68.18	6.26	11.05	91.05	16.61
32	2.41	68.99	6.34	11.28	92.58	16.39
64	4.03	70.27	8.53	11.26	91.29	16.57
96	4.78	70.44	107.47	11.19	91.15	130.72
	$size = 8k$			$size = 64k$		
2	1.08	5.49	5.43	3.72	6.21	8.27
4	1.23	4.74	5.74	3.67	7.35	9.81
8	1.54	4.96	5.77	12.09	15.21	17.72
16	1.95	70.63	6.22	17.56	120.44	24.28
24	2.32	71.72	6.74	17.42	120.19	24.12
32	3.07	73.27	7.63	17.73	120.61	24.03
64	5.69	72.24	10.67	17.74	119.33	23.90
96	5.96	73.56	110.62	17.70	119.28	24.08

Table 5.1: output of lmbench "lat_ctx" benchmark

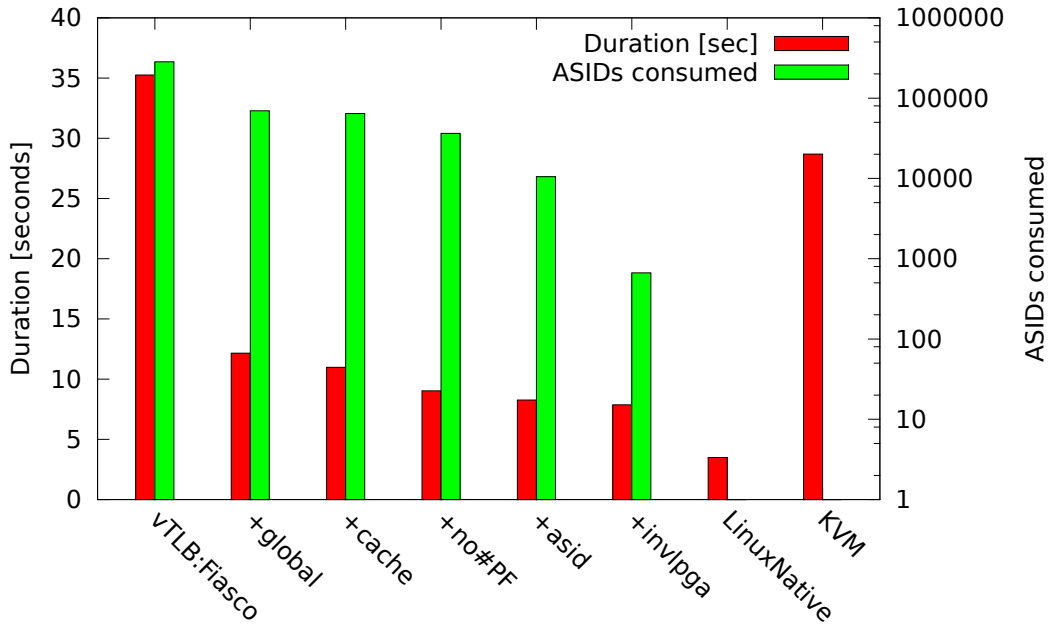

Figure 5.3: Statistics for the "sockxmit" benchmark

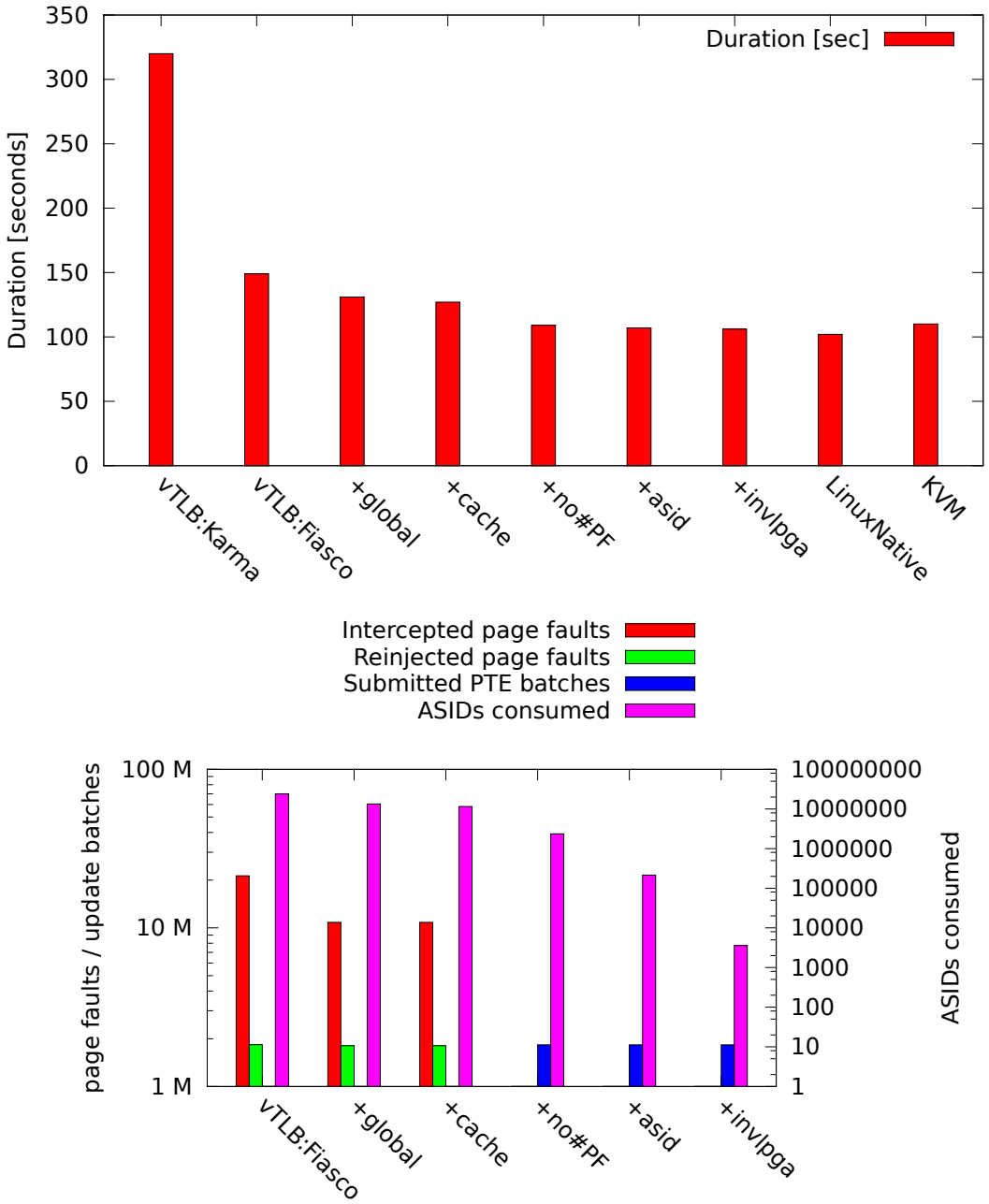

Figure 5.4: Statistics for Linux kernel compilation (`fs` subtree)

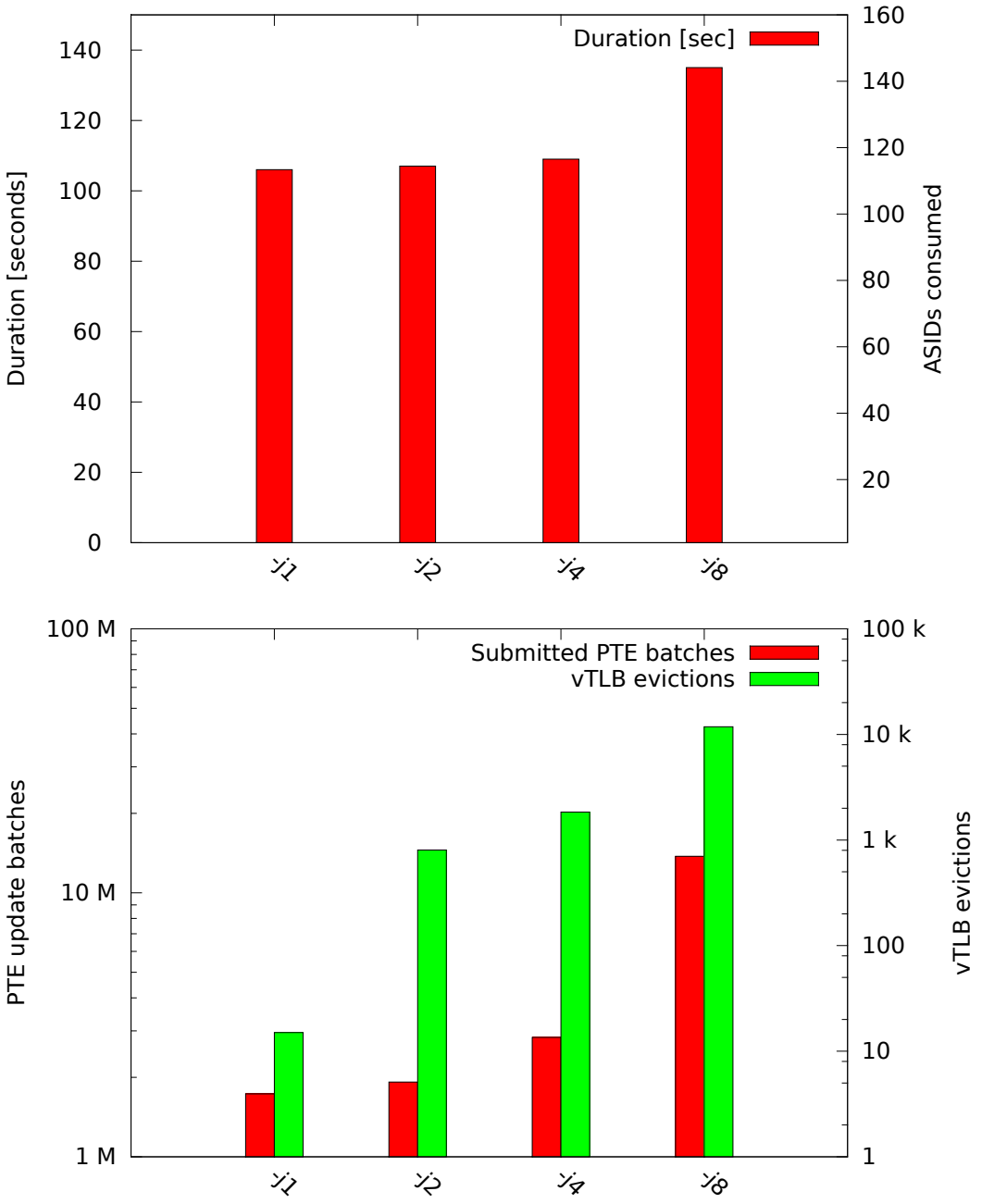

Figure 5.5: Multiple-job statistics for Linux kernel compilation (`fs` subtree)

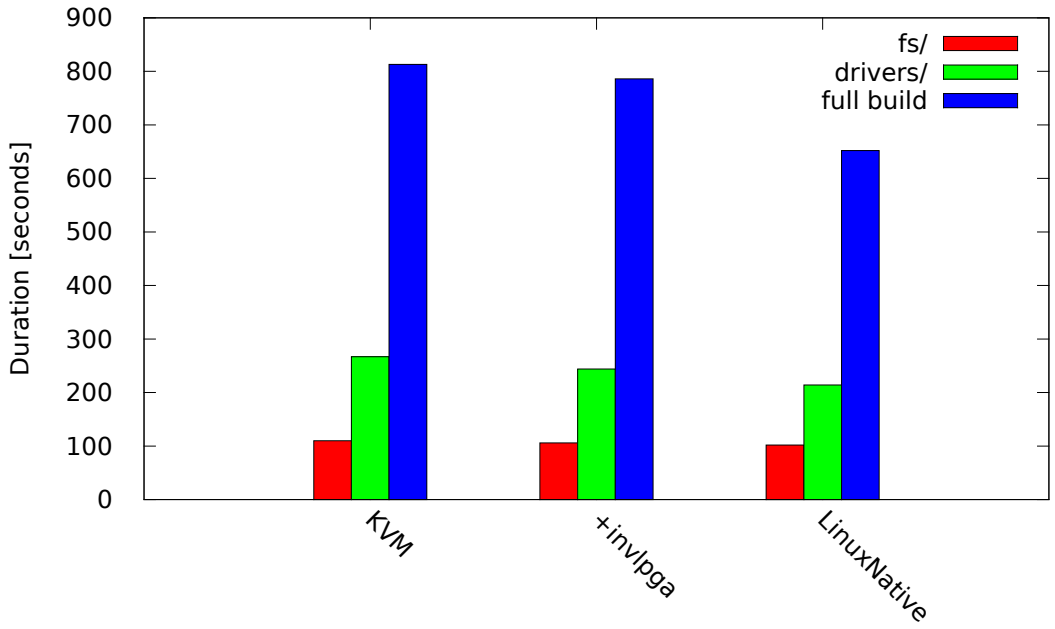

Figure 5.6: Statistics for Linux kernel compilation for different subtrees

6 Related Work

In this chapter I will give a succinct overview over other projects which have had to solve the problem of MMU virtualization. I will also describe how they attacked it and, if possible, what conclusions can be drawn from their results.

6.1 L4Linux

Linux was the first commodity operating system which was ported to run as a rehosted variant on the then young L4 microkernel. Its success (see [HHL+97]) proved that the L4 interface specification (whose latest version is available in [Gro11]) was indeed suitable and flexible enough to host a paravirtualized guest OS.

The modifications to the Linux kernel involve creating a special kernel thread with control over the whole amount of memory dedicated to the Linux guest OS. Address spaces of Linux user space processes get their memory from the Linux kernel thread instead of L4's central σ_0 memory server, and the kernel thread is also responsible for handling page faults caused by the Linux user space processes.

L4Linux presents an excellent counterpart for performance evaluations, especially of virtualization solutions. Furthermore it shares many problems with the virtualized solution, e. g. efficient implementation of process (and address space) creation, even if they both have different interfaces at their disposal.

6.2 User Mode Linux

Development on *User Mode Linux* (UML) was started around the turn of the last decade and was meant to prove that the syscall interface provided by the Linux kernel was actually sufficient to run rehosted guest OSes on. The results of the attempt were presented in [Dik00]. Although it was ultimately successful in rehosting Linux on itself, the solution is exceedingly complex and bulky. Similar to L4Linux, it requires a special "tracing thread" which is tasked with signal delivery, context switching and other administrative tasks, e. g. process creation and destruction. The performance is severely restrained by the API, as core functionality of the guest OS (executing syscalls, handling signals on an alternate stack etc.) has to be manually emulated using the all-purpose `ptrace()` process modification facility.

Although the performance of UML is incomparable to other virtualization solutions, it has served its purpose of providing an easy way to debug the Linux kernel itself very well.

6.3 NOVA & Vancouver

NOVA is a very recent project to build a Type I hypervisor, i. e. a microkernel which also takes on the role of a hypervisor (see [SK10]). It comprises of the necessary microkernel feature set (communication, scheduling, resource management), but additionally has support for register and memory virtualization. The accompanying VMM implementation *Vancouver* provides the supplementary functionality of handling the actual VMEXITs, emulating sensitive instructions and coordinating access to hardware devices.

While NOVA does not make hypercalls systematically impossible, its design principles are to provide strong isolation between unmodified guest operating systems with a minimal code base. Its developers therefore relied heavily on sufficient hardware virtualization support.

Due to this design the performance of the NOVA vTLB should match that of my earlier vTLB implementations which do not require changes to the guest kernel, and indeed it does. While NOVA reports 72.3% of native performance for a compilation of the Linux kernel on a Intel Core i7, my baseline implementation reaches about 68% on an AMD Athlon 64 X2, 78% with global guest page support.

6.4 KVM & Qemu

The kernel-based virtual machine (KVM) is a module of the Linux (and recently BSD) kernel and provides a virtualization interface to user space. It provides no emulation or device drivers itself, so it is merely a kernel extension acting as a hypervisor.

Qemu is a processor emulator which usually resorts to *binary translation*, i. e. the automatic replacement of sensitive instructions in the guest with hypercalls, to execute its guest code. Its outstanding feature is high portability, as Qemu is even capable of executing binary code which was compiled for a different than the host architecture.

On the other hand, it is also capable of acting as VMM and leveraging the capabilities of KVM to create and govern virtual machines. This combination has been used in Chapter 5 as vTLB reference implementation.

6.5 Xen

Xen is a Type II hypervisor originally developed at the University of Cambridge (see [BDF+03]) aimed at high performance. Its design favoured making explicit changes to guest kernels over being capable of hosting unmodified guests (by means of virtualization support on platforms where it was available or by resorting to binary translation otherwise). As tagged TLBs were only available on server architectures like Alpha at that time, Xen already contained a sophisticated vTLB implementation with support for PTE Update Batching. Its vTLB concept however used only a single set of page tables which were visible to the guest, but only mapped read-only. Thus every write access to them could be trapped and verified by Xen.

While this approach yields promising performance results (above 90% in macrobenchmarks), it deviates even more from native memory management than my vTLB implementation, and thus requires more changes to the guest OS code base.

6.6 VMware

The product family available from the VMware, Inc. company contains both Type I and Type II hypervisors. Their goal is to ensure maximum portability, which motivated a design philosophy which is directly opposite to Xen: VMware products use almost exclusively binary translation to run unmodified guest operating systems and emulate a fixed set of devices (hard disk controller, video card etc.) to their guests, so that guests can be seamlessly migrated between computers with different hardware configurations.

Since a few years some VMware products are also capable of leveraging the x86 hardware virtualization extensions, including MMU virtualization.

7 Conclusion

The results of this thesis have shown that while a properly modified guest OS is able to achieve almost native performance despite the delay induced by the vTLB, several microbenchmarks have clearly indicated that there are still vTLB operations that incur unacceptable delays. Generally speaking, the optimization efforts appear to have been fruitful.

On the other hand, adding the MMU paravirtualization extensions to the guest OS required intimate knowledge of its abstractions for the hardware memory management structures. Linux has made this task a little easier because there already was a paravirtualization interface, even though it became clear in later development steps that the code points chosen for Linux' paravirtualization hooks were not always optimal (or even useful).

It is likely that there is not much more room for improvement of the overall vTLB performance. However there are a few topics which might be interesting to explore and which are directly based on the work of this thesis.

Firstly, the Linux modifications also included batching page table update and identifying all exit paths from kernel to user space. L4Linux still uses a simpler, more straightforward approach at handling page tables. Porting these changes to L4Linux and evaluating whether the performance penalty of a rehosted Linux can be further decreased is definitely worthwhile.

Secondly, the possibilities of vTLB Caching with respect to realtime tasks have not been explored. In order to ensure that a virtualized OS retains the realtime scheduling semantics it had when running natively, each hypervisor/VMM intervention must either consume a constant and relatively small amount of time, or it must be possible to postpone the intervening operation until a non-realtime task is active. Even splitting the intervening code into top and bottom half (like e. g. interrupt handlers usually are) might be possible.

Peter et al. have already identified several fundamental obstacles of retaining realtime semantics in [PSLW09], among them the latencies induced by entering and leaving the VM and by long-running instructions in the guest (e. g. *WBINVD*). Due to the intercept-based nature of the vTLB implementation, any instruction which accesses a memory address with interrupts switched off can potentially become a long-running instruction because the vTLB handler cannot be interrupted.

Luckily, the vTLB implementation developed during this thesis contains only one operation which takes a variable amount of time to complete: the time needed to destroy a vTLB depends on its density, i. e. the number of allocated page directories. The implementation even contains the possibility to pin tasks into the vTLB cache, so it is easy to ensure that no vTLB destruction is required when switching to a realtime task.

However, when destroying a vTLB in order to switch in another non-realtime task interrupt delivery is suspended, because the Linux context switch code (whose write to CR3 triggers the vTLB) initially turns interrupts off. As realtime interrupts can no longer be delivered, RT tasks might still indirectly suffer from this delay.

A possible solution might be to switch to a special static (i. e. read-only) vTLB during the context switch if no cache slot is free and continuing with this vTLB until the critical section is left and interrupts are enabled again. It should be safe at that point to reenter the hypervisor in order to perform the vTLB destruction.

All the static vTLB has to fulfill is that it is always available and that it contains only pages which the guest has marked as global. This should be both secure (because no task's user space pages can be accidentally accessed during the delicate process of a context switch) and safe (because that code should not have to access user space anyway).

As the Linux kernel itself uses a kernel-only address space for its own service processes (e. g. workqueue daemons, filesystem journaling daemons), the static vTLB could be coupled with this guest CR3, thus not even using up an additional slot in the vTLB cache.

Appendix A

Code

The code base of this thesis is available at:
http://hessophanes.de/diploma_thesis/source_code.tar.bz2
SHA512: ddf5da3d86a16d806efa27e3d772315e3bf0cbdbe1edeeff400b29b53b09c633
d8b835296e8214db13aa2a8ab1110b9a72cbfa0ad52763559b363c9b3933de5b

The tarball contains unified diffs against the following repositories:

- the TU Berlin git repository of the Fiasco.OC microkernel, branch "kernel_vtlb_-jan", HEAD 2e7a215a5658fc508ee73aefd8394d3ee1640dbc:
  ssh://repo.sec.t-labs.tu-berlin.de/git/janis/fiasco.git
- the TU Dresden subversion repository of the L4Re user land, rev. 27:
  http://svn.tudos.org/repos/oc/tudos/trunk
  The user land was modified to accomodate the changes in the Fiasco.OC virtualization API — this change was not part of this thesis.
- the TU Berlin git repository of Steffen Liebergeld's VMM Karma, branch "hpet_-experiment", HEAD 1b7ca38d051107a9d67a1431a4bba17da45d7d21:
  ssh://repo.sec.t-labs.tu-berlin.de/git/janis/karma.git
- the TU Berlin git repository containing a paravirtualized variant of the Linux kernel, version 2.6.33.7, branch "rt_karma_fiasco_vtlb", HEAD 95e314e0c34a957adcb8fd84df290d3e83ee35bf:
  ssh://repo.sec.t-labs.tu-berlin.de/git/janis/karma_patched_linux.git

All aforementioned TU Berlin git repositories already contain parts of my thesis on different branches, as my work has been already partially integrated into the chair's L4/Fiasco development efforts. In all cases the commits are clearly marked as such.

Additionally, the tarball contains the source code for the microbenchmarks used for the evaluation.

Glossary

ASID Address space ID. On x86 machines with virtualization support, each VM must be assigned a non-zero ASID, which is used during its execution to store and fetch TLB entries. The host OS uses the TLB with the reserved ASID 0. Compare *TLB*.

Hypervisor An application which enforces the separation of virtual machines and switches control between them. Due to its regulative tasks it has to run at the highest privilege level. Contrast *VMM*.

L4 The successor to the L3 microkernel, developed by Jochen Liedtke. Several universities later reimplemented its ABI and "L4" became the collective name of the microkernel family.

#PF Page fault, exception vector 14 on the Intel x86 platform. If a memory location is accessed whose physical address is unknown or inaccessible because the page table entry is not present or has insufficient permissions, a page fault is triggered. The operating system is then tasked to determine whether the access was benign and should be granted (by augmenting the page table) or malicious and should be blocked (e. g. by terminating the process).

TLB Translation Lookaside Buffer. A cache for page table entries very close to the CPU. On the Intel x86 platform, both virtual address resolution and TLB entry insertion is done in hardware, and there are special instructions which trigger full or selective resets of the TLB. x86 CPUs with virtualization support contain a Tagged TLB which is able to store translation entries for multiple address spaces. Compare *ASID*.

VM Virtual Machine. A virtual instance of an operating system running under the control of a host operating system, assisted by a hypervisor and a VMM. The term does not specify whether the virtual machine is aware of its virtualized state or even capable of detecting this state. Execution of a VM is resumed by loading the contents of a VMCB using the *VMLOAD* instruction and then issuing the *VMRUN* instruction. The VM then continues to run until an event occurs which was selected for interception (see VMCB), at which point execution is returned to the host after the *VMRUN* instruction. The *VMSAVE* instruction writes the VM state back to the VMCB which can then be inspected to determine the exit reason.

VMCB Virtual Machine Control Block. A data structure in memory, divided int several areas which serve different purposes. The *State Save Area* provides storage space

for the register set, the segment selectors and other virtualized processor state variables. The *Control Area* consists mostly of bitfields which select events for interception or toggle other special VM behaviour (like flushing the TLB upon VM entry).

VMM An application which provides its virtual machine with access to hardware devices. It is called by the hypervisor whenever the VM has tried to execute a privileged instruction. The VMM can then emulate the instruction and pass control back to the VM. Contrast *hypervisor*.

vTLB Virtual TLB, also known as "shadow page table". A page table that maps guest-virtual to host-physical addresses and therefore has to be governed by the host operating system. See also *page table*, *TLB*.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev.*, 40:2–13, October 2006. 8, 28

[AMD] AMD (Advanced Micro Devices). *AMD64 Architecture Programmer's Manual*. Volumes 1 (Application Programming, #24592), 2 (System Programming, #24593), 3 (General Purpose and System Instructions, #24594), 4 (128-bit and 256-bit media instructions, #26568) and 5 (64-bit media and x87 Floating Point Instructions, #26569). 21

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003. 46

[BSSM08] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGOPS Oper. Syst. Rev.*, 42:26–35, March 2008. 12

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society. 1

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association. 45

[Gol73] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1973. 7

[Gro11] System Architecture Group. *L4 Experimental Kernel Reference Manual Version X.2 Revision 7*. Karlsruhe Institute of Technology, 2011. 7, 45

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13:238–241, April 1970. 6

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. *SIGOPS Oper. Syst. Rev.*, 31:66–77, October 1997. 6, 45

[Int] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volumes 1 (Basic Architecture), 2A and 2B (Instruction Set Reference), 3A and 3B (System Programming Guide), Intel manuals #253665-253669. 19

[Lie95] Jochen Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29:237–250, December 1995. 6

[Lie10] Steffen Liebergeld. *Lightweight Virtualization on Microkernel-based Systems.* Diploma thesis, Technische Universität Dresden, 2010. 3

[PG73] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *SIGOPS Oper. Syst. Rev.*, 7:121–, January 1973. 7

[PSLW09] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. Virtual machines jailed: virtualization in systems with small trusted computing bases. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, VTDS '09, pages 18–23, New York, NY, USA, 2009. ACM. 49

[REH07] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 4:1–4:6, Berkeley, CA, USA, 2007. USENIX Association. 2

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17:388–402, 1974. 6

[SK10] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM. 46