Design and Implementation of the VirtuOS Operating System

Ruslan Nikolaev

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

December 6, 2013

Blacksburg, Virginia

Keywords: operating systems, hypervisor, microkernel, driver isolation, profilers, perfctr,

Xen, virtualization, IOMMU, exceptionless system calls

Design and Implementation of the VirtuOS Operating System

Ruslan Nikolaev

(ABSTRACT)

Most operating systems provide protection and isolation to user processes, but not to critical system components such as device drivers or other systems code. Consequently, failures in these components often lead to system failures. VirtuOS is an operating system that exploits a new method of decomposition to protect against such failures. VirtuOS exploits virtualization to isolate and protect vertical slices of existing OS kernels in separate service domains. Each service domain represents a partition of an existing kernel, which implements a subset of that kernel's functionality. Service domains directly service system calls from user processes. VirtuOS exploits an exceptionless model, avoiding the cost of a system call trap in many cases. We illustrate how to apply exceptionless system calls across virtualized domains.

To demonstrate the viability of VirtuOS's approach, we implemented a prototype based on the Linux kernel and Xen hypervisor. We created and evaluated a network and a storage service domain. Our prototype retains compatibility with existing applications, can survive the failure of individual service domains while outperforming alternative approaches such as isolated driver domains and even exceeding the performance of native Linux for some multithreaded workloads.

The evaluation of VirtuOS revealed costs due to decomposition, memory management, and communication, which necessitated a fine-grained analysis to understand their impact on the system's performance. The interaction of virtual machines with multiple underlying software and hardware layers in virtualized environment makes this task difficult. Moreover, performance analysis tools commonly used in native environments were not available in virtualized environments. Our work addresses this problem to enable an in-depth performance analysis of VirtuOS. Our Perfctr-Xen framework provides capabilities for per-thread analysis with both accumulative event counts and interrupt-driven event sampling. Perfctr-Xen is a flexible and generic tool, supports different modes of virtualization, and can be used for many applications outside of VirtuOS.

Perfctr-Xen is based upon work supported by the National Science Foundation (NSF) under Grant CSR–AES #0720673. VirtuOS is not supported by any grant.

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Godmar Back who has helped and supported me throughout my Ph.D. studies. It would not be possible to deliver high quality results without his supervision. I especially appreciate his direction, patience, and understanding during our long discussions and meetings. I am grateful to him for spending additional time on weekends and holidays.

I would also want to thank my other committee members: Dr. Ali Butt, Dr. Bidyut Gupta, Dr. Dennis Kafura, and Dr. Eli Tilevich. Their suggestions, pointed questions, and support have helped improve my dissertation.

Suggestions and comments of the anonymous reviewers for VirtuOS and Perfct-Xen papers have helped greatly improve my work. I would also like to thank VirtuOS paper's shepherd Dr. Kevin Elphinstone for his insightful comments and suggestions.

Finally, I would like to thank graduate students from Computer Science and other departments I happened to know as well as my TA colleagues – Ali, Hari, Ioannis, Parang, and many others. I appreciate their support during my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An *operating system* (OS) is an essential part of modern computer systems. It not only manages system resources such as CPU, memory, and peripheral devices, but also provides standard interfaces for user programs to access these resources safely and concurrently. Modern operating systems support multiuser environments, provide sophisticated graphical user interfaces (GUI), and go well beyond simple resource management.

As the complexity of OS grows, it becomes crucial to build safe, secure, and reliable systems that are more resilient to bugs, less vulnerable to malicious code and provide stronger protection guarantees for their components. Despite substantial advances in operating system technology over the past several decades, building such robust systems remains a challenge.

1.1 Motivation

Modern general purpose OS require that an application runs in its own protected virtual address space. System critical data resides in the kernel's address space where it cannot be directly accessed by applications.

This mechanism protects user processes from each other and the kernel from misbehaving user processes, but falls short of protecting the system from failing kernel components. The failure of just one kernel component generally causes the entire system to crash. Major offenders are device drivers [43, 47, 54], which reportedly caused 65-83% of all crashes in Windows XP [43, 54, 73]. These components are numerous, hardware specific, often shipped by third parties with various degree of quality and are likely to be less tested than core system components due to a more limited user base. Common software bugs in drivers include improper use of resources and protocols, race conditions, and deadlocks (39% of all Linux driver bugs [84]).

To reduce the impact of faults, existing methods rely on *decomposition*, a process of splitting an OS kernel into multiple parts. The key challenge in this process is to provide *strong isolation* of the decomposed parts while retaining good *performance*, and *compatibility* with existing systems code and applications. Existing solutions run system components in separate *protection domains* to isolate system resources such as memory. For instance, microkernels move device drivers and other system critical code to user space. A number of driver protection schemes provide separate virtual memory address (VMA) spaces for kernel-mode drivers through individual per-driver page tables, or move drivers to user space. Lastly, virtual machine-based approaches rely on a hypervisor to isolate memory and other resources. With a few notable exceptions [92], most existing solutions also run system components in a less privileged mode to provide stronger isolation and limit the impact of malicious code. An Input/Output Memory Management Unit (IOMMU) [3, 7] can additionally be used in some systems to guard against misbehaving devices and provide even stronger isolation.

Retaining good performance is often challenging. Microkernels historically required effort to reduce interprocess communication (IPC) costs [50, 65]. Reducing IPC overheads is critical since components in microkernels interact solely through IPC. A single operation such as reading a file may require several IPCs to complete.

Compatibility with applications and systems code can also be challenging. Microkernels historically required effort to retain compatibility with existing applications. To provide transparency for applications, either an emulation layer was implemented [49], or a multi-server OS [45,55] was built on top of a microkernel. Alternatively, hardware virtual machines can be used to create strongly isolated domains in which to separate software components. The use of virtual machines is particularly attractive if application and kernel compatibility is desired. Similar to microkernels, virtual machines require careful consideration and optimization of their inter-VM and VM to hypervisor communication [52, 71, 85].

Although virtual machines and microkernels have nonidentical goals and motivations, they occupy the same larger design space [49, 52]. For example, modern hypervisors were used to achieve microkernel-like isolation of device drivers [42]. On the other hand, microkernels were used as a foundation for virtual machine monitors [41], or even to support the reuse of existing device drivers [64] within the confines of a virtual machine.

1.2 VirtuOS

This work explores VirtuOS[1], an alternative OS design that occupies a new point in the microkernel-VM design space. VirtuOS uses hardware-based virtualization to safely isolate larger pieces of kernel code into separate *service domains*, which house device drivers and a specific set of related with them kernel services, such as a networking stack or file system drivers. VirtuOS provides transparency to applications, excellent compatibility with existing systems code, and retains good performance especially for multithreaded programs. User processes run in a dedicated *primary domain* as shown in Figure 1.1. Since all programs communicate with the underlying kernel exclusively through system calls, the system call layer is an ideal place to dispatch program requests to different virtual machines. VirtuOS allows user processes to directly interact with service domains by means of an exceptionless system call interface [88], which was originally proposed as a substitute for the traditional, exception-based system call mechanism used in Linux. It reduces both direct costs due to CPU mode switches as well as indirect costs due to TLB and cache pollution. The use of an exceptionless model can provide substantial performance improvements, particularly

[1]First published in [75]: "VirtuOS: An Operating System with Kernel Virtualization" by Ruslan Nikolaev and Godmar Back. In Proceedings of the 24th ACM Symposium on Operating Systems Principles (Farmington, PA, USA, 2013), SOSP13, pp. 116-132.

for multithreaded server applications. In our work, we adopt the exceptionless interface to organize efficient communication between user processes and service domains. We separate system calls across primary and service domains and dispatch user process system calls to different domains. A demultiplexing mechanism allows application-based specialization and redundancy by operating multiple service domains for groups of user processes. Because programs usually do not invoke system calls directly but rather make use of the C library as an intermediary, exceptionless calls can simply be integrated into the C library, thus obviating the need to modify programs. VirtuOS design is fully transparent to programs which are unaware of the service domain's presence and do not need to be modified for VirtuOS.

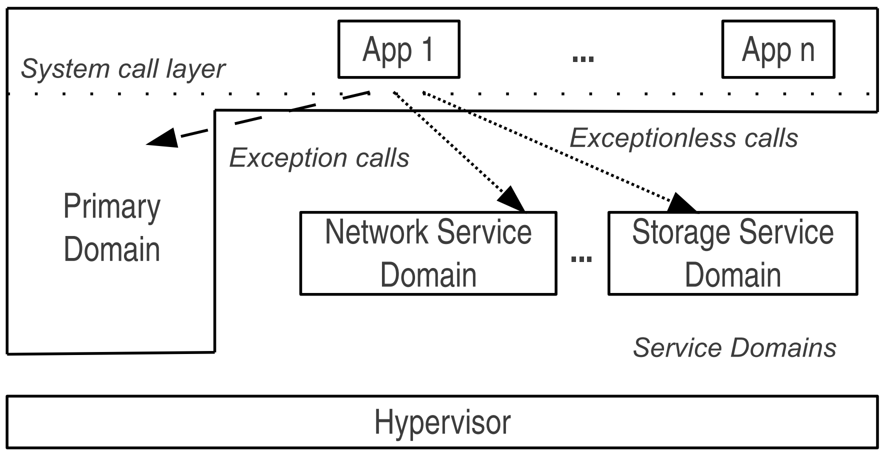


Figure 1.1: Architecture of VirtuOS.

The architecture of VirtuOS is extensible by adding new service domains. VirtuOS service domains can be created with relatively little effort. Each service domain includes a specialized *back-end* driver which facilitates handling of system calls. The back-end driver from one service domain can be reused for another by modifying domain-specific system call handler routines. This design also retains excellent compatibility with existing code; most of the handler routines simply redirect requests to the corresponding kernel functions designated to handling traditional system calls arriving through a call gate. For example, a network service domain runs a slightly adapted version of an existing OS kernel ($\sim$ 1600 lines of changes) and its back-end driver ($\sim$ 3100 lines), allowing reuse of major components such as the socket layer, TCP/IP implementation, and unmodified device drivers. We allow direct access to the hardware from service domains using PCI passthrough [9] which maps one or more physical devices to a virtual machine. We also use IOMMU [3, 7] to guard against misbehaving devices.

To adopt exceptionless system calls, we had to solve a number of unique challenges because service domains are isolated from each other and from the primary domain in separate virtual machines. File descriptors, credentials, and other relevant process information need to remain consistent across different domains. Since each domain manages file descriptors independently, VirtuOS uses a translation mechanism which allows processes to view all file descriptors as if they belonged to a single domain. Considering that system call arguments may point to user memory, we also had to implement interdomain memory sharing for each process allowing the exchange of data across different domains.

We developed a prototype which implements a networking and a storage service domain to demonstrate the feasibility of this design. We used the Xen hypervisor along with the Linux kernel for VirtuOS's domains, including modified versions of the uClibc [14], NPTL [34] and libaio [6] libraries. We tested our system with a wide range of server and client programs such as OpenSSH, mySQL, Apache, and Firefox.

We evaluated the performance for server-based workloads both under failure and non-failure scenarios. We found that VirtuOS can recover from the failure of individual service domains after restarting those domains and the processes that were using them. For network throughput tests and multithreaded transaction processing benchmarks, we found that VirtuOS meets or exceeds the performance of not only a split-driver model but also native Linux, indicating that it retains the performance benefits of exceptionless system call dispatch for those workloads. The performance loss for applications that do not benefit from the exceptionless model remains within a reasonable range.

1.3 Perfctr-Xen

Running performance-critical applications in virtualized systems is challenging because of virtualization overhead and the difficulty of making appropriate resource allocation and scheduling decisions, and VirtuOS is no exception. Commonly used performance evaluation frameworks extensively exploit profiling to allow systems and application developers to understand the performance of their applications. Such profiling frameworks rely heavily on hardware performance counters provided by modern CPUs. These counters provide information about hardware-related events such as cache misses, branch mispredictions, and many others.

Thorough performance analysis of VirtuOS mandates per-thread monitoring support for applications. Because guests running in the system are unaware of each other, hardware performance counters must be specially treated by the hypervisor to ensure that each guest sees virtualized counters that account only for events occurring for that guest. However, existing solutions for the Xen, KVM, and VMware ESX hypervisors [4, 36, 37, 71] have a number of restrictions, which include lack of per-thread monitoring, reliance on specific modes of virtualization, support for a limited number of microarchitectures, and limited monitoring mode support (e.g., profiling but not programmatic accumulative counting).

As the second contribution of this dissertation, we developed Perfctr-Xen[2], an infrastructure to provide direct access to hardware performance counters in virtualized environments using the Xen hypervisor. Our framework supports both programmatic accumulative mode counting and interrupt based profiling. In accumulative mode, the hypervisor maintains offset values and exposes them to guests allowing global performance counters to be translated into per-thread values (offsetting technique). Perfctr-Xen relies on the cooperation of guest kernel and underlying hypervisor to provide profiling tools running in the guest with access to performance counters that is compatible with the APIs used in native, unvirtualized environments, notably PAPI [23]. Consequently, frameworks and libraries that rely on PAPI can be used inside Xen and VirtuOS, such as HPCToolkit [12] or TAU [87]. To accomplish this compatibility, we modified both the Xen hypervisor as well as the guest kernel running inside

[2]First published in [74]: "Perfctr-Xen: a framework for performance counter virtualization" by Ruslan Nikolaev and Godmar Back. In Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (Newport Beach, CA, USA, 2011), VEE11, pp. 15-26.

each virtual machine. Perfctr-Xen supports all modes of guest virtualization and exploits optimizations that avoid trap-and-emulate overhead. Although our implementation focuses on Xen, the techniques we use are applicable to other hypervisors.

We used Perfctr-Xen to analyze memory copying overhead in VirtuOS by measuring the numbers of various L1/L2 cache events. Perfctr-Xen has proven to be useful in validating a hypothesis for the cause of this overhead.

1.4 Contributions

The technical contributions of this dissertation with respect to operating system design are the following:

(1) An approach to partitioning existing operating system kernels into primary & service domains, each providing a subset of system calls;

(2) A method for intercepting and demultiplexing of system calls using a user library; and the dispatching of remote calls to service domains using an exceptionless mechanism;

(3) A way to coordinate process and memory management in the primary and service domains so that applications can make transparent use of service domains.

The technical contributions with respect to performance monitoring in virtualized environments are the following:

(1) The application of an offsetting technique that allows direct access to logical per-thread

counter values from user mode while avoiding the costs associated with saving and restoring hardware performance counter registers;

(2) The optimization of guest and hypervisor communication to minimize and amortize the costs associated with their coordination, while avoiding the costs of trapping and emulating counter-related instructions;

(3) A technique for increasing the accuracy of performance monitoring by correcting for monitoring overhead.

1.5 Roadmap

Chapter 2 reviews principles of virtualization, OS design, abstractions, and performance monitoring libraries and toolkits.

Chapter 3 discusses challenges and introduces a general design outline for VirtuOS. Chapter 4 focuses on implementation aspects of VirtuOS.

Chapter 5 lays out the design and implementation of Perfctr-Xen. We discuss hardware performance counters and introduce an approach that allows us to adopt tools for virtualized environments. We evaluate Perfctr-Xen with series of micro- and macrobenchmarks.

Chapter 6 discusses evaluation of VirtuOS using Perfctr-Xen, demonstrates real-life performance with mySQL and Apache, and presents a failure recovery example.

Chapter 7 provides an in-depth review of related work in the areas of kernel fault tolerance, OS optimization and performance monitoring. We conclude our work in Chapter 8.

Chapter 2

Background

2.1 OS Abstractions

OS organize the control flow of programs, manage shared access to the CPU, memory, devices and other resources, protect core system components and programs from each other, and provide services to programs. Using a multi-tasking OS, several user programs can run concurrently on the same system. *Processes* and *threads* function as abstractions for the CPU. They encapsulate the execution state of programs and their constituents. An OS scheduler allocates and shares CPU resources for processes and threads. All processes run isolated from each other in their own *virtual memory* (VM) address space. VM provides a view of memory that is specific to each process and its constituents and which is distinct from the actual *physical memory* address space. An OS memory manager allocates physical memory and maintains *page tables* for virtual-to-physical address mappings. Core components such as the scheduler and memory manager together constitute the OS *kernel*.

Modern OS provide protection against misbehaving programs through dual mode operation. For this purpose, CPUs provide *privileged* and *deprivileged* modes of execution. The latter restricts access to privileged CPU operations (e.g., reloading page tables). All user program code executes in deprivileged mode; this helps to isolate and contain any invalid operations in a program and avoids adverse effects on the remaining parts of the system. A misbehaved program will be terminated without causing any unrecoverable damage to the system. CPUs provide support for a *mode switch* between privileged and deprivileged modes.

Most operating systems today use *files*, which are named entities comprising a sequence of bytes stored on a storage device. Files are abstractions for I/O devices. OS provide facilities to open and close files and associate open files with file descriptors. UNIX-compatible OS such as Linux adopted the concept of *"everything is a file,"* which postulates that not only regular on-disk files but also devices, network sockets and pipes are handled through the same file-descriptor based interface. File descriptors are represented as integers; the kernel maintains *file descriptor tables* to look up any file related data structures. Devices are handled using device drivers, and an OS exposes access to them through the */dev* file system.

The OS kernel provides services for programs, such as access to files, by means of a *system call* interface. Most often, the C library is used as an intermediary between a kernel and programs. The C library encapsulates most system calls into function calls to provide an architecture-independent interface.

2.1.1 Processes, Threads and Interrupts

Each process consists of one or more *threads*, which are entities that share a process's virtual memory address space, file descriptor table, and other system defined resources. Threads can be executed concurrently but are not isolated from other threads in their respective processes.

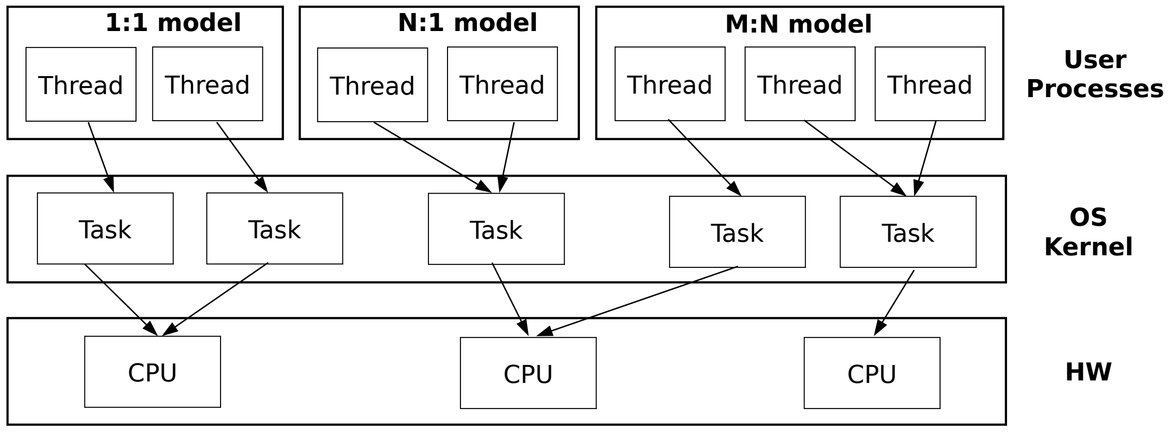


Figure 2.1: Threading models.

For each process, the kernel defines one or more *tasks*, which are abstractions that comprise the execution state of entities that can be executed concurrently and possibly on different CPUs. Since CPU resources are limited, the OS scheduler must accommodate the execution of all tasks by suspending some tasks and resuming others from time to time. This procedure is known as a *task context switch*.

Threading models determine how threads are mapped onto tasks, three examples of which are shown in Figure 2.1. For example, the N:1 model requires only one task per process since all threads within the process are mapped onto a single task. Tasks are scheduled by the OS scheduler but the OS kernel is unaware of threads mapped onto the task. Consequently, processes require a user level scheduler to perform *thread context switches*, which periodically suspend one thread and resume another. The N:1 model needs only minimal kernel support but it only allows the use of a single CPU for all threads, which limits its usability in manycore systems. Most modern OS such as Linux adopted an 1:1 model, which mandates that every thread is provided with a separate task. The 1:1 model is straightforward and makes scheduling less complex since it does not need a user level scheduler. All threads are scheduled directly by the OS scheduler. Moreover, the 1:1 model does not require special handling of blocking system calls; since all threads have separate tasks, one task's blocking will affect only one thread and not prevent other threads from running, unlike in the N:1 model. The 1:1 model has the disadvantage that it requires a mode switch for each thread context switch because context switches are only done through the OS scheduler. The M:N model was proposed to reconcile the N:1 and 1:1 models. M threads belonging to a single process are mapped onto N tasks. Since normally $M \geq N$, a user mode scheduler is still required to suspend some threads and resume others. The M:N model can be used in manycore systems and has the advantage of being capable to context switch between different threads in a process without using underlying OS kernel.

To organize asynchronous communication with devices, OS use *interrupts*. Interrupts temporarily change the flow of control of the currently scheduled task to execute directly in the context of that task. As a consequence, interrupt handlers cannot use potentially blocking operations, i.e. cause a task to be switched out. If blocking is needed, special tasks known as *kernel threads* can be scheduled by an interrupt handler to handle specific jobs. These tasks run in privileged mode and use the kernel's virtual address space. To minimize the number of kernel threads and control CPU consumption, modern kernels implement per-CPU *work queues*, which may accept jobs from any interrupt handlers and execute them in the context of dedicated kernel threads.

2.1.2 System Calls

User program tasks run in deprivileged mode but often need to execute operations implemented by the OS kernel. Since kernel code needs to run in privileged mode, a *system call* mechanism makes it possible for these tasks to invoke privileged kernel code.

Traditional system call implementations rely on an exception mechanism that transitions the processor from deprivileged to privileged mode, then executes the system call code within the context of the current task. For this purpose, modern CPUs provide special instructions to either increase or decrease the CPU's *current privilege level*. To prevent unsanctioned privilege escalation by a user program, transitions to the higher privilege level can happen only through special *call gates* which define tables with specific kernel addresses to which user code is allowed to jump. User code passes arguments in registers on most architectures. The system call service routines referenced by the table are obliged to handle any arguments containing pointers with care, taking into account that any memory addresses specified by user code may point to invalid locations. Traditional system calls impose substantial costs, both direct costs due to the cycles wasted during the mode switch, and indirect costs due to cache and TLB pollution caused by the different working sets of user and kernel code.

Exceptionless system calls [88] avoid this overhead. Instead of executing system calls in the context of the current task, a user-level library places system call requests into a buffer that is shared with kernel worker threads that execute system calls on the task's behalf, without requiring a mode switch. Effective exceptionless system call handling assumes that kernel worker threads run on different cores from the user threads they serve, or else the required context switch and its associated cost would negate its benefits. A key challenge to realizing the potential gains of this model lies in how to synchronize user and kernel threads with each other. Since application code is generally designed to expect a synchronous return from the system call, user-level M:N threading is required, so that a thread can context-switch with low overhead to another thread while a system call is in progress. Alternatively, applications can be rewritten to exploit asynchronous communication, such as for event-driven servers [89]. VirtuOS uses the exceptionless model for its system call dispatch, but the kernel worker threads execute in a separate virtual machine.

2.2 Kernel Design Models

When designing an OS, we must decide what components should be placed in its kernel. In this regard, two major approaches exist: monolithic kernels and microkernels.

2.2.1 Monolithic Kernels

Monolithic kernels run the majority of system components such as the OS scheduler, memory manager, device drivers, network stack, and file system drivers in the confines of a single large kernel as shown in Figure 2.2. Consequently, all these components run in privileged CPU mode and have direct access to underlying hardware.

Because components run in the kernel, they have unrestricted access to the CPU and any hardware resources such as I/O ports, PCI registers, and DMA buffers, which makes it easier to develop device drivers. A monolithic kernel design simplifies the interaction between various components such as bus and device drivers because components can call each other directly and share the same address space.

The downside of the design is the lack of isolation and confinement of components. For example, a buggy disk device driver may access incorrect PCI bus registers, causing other hardware and/or drivers to fail. Unrestricted access to computer resources can additionally lead to in-memory data corruption, improper access to I/O ports, etc.

2.2.2 Microkernels

Microkernel-based OS provide an alternative to the monolithic design. They move all key system components out of the kernel, leaving only core parts such as IPC (inter-process communication) inside a *microkernel* as shown in Figure 2.3. System components run in deprivileged CPU mode inside dedicated user processes.

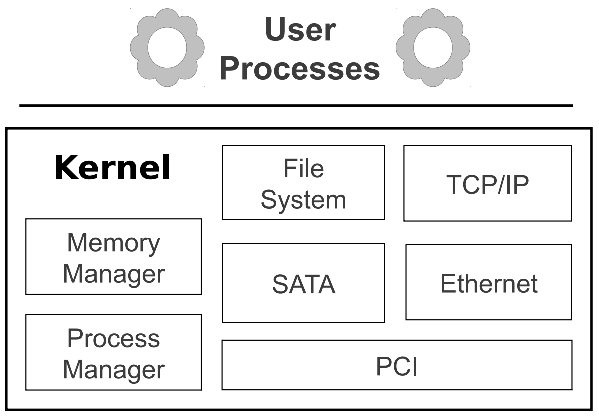


Figure 2.2: Monolithic OS.

The key advantage of the design is its ability to completely isolate different components from each other and from the underlying kernel. Moreover, failed components can be transparently restarted in a number of failure scenarios such as buffer overflow, interrupt errors, deadlocks, etc. Failures do not typically result in a complete OS crash, obviating the need for system reboot.

Access to hardware resources is more complex than in monolithic systems and requires careful handling and protection. For example, device drivers should be both protected from each other as well as from the underlying bus driver. Each driver should only be allowed to access DMA buffers, I/O ports, and PCI registers pertaining to the device(s) it serves. The kernel must map memory-mapped I/O pages and I/O ports to each process which executes a driver.

In a microkernel-based design, components communicate through IPC mechanisms such as *message passing*. If one component needs to get access to facilities provided by another component, it sends a message with a request and receives a response eventually. Microkernels

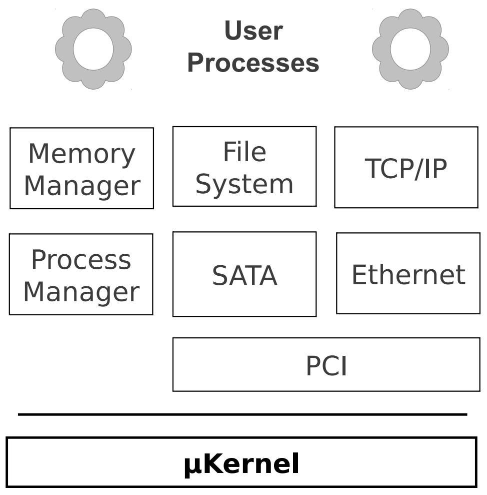


Figure 2.3: Microkernel-based OS.

often support both *synchronous* and *asynchronous* IPC calls. Because the number of IPC calls in microkernels may be substantial, it becomes crucial to reduce their cost [65].

Aside from optimizing IPC performance, microkernel-based systems often devote substantial effort to creating compatibility layers for existing system APIs, e.g. POSIX. Multiserver operating system designs pursue an alternative approach by attempting to deconstruct a monolithic kernel's functionality into separate servers running on top of a microkernel.

2.3 Virtualization

Virtualization allows multiple instances of an operating system to run on a single computer. Originally introduced for VM/370 [32], the idea has later been revitalized for modern platforms [24,83]. A *hypervisor* is a software layer that separates the virtual hardware an OS sees from the actual hardware and arbitrates access to physical resources such as CPU or memory. Among widely known hypervisors are Xen [17,29], KVM [48,61], VMware ESX [13,76], and VirtualBox [27].

Virtualization improves *isolation* and *reliability* because each OS runs independently from the others on its own virtual processors so that OS failures are contained; increases *resource utilization* as the same hardware can be used for multiple purposes; and leads to better *productivity* as large pieces of software can be preconfigured and installed very easily.

Over the years, virtualization has been adopted to solve various problems. Server applications extensively use system resources and may need specialized OS for efficient resource management. For this, researchers proposed "virtual appliances," [86] which are software bundles containing preconfigured packages with their own OS aimed at simplified distribution and deployment. They run along with other virtual appliances and general purpose OS on a single machine. Commercial providers of infrastructure as a service (IaaS) solutions rely on virtualization to provide business solutions for server consolidation. Virtualization has also been proposed as a means of utilizing manycore platforms efficiently [15]. Finally, it has been shown that hypervisors can be utilized to provide better security and robustness for operating systems [42,64,81].

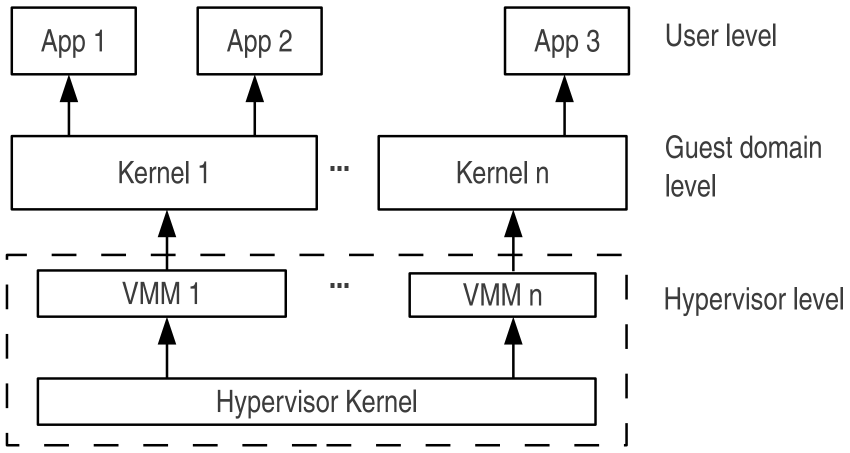


Figure 2.4: Architecture of a Type-I hypervisor.

2.3.1 Hypervisor

In this work, we use Xen [17], a widely known Type I hypervisor [60] that allows execution of virtual machines in *guest domains*. Type I hypervisors run directly on hardware and should be distinguished from Type II hypervisors which run inside some existing operating system known as *host OS*. In Figure 2.4 we present a diagram showing the different layers of a Type I hypervisor system. The hypervisor itself forms the lowest layer, which consists of the hypervisor kernel and Virtual Machine Monitors (VMMs). The kernel has direct access to the hardware and is responsible for resource allocation, scheduling and sharing. A VMM is a layer responsible for virtualizing and providing resources to a given operating system. Depending on the hypervisor's design, VMMs may or may not exist as entities separate from hypervisor kernel. VMWare ESX [13] is one example where a clear distinction between hypervisor and VMM is put in place.

2.3.2 CPU Virtualization

To run several OS on a single machine, a hypervisor needs to provide guests with virtual CPUs, entities that allow guests to have access to CPU resources while allowing the hypervisor to schedule different operating systems on the available physical CPUs or cores. Complete and transparent CPU virtualization traditionally required architectural support. Popek and Goldberg formulated necessary and sufficient conditions for virtualization in [78]. Among the most important characteristics are *fidelity*, *performance*, and *safety*. Fidelity requires that the virtual environment in which programs run should be indistinguishable from the real hardware. To achieve performance, a substantial amount of instructions need to be executed directly by hardware and without any additional hypervisor handling. Finally, safety requires that the hypervisor has full control of all system resources. *Sensitive instructions* used by guests need to be trapped and emulated if they could affect the correctness of the hypervisor's behavior. Trap-and-emulate is a technique for intercepting CPU instructions and executing them using special (hypervisor) handlers. Some architectures such as the original IA32 architecture do not meet described criteria, as they do not allow to intercept all necessary instructions when executed in deprivileged mode [82]. To overcome this problem, *binary translation* and *paravirtualization* techniques were proposed.

In a binary translation [11] mechanism, machine code is being processed by a translator program. The translator program allows prefetching, inspecting and special treatment of all relevant instructions. It was popularized by the VMware hypervisor, which first introduced binary translation for the purpose of IA32 platform virtualization.

Paravirtualization, a concept pioneered by the Xen hypervisor, is the adaptation of guest operating system code to a particular hypervisor. Paravirtualization is a series of adaptations inside the guest OS kernel code that avoids the use of untrappable instructions, reducing emulation and management costs and yielding better performance. In a broader sense, paravirtualization has become synonymous with any adaptation in a guest OS that facilitates its execution on top of a virtual machine monitor.

Xen's original paravirtualized guest implementation relied on the 4-ring protection hierarchy present in IA32 processors, executing the guest kernel in Ring 1, while user processes executed in Ring 3 and the privileged hypervisor code executed in Ring 0. The MMU's segmentation facilities protected guest kernel code and data from user processes, but guest kernels had to be adapted to successfully run in this mode. The advent of the x86_64 64-bit extension of IA32 removed segmentation, making this approach less practical. Instead, processor vendors introduced hardware virtualization extensions such as VT-x and AMD-V that allowed the execution of unchanged guest kernels. The first generation of these extensions supported the safe virtualization of the CPU by introducing a VMM mode distinct from the mode in which privileged guest kernel code executes; later generations added support for MMU virtualization via nested paging without resorting to shadow page tables. Well known hypervisors such as Xen and VMware now provide support for hardware virtualization.

Currently Xen implements hardware machine containers (HVM) to exploit those hardware virtualization capabilities. In plain HVM mode, a guest OS is entirely unaware that it is not running on physical hardware. As a result, an unchanged guest kernel (such as an out-of-box image of a commercial OS) can be executed. Xen reintroduced the ability of guests to more efficiently communicate with and use facilities of the underlying hypervisor in its PVHVM mode and will continue this approach in future releases using a new design referred to as PVH mode. VirtuOS uses PVHVM mode in its service domains; we plan to use PVH mode once it is available. Perfctr-Xen supports PVHVM as well as traditional paravirtualization, thus making an in-depth performance analysis of VirtuOS feasible.

2.3.3 Memory Virtualization

Conventional OS manage memory assuming that they have unrestricted control over physical memory. Hypervisors introduce special adaptations to guest OS allowing to run multiple guest domains at the same time. For this purpose, hypervisors distinguish *virtual*, *physical* and *machine* memory address spaces. The latter is actual memory exposed by hardware. Physical and virtual memory are guest specific and have their conventional meanings with the only difference that physical addresses correspond to memory exposed by the VMM rather than hardware. Hypervisors and CPUs provide support for virtual-to-physical, physical-to-machine and virtual-to-machine address translation. Since address translation is done at page granularity, hypervisors and guests refer to the physical and machine addresses using their physical and machine *frame* numbers.

Fully-virtualized guests are completely unaware of the underlying hypervisor. Thus they do not know anything about machine frames and treat physical frames as if they represented actual hardware exposed memory. Guests create page tables containing virtual-to-physical memory mappings. However, these page tables cannot be used by the hardware since physical frames do not correspond to actual memory locations. The hypervisor will trap any attempt to load such page tables and substitute them with *shadow page tables*. Shadow page tables are created and maintained by the hypervisor; their entries are created on demand and contain virtual-to-machine memory mappings generated by the hypervisor from the corresponding guest page tables and the physical-to-machine (P2M) hypervisor translation tables. The hypervisor manages shadow page tables transparently; they are not exposed to guests. Because maintaining shadow page tables is costly, CPU vendors introduced *nested page tables*. This technique eliminates shadow page tables and trapping by providing an additional hardware translation layer. The first layer is for regular OS page tables containing virtual-to-physical mappings. The second layer is for physical-to-machine mappings and is managed by the hypervisor.

In the case of paravirtualization, guests are adapted to assist the hypervisor with memory management. Paravirtualized guests still have physical frames but unlike in the fully-virtualized case, they have direct access to the P2M translation table, allowing them to perform physical-to-machine translation. Because of this, no shadow page tables are necessary, and guests create page tables containing entries with virtual-to-machine mappings. The hypervisor will verify the consistency of page tables before loading them to make sure that the range of machine addresses is valid.

2.3.4 I/O Virtualization

Apart from virtualizing CPU and memory, a hypervisor needs to organize device sharing. Because hypervisors are typically small and do not include third party components, device drivers are usually executed in a host OS (Type II hypervisors), specialized driver domains, or a dedicated privileged domain (e.g., Dom0 in case of Xen). Usually, the privileged domain has unrestricted access to hardware. For all other guest domains, 3 options exist: (1) Complete emulation of devices. In this case, the privileged domain will emulate devices and guests will access them as if devices were some real piece of hardware; (2) a Split driver model in which drivers consist of two parts that interact with each other using inter-domain communication facilities. One side (normally a privileged domain) runs a *back-end* driver which provides shared access to physical devices, and the other side (other guests) runs a *front-end* driver to expose access to that device; (3) Direct I/O in which a device is disabled in the privileged domain and reassigned to a specified guest.

Split Driver Model

To share devices and provide safe access to guest OS efficiently, some hypervisors such as Xen introduced specialized drivers that rely on inter-domain communication facilities to achieve good performance. The hypervisor includes an inter-domain memory sharing API that allows domains to share memory, which is accessed through guest kernel extensions. Event-channels provide a inter-domain signaling facility. Using an event channel, a guest domain can trigger an interrupt in another domain such that a corresponding IRQ handler is executed. Split drivers use these facilities to implement I/O device ring buffers to exchange data; batching of concurrent requests may be used to minimize the number of interrupts. The split driver model also provides the option of placing drivers in their own, dedicated driver domain [42, 85]. Though VirtuOS does not use a split driver model, it uses both the shared memory facilities and event channels provided by Xen.

Direct I/O

The most recent generation of processors and chip sets provides safe I/O virtualization facilities, which are supported by modern hypervisors such as Xen. These facilities allow guest domains direct access to a piece of hardware in a manner that allows the safe assignment of devices to domains. Devices and drivers are isolated so that failure in either of them cannot adversely affect other domains. VirtuOS relies on two key facilities: PCI passthrough and IOMMU.

In Xen, PCI passthrough allows hypervisors to assign access to devices residing on a PCI bus to guests other than Dom0, without requiring emulation or paravirtualization. These guests can access those devices using unchanged drivers and they have full ownership of those devices. To avoid collisions, the privileged Dom0 domain excludes those PCI addresses from the I/O space it manages. To make PCI passthrough safe, the physical presence of an Input/Output memory management unit (IOMMU) is required, along with software support from the hypervisor. An IOMMU remaps and protects addresses and interrupts used by memory-mapped I/O devices. It thus protects from devices and drivers that might make improper use of DMA or interrupts. VirtuOS relies on safe I/O virtualization facilities to execute unchanged driver code in service domains.

Self-virtualizing Hardware

Hardware designers have proposed the use of self-virtualizing devices [79], which define virtual devices that share same physical device. The device's firmware is aware of the presence of multiple guest domains and provides support for multiplexing its features to them. This facility could be used in VirtuOS to share the same physical device by multiple service domains. Therefore, VirtuOS should be able to benefit from this technology as it emerges.

2.4 Sharing vs. Isolation

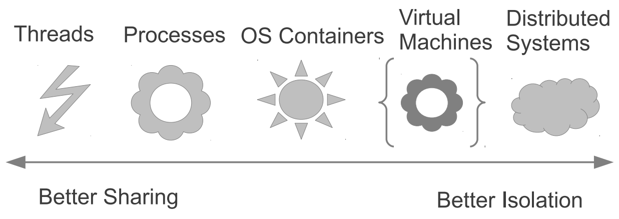


Figure 2.5: Spectrum of sharing vs. isolation.

As operating systems evolved, different approaches with respect to sharing and isolation were proposed. In Figure 2.5, we show the relationship of different abstractions with respect to these two parameters. Threads provide virtually no isolation but they share almost all crucial program resources such as virtual memory space and file descriptors. While processes provide good isolation with respect to memory accesses and file descriptor management, they share the same file system, process ID namespace, system devices, etc. In a corporate environment, it is useful to share computer system resources between all employees. This leads to operating system sharing and the need to set up independent and isolated environments for different employees. LxC (Linux Containers) [90] is an example of a container based operating system which is able to provide different user environments through process filtering, changing root file system and networking isolation. LxC, however, does not allow to run unrelated operating systems, shares same device drivers, memory management, CPU scheduler and other OS critical components. Since it does not provide complete isolation of core components, it may sometimes be limiting for people who want complete isolation of user environments. Virtual Machines solve this problem by isolating entire operating systems from each other and leaving only fundamental mechanisms (e.g., network sockets) for communication between various operating systems running on the same host. Finally, distributed systems run their components on different physical hosts, and each host may use any of the above-mentioned methods to organize isolation or sharing within a host. One example of these are cloud systems which provide specific services to remote users such as IaaS (Infrastructure as a Service) or SaaS (Storage as a Service).

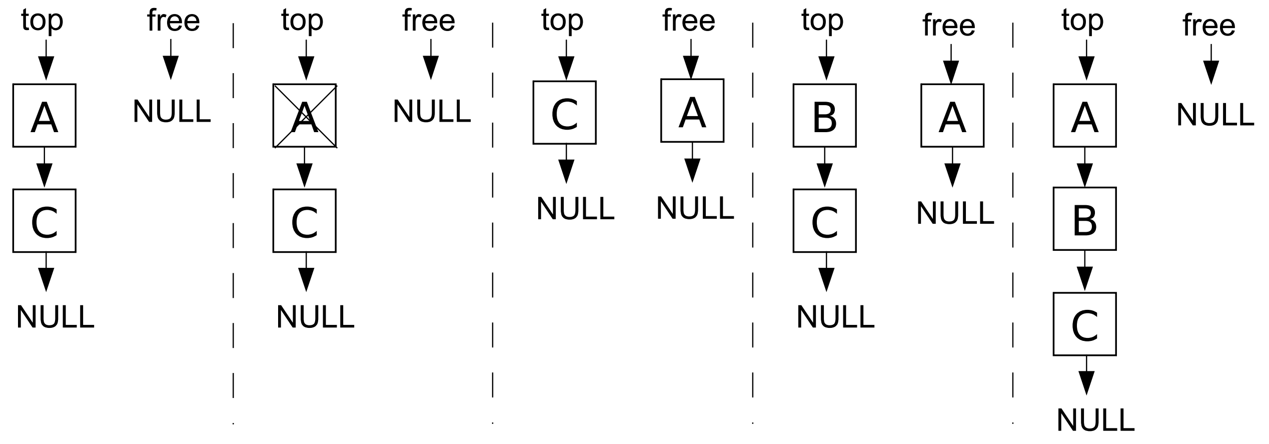
2.5 Lock-free Algorithms

To support safe concurrency without locks, systems designers proposed to use lock-free algorithms. Lock-free algorithms are typically built using CPU instructions that provide consistent memory updates across different CPUs, such as atomic read-modify-write (RMW) operations and/or memory barriers. Unlike spinlocks, lock-free algorithms are not affected by preemption since all threads collaborate to guarantee that at least one thread is making progress. If a thread is preempted in the middle of an operation, another thread will *help* complete the operation to avoid blocking and maintain consistency. Lock-free algorithms that also prevent starvation are known as wait-free algorithms. Despite their advantages, they are hardly used in practice due to their implementation complexity.

Modern CPUs support RMW through either load-linked/store-conditional (LL/SC), or compare-and-set (CAS) instructions. LL and SC instructions are always used in pairs. LL loads a value from memory, and SC stores an updated value only if the previous value remains unchanged. CAS atomically loads a value from memory, compares it with the *old value* specified by the program and replaces it with the *new value* if the old value matches the value stored in memory. The CAS operation returns a boolean status indicating if the old value was updated. LL/SC provide a more general facility than CAS. Despite this, LL/SC is often used to simply implement CAS and is not exposed directly to the programmer [93] because of memory access restrictions that most CPUs impose in between a pair of matching LL and SC instructions.

2.5.1 ABA problem

Because multiple threads may access elements simultaneously, care must be taken to ensure that the deallocation of an element by one thread will not result in memory access violations by other threads that access the same element. In garbage-collected environments, such memory access violations are impossible since the element will not be deallocated if some context can still reference it. For environments with manual memory management, such as an OS kernel, the simplest approach is to maintain a list of free elements that can be reclaimed in the future. In this approach, memory access violations are impossible since memory is not actually deallocated by the OS memory manager.



(a) Initial state (b) A is removed (c) A is in the free list (d) B is inserted (e) A is reclaimed

Figure 2.6: ABA problem.

When elements are added to a list of free elements, other threads need to identify that the data structure has changed. For example, consider operations that insert and remove elements from a list-based LIFO queue (stack). Typically, any stack implementation maintains a pointer to the topmost element as shown in Figure 2.6a. In our example, we also maintain a list of free elements which is empty initially. One thread reads a pointer to the topmost element (Line 2, Algorithm 2.1) before being preempted. Another thread removes A and adds it to the free list as shown in Figures 2.6b and 2.6c. Then the second thread also inserts B (Figure 2.6d), reclaims and inserts A (Figure 2.6e). Note that when the first thread is resumed, the CAS operation which updates the topmost pointer (Line 5, Algorithm 2.1) will erroneously succeed even though the stack has changed and A is no longer the same element. This is known as the *ABA problem*.

Algorithm 2.1 A LIFO queue operation.

```
1: repeat
2:     $x \leftarrow top$
3:     ...
4:     new topmost element is going to be $x\_new$
5: until $CAS(addr = \&top; old = x; new = x\_new)$
```

To solve the ABA problem either LL/SC, or CAS with tagging needs to be used. LL/SC will prevent the ABA problem because SC (used in lieu of Line 5, Algorithm 2.1) succeeds only if the specified A memory location is unchanged since LL (used in lieu of Line 2, Algorithm 2.1) was executed. To solve the ABA problem with CAS, each value is associated with a unique *tag* which is incremented each time a corresponding value is updated. The tags need to be updated atomically together with their associated memory values. If the values are pointers, a double-width operation is necessary. For the x86_64 architecture, *cmpxchg16b* corresponds to the double-width CAS operation.

2.6 Performance Monitoring

2.6.1 Hardware Event Counters

Modern CPUs provide access to hardware event counters through programmable performance monitoring registers. Such registers can be programmed to count events of interest, such as cache accesses or misses or branch mispredictions. The registers may be read-only or read-write. System software controls whether registers are directly accessible to non-privileged user applications or whether accesses must be done in privileged mode from system code. The number of registers is typically smaller than the set of event types that can be counted, requiring that the user select a subset of events of interest. The set of event types is specific to a given microarchitecture and frequently changes as the microarchitecture evolves. Most performance monitoring registers can be set up to trigger interrupts when they overflow. This mechanism is useful to perform statistical profiling using sampling intervals that contain a constant number of the events of interest in each interval.

2.6.2 Frameworks and Libraries

A cornucopia of performance monitoring frameworks and libraries exist. A representative set of examples is shown in Table 2.1. These frameworks and libraries differ in their functionality, level of abstraction, granularity of monitoring, and the interfaces upon which they rely.

At the bottom level, *low-level performance counter libraries* provide a thin layer over the

Table 2.1: Characteristics of Performance Monitoring Libraries and Frameworks.

Framework	Type	Monitoring	Direct access	Interfaces Used
perf_events	low level	Per thread	Yes	ioctl, mmap, sysctl
perfctr	low level	Per thread	Yes	ioctl, mmap, dev
perfmon	high, low level	Per thread	No	syscalls, mmap, signals
PAPI	high level	Per thread	Yes (w/perfctr)	perfctr, perfmon, etc
OProfile	profiler	System wide	N/A	oprofilefs
XenoProf	profiler	System wide	N/A	oprofilefs
PerfExplorer	profiler	Per thread	N/A	PAPI
HPCToolkit	profiler	Per thread	N/A	PAPI

facilities provided by the hardware, which typically does not hide architecture-specific event types from the user. These systems consist of kernel extensions and a corresponding user library. The kernel extensions implement operations that require privileged access, such as reprogramming counters or setting up interrupt handling and forwarding interrupt notifications to processes. The user library provides an API for accessing event counters.

Events may be counted globally (system-wide), or per thread. Most libraries support both modes, although some (e.g., OProfile [30]) provide only global counting. Global recording of events has the advantage that it can account for vertical interactions at all levels of the software stack as well as for horizontal interactions with other programs, such as local servers. On the other hand, global profiling makes it difficult to separate events of interest from unrelated system activities or noise.

Per-thread counting provides each thread with its own logical set of performance counters, just like each thread has its own logical set of machine registers. To implement per-thread accounting, the performance counter framework needs to maintain per-thread state, which is updated on each context switch.

Some libraries (e.g., perfctr [77] and perf_events [46][1]), allow a thread to directly read the physical performance monitoring register in user mode in order to obtain fine-grained and precise information about events during its execution, while other libraries (e.g., perfmon [40]) require a system call to obtain access to this information. Those systems that provide direct access must either maintain the thread's logical value in the physical register while the thread is scheduled, or they must place a correction value (offset) in an agreed-upon location (such as a memory-mapped area in the thread's address space) that allows a thread to compute its logical value based on the value read from the physical register.

High-level performance counter libraries such as PAPI [23] provide a layer that hides microarchitecture specific event types behind a uniform, higher-level API. Performance profilers such as TAU [87] and HPCToolkit [12] in turn are built on top of higher-level performance counter APIs. These profilers statistically sample events and present cumulative statistics to the user that relates these events to instructions and functions in the user code, with appropriate references to the source code if available.

The choice of framework influences the accuracy of measurements [97]. Bypassing high-level APIs in favor of low-level APIs typically reduces the measurement error, but requires architecture-specific code. The accuracy depends also on which events should be counted (user mode only vs. user and kernel mode events [97]).

[1]perf_events was previously known as perf_counter

To compensate for the limited number of performance counter registers, some frameworks (perfmon, PAPI) support event multiplexing. This technique applies only a subset of the desired event sets during subsections of a program's execution, then scales the results to extrapolate their values for the entire program.

2.6.3 Support in Virtual Machines

Support for performance event monitoring depends on the type of virtualization being used. There is limited support in Xen 4.2 for selected microarchitectures when hardware-assisted virtualization is used. In this approach, accesses to performance monitoring registers are intercepted via traps. However, the set of architectures supported is far smaller than that supported by PAPI, and it lacks support for traditional paravirtualized domains. A similar approach for the KVM hypervisor is described in [36, 37].

The XenoProf [71] framework extends the OProfile [30] system-wide profiler to allow per-domain (e.g., per guest) profiling in Xen, even when hardware-assisted virtualization is not used. However, XenoProf does not allow independent and simultaneous profiling of different domains. VMware's vmkperf [4] is yet another tool for performance analysis but with no profiling support.

Aside from these approaches, most current virtual machines disallow access to the performance monitor registers by the guest operating systems, thus preventing widely used low-level libraries such as perfctr and perf_events from being used. Consequently, users cannot benefit from high-level performance profilers such as TAU PerfExplorer and HPCToolkit to diagnose the performance of their applications when executing on top of virtual machines.

Chapter 3

Design & Architecture of VirtuOS

VirtuOS's primary goal is to explore opportunities for improved isolation of kernel components by exploiting virtualization without significant compromises in performance. We present the architecture of our system in Figure 3.1. Our design partitions an existing kernel into multiple independent parts. Each part runs as a separate *service domain* which represents a light-weight subset of kernel functionality dedicated to a particular function. A single, *primary domain* is dedicated to core system tasks such as process management, scheduling, user memory management, and IPC.

Service domains do not run any user processes other than for bootstrapping and system management related to a domain's function. Our design attempts to minimize their numbers, because the primary sole task of service domains is to handle requests coming from the user processes managed by the primary domain.

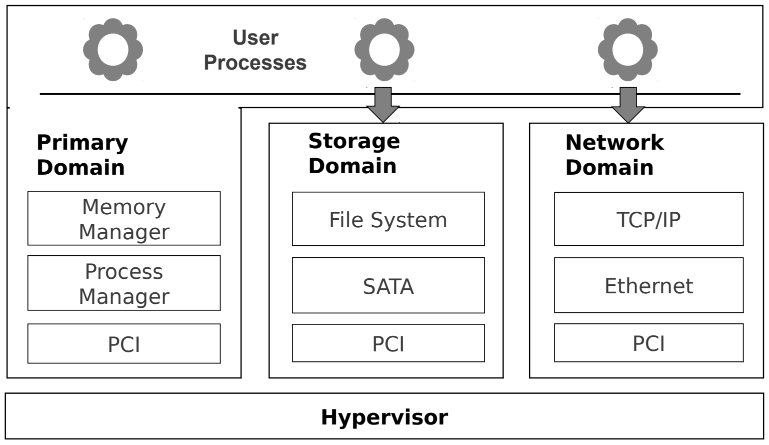


Figure 3.1: Decomposition in VirtuOS.

Our design does not assume that there is only one primary domain in which user processes run; it could be extended to support multiple user environments, each having its own set of primary and service domains as shown in Figure 3.2. In that scenario, conventional hypervisor approaches such as split model drivers or self-virtualizing hardware could be used to multiplex different service domains. This makes VirtuOS design applicable in traditional virtualization applications.

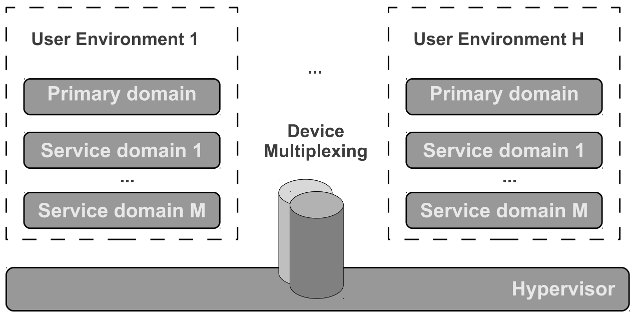


Figure 3.2: Generalization of VirtuOS design.

3.1 Failure Model

Our design goal is to contain faults originating in service domains only; we assume that the primary domain is stable enough to perform core system tasks such as task scheduling, IPC, and memory management. Service domains execute code that is potentially less reliable, such as drivers and corresponding software stacks.

VirtuOS provides recovery guarantees with respect to failures caused by software errors and transient hardware faults. Such hardware faults include invalid DMA memory accesses or interrupt signaling errors. Service domain failures can be contained as long as the hypervisor itself enforces isolation. We designed all communication between the primary domain and all service domains such that it can tolerate arbitrary service domain failures, which implies careful handling of any requests or responses from those domains. If a failure is detected, the service domain must be restarted using standard hypervisor utilities. Service domain failures affect only those processes that have started using the failed domain; only these processes may need to be restarted.

We believe this model provides advantages compared to the alternative of rebooting the primary domain or the entire machine, especially when multiple service domains are used for different hardware components, such as separate network interfaces or storage devices, which may be accessed by disjoint subsets of processes. In addition, server applications such as web servers are often designed to use multiple OS processes, which can be restarted if failures occur.

3.2 System Call Design

VirtuOS processes communicate with service domains at the level of system calls. We refer to system calls destined for a service domain as remote system calls, whereas local system calls are directly handled by the primary domain. A modified C library contains all necessary infrastructure to transparently demultiplex local and remote system calls and forward remote calls to service domains. Since most programs and libraries do not execute system calls directly, this design enables source and binary compatibility with dynamically linked binaries.

Since most POSIX system calls use file descriptors, we tag file descriptors with their corresponding domain. As an example, a *socket(2)* system call for the `AF_INET*` families may be forwarded to the networking service domain, which creates a socket and assigns a file descriptor number in return. Any subsequent operation such as *read(2)* or *write(2)* will then be dispatched to the service domain from which it originated. To avoid the need for coordination between service domains and the primary domain in assigning file descriptor numbers, VirtuOS's C library translates user-visible file descriptors to domain file descriptors via a translation table. This design also allows the implementation of POSIX calls (e.g., *dup2()*) that assume that a process has control over its file descriptor space, but it requires that the user-level library interpose on all file descriptor related calls.

3.2.1 Exceptionless Dispatch

To dispatch system calls to a service domain, we initially considered the use of a traditional exception-based mechanism. We discarded this design option because every system call would then have required exiting the virtual machine in order to use an interdomain communication facility such as event channels, in addition to the costs associated with the mode switch itself. Instead, we adopted the concept of exceptionless system calls described in Section 2.1.2.

The implementation of exceptionless system calls across virtual machines poses a number of unique challenges that are not present when applying this method to optimize the performance of native system calls as done in FlexSC [88]. In FlexSC, the kernel worker threads handling the system calls can easily obtain direct access to a client thread's address space, file descriptor tables, credentials, and POSIX signal settings. Such direct access is impossible in VirtuOS since the kernel worker threads reside in a different virtual machine. Our implementation addresses these differences, which requires the primary domain to communicate essential information about running processes to the service domains, which we describe in Section 4.2.

We present the detailed architecture of VirtuOS in Figure 3.3. A *front-end* driver in the primary domain kernel communicates with *back-end* drivers in the service domains to inform them when processes are created or have terminated. The front-end and back-end drivers also cooperate to establish the necessary shared memory areas between user processes and service domains. Each process creates two such shared areas for each service domain: (1) one area to hold the request queue for outstanding system call requests, and (2) an area used as a temporary buffer for system calls that transfer user data. We also added a component to the underlying hypervisor to keep track of service domain states and domain connection information, which is necessary for domain initialization and recovery after failures.

The request queue for outstanding system call requests consists of fixed-sized system call entries, which contain the information needed to execute a system call. System call entries are designed to be small so they fit into a single cache line. When a system call is dispatched to a service domain, a system call entry is added to the request queue shared with that domain. We adapted a FIFO/LIFO lock-free queue with ABA tagging [56, 72, 94] to ensure that the request queue, as well as other shared queues, can be accessed safely by both the user process and the service domain. We discuss implementation of queues in Section 4.5.

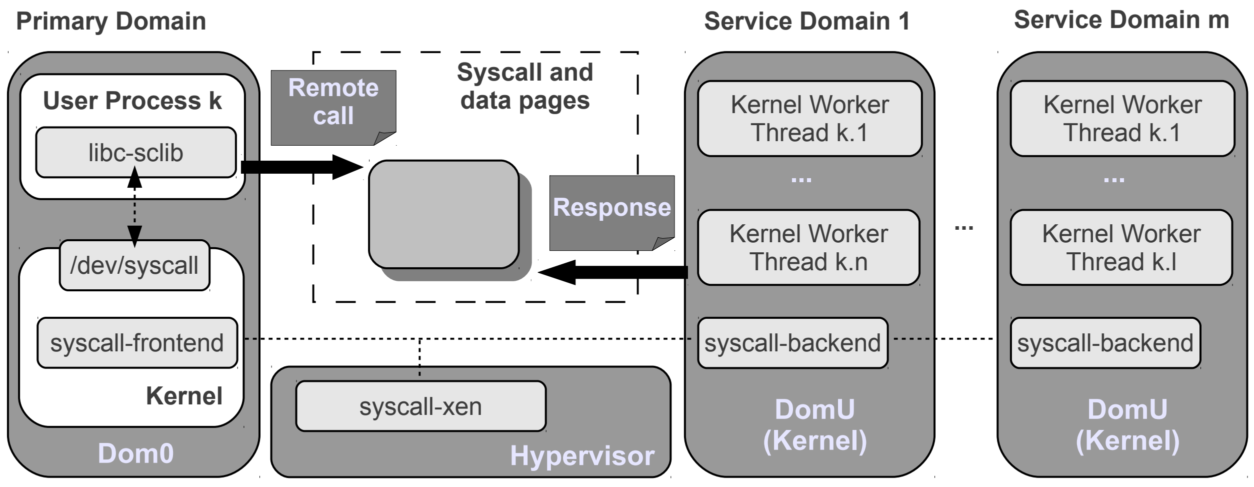


Figure 3.3: Detailed architecture of VirtuOS.

3.2.2 Managing User Memory Access

System call arguments may refer to user virtual addresses, such as when pointing to buffers a system call should copy into or out of. Our design uses a copying-based strategy in which the user process copies data into or out of a temporary buffer of memory shared with the service domain. A shared memory region is mapped in a continuous virtual address space region in the user program and in the service domain. During initialization, user programs use a pseudo */dev/syscall* device to create a memory mapping for this region. The primary domain's front-end driver, which services this device, then communicates to the back-end drivers within the service domains a request to allocate and grant access to pages that can be mapped into the user process's address space.

A special purpose allocator manages the allocation of buffers for individual system calls from this shared region. For simplicity, our implementation uses a simple explicit list allocator, along with a per-thread cache to reduce contention. If the free list does not contain a large enough buffer, the region can be dynamically grown via the */dev/syscall* device. The region can also be shrunk, although the allocator used in our current prototype does not make use of this facility. Since the POSIX API does not impose limits on the sizes of memory buffers referred to in system call arguments, we split large requests into multiple, smaller requests to avoid excessive growth.

Although this design for managing memory access requires an additional copy, it sidesteps the potential difficulties with designs that would provide a service domain with direct access to a user process's memory. Since a user process may provide any address in its virtual address space as an argument to a system call, direct access would require coordination with the primary domain's physical page management. Either pages would have to be pinned to ensure they remain in physical memory while a system call is in progress, which would severely restrict the primary domains flexibility in managing physical memory, or the primary domain would have to handle page faults triggered by accesses from the service domain, which would require complex and expensive interdomain communication. Moreover, it would be difficult to identify which pages should remain mapped to the service domain.

3.2.3 Polling System Calls

Polling system calls such as *select(2)*, *poll(2)*, or Linux's *epoll_wait(2)* operate on sets of file descriptors that may belong to different domains. These calls block the current thread until any of these file descriptors change state, e.g. become readable, or until a timeout occurs. We implement these calls using a simple signaling protocol, in which the primary domain controls the necessary synchronization. We first partition the file descriptor set according to the descriptors' target domains. If all file descriptors reside within the same domain (local or remote), a single request to that domain is issued and no interdomain coordination is required. Otherwise, we issue requests to each participating service domain to start the corresponding polling calls for its subset. Lastly, a local system call is issued that will block the current thread. If the local polling call completes first, the primary domain will issue notifications to all participating service domains to cancel the current call, which must be acknowledged by those domains before the call can return. If any remote call completes first, the corresponding service domain notifies the primary domain, which then interrupts the local call and starts notifying the other domains in the same manner as if it had completed first. The C library combines results from all domains before returning from the call.

3.3 Thread Management

VirtuOS uses separate strategies to schedule user-level threads issuing remote system call requests and to schedule worker kernel threads executing in service domains.

3.3.1 User-level Thread Scheduling

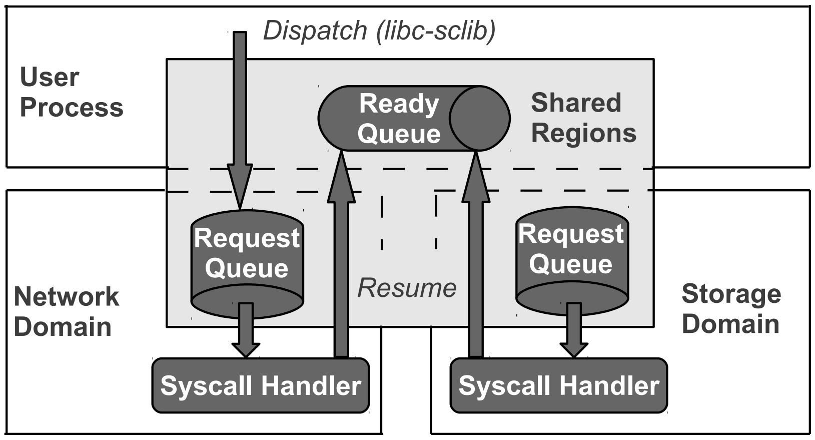


Figure 3.4: Sharing ready and request queues.

To retain the performance benefits of exceptionless system call dispatch, we must minimize the synchronization costs involved in obtaining system call results. VirtuOS uses a combination of M:N user-level threading and adaptive spinning to avoid the use of exceptions when possible. The threading implementation uses a single, per-process ready queue, which resides in memory that is shared with all service domains. Like the per-service domain request queues, it is implemented in a lock-free fashion to allow race-free access from the service domains. Figure 3.4 shows the relationship between a process's ready queue and its per-domain requests queues. When a system call request is placed into a service domain's request queue, the issuing user-level thread includes a pointer to its thread control block in the system call entry. If other user-level threads are ready to execute, the current user-level thread blocks and performs a low-overhead context switch to the next ready user-level thread. Once the service domain completes the system call request, it directly accesses the user process's ready queue and resumes the blocked thread based on the pointer contained in the system call entry.

If there are no ready user-level threads after a system call request is issued, a user-level thread spins for a fixed amount of time, checking for either its system call request to complete or a new thread to arrive in the ready queue. Otherwise, it blocks the underlying kernel thread via a local system call, requiring an exception-based notification from the remote service domain when the request completes. Such spinning trades CPU capacity for latency. We determined the length of the spinning threshold empirically so as to maximize performance in our benchmarked workloads, as we will further discuss in Section 6.2. Alternative approaches include estimating the cost of exception-based notification in order to optimize the competitive ratio of the fixed-spin approach compared to an optimal off-line algorithm, or using an on-line algorithm based on sampling waiting times, as described in the literature [59].

Service domains perform range checking on any values read from the shared area in which the ready queue is kept, which is facilitated by the use of integer indices. Thus, although a failing service domain will affect processes that use it, sharing the ready queue will not cause faults to propagate across service domains. Though request queues are not shared across service domains, service domains must perform range checking when dequeuing system call requests to protect themselves from misbehaving processes; moreover, the domain subsequently subjects any arguments contained in these requests to the same sanity checks as in a regular kernel.

3.3.2 Worker Thread Scheduling

Each service domain creates worker threads to service system call requests. Our approach for managing worker threads attempts to maximize concurrency while minimizing latency, bounding CPU cost, maintaining fairness, and avoiding starvation.

We create worker threads on demand as system call requests are issued, but always maintain one spare worker thread per process. Once created, a worker thread remains dedicated to a particular process. This fixed assignment allows us to set up the thread's process-specific data structures only once. Although handling a system call request from a process must be serviced by a worker thread dedicated to that process, all worker threads cooperate in checking for new requests using the following strategy.

When a worker thread has completed servicing a system call, it checks the request queues of all other processes for incoming requests and wakes up worker threads for any processes whose request queue has pending requests. Finally, it checks its own process's request queue and handles any pending requests. If no request is pending in any queue, the worker thread will continue to check those queues for a fixed spinning threshold. If the threshold is exceeded, the worker thread will block.

To avoid excessive CPU consumption due to having too many threads spinning, we also limit the number of worker threads that are spinning to be no larger than the number of virtual CPUs dedicated to the service domain. In addition, our design allows a service domain to eventually go idle when there are no requests for it. Before doing so, it will set a flag in the domain's state information page, which is accessible via a read-only mapping to user threads. User-level threads check this flag after adding system call requests to a domain's request queue, and initiate a local system call to wake up the service domain if needed. To avoid a race, the service domain will check its request queue one more time before going idle after it has set the flag.

Chapter 4

VirtuOS Implementation

This chapter describes the specific implementation strategy used in our VirtuOS prototype, as well as difficulties and limitations we encountered.

4.1 Effort

We implemented our system based on the Linux 3.2.30 kernel and the Xen 4.2 hypervisor. We use the same kernel binary for the primary domain as for service domains. On the user side, we chose the uClibc library, which provides an alternative to the GNU C library (glibc). We selected uClibc after concluding that glibc's code generation approach and extensive system call inlining would make comprehensive system call interposition too difficult. Unlike when using exceptionless system calls for optimization, we require that all system calls, no matter at which call-site, are dispatched to the correct service domains. We replaced Linux's pthread

Table 4.1: New or modified code in VirtuOS.

Component	Number of Lines
Back-end Driver	3115
Front-end Driver	2157
uClibc+NPTL	11152
libaio	2290
Linux kernel	1610
Xen	468
Total:	20792

library with our own to provide the M:N implementation described in Section 3.3.1.

Table 4.1 summarizes our implementation effort with respect to new or modified code. The relatively small number of changes needed to the Linux kernel shows that our virtualization-based approach enabled vertical slicing with comparably little effort.

4.2 Service Domain Implementation

Service domains handle remote system calls by redirecting requests to the corresponding functions of the service domain kernel. To keep our implementation effort small, we reused existing facilities and data structures whenever possible. For each process, we create a shadow process control block (PCB) that keeps track of that process's environment, including its credentials and file descriptors, as well as additional information, such as the location of the regions containing system call arguments and data. The data structures contained in the shadow PCB track the data structures referenced by the process's PCB in the primary domain. For instance, the primary domain must keep the service domains' view of process's credentials and capabilities in sync. Any changes are propagated to all service domains, followed by a barrier before the next system call is executed.

We reuse the existing infrastructure to manage per-process file descriptor tables, although the shadow PCB's table includes only those file descriptors that were created by the service domain for a given process. Since VirtuOS's C library translates user visible file descriptors before issuing system call requests, a service domain can directly use the file descriptor numbers contained in system call arguments. Reusing the existing data structures also ensures the correct semantics for the *fork(2)* system call, which requires duplication of a process's file descriptor tables and credentials.

We exploit the existing mechanisms for the validation of user-provided addresses, i.e. the copy_from_user(), copy_to_user(), etc. functions that are already used in all system call handler functions when accessing memory. We set the executing worker thread's user memory range (via set_fs()) to accept only addresses within the shared region used for this purpose, as discussed in Section 3.2.2. Since all data pages are in contiguous regions in both the user process and the service domain, our system call library can pass addresses that are valid in the service domain as system call arguments, which are computed by applying an offset relative to the region's known base address.

4.3 Process Coordination

The primary and the service domains need to communicate when new processes that make use of a service domain are created or destroyed. The front-end driver in the primary domain informs the back-end driver of each new process, which then performs initial memory setup and prepares for receiving system call requests.

The drivers communicate using Xen's interdomain communication facilities. We use Xen event channels as an interdomain interrupt facility and Xen I/O ring buffers to pass data between domains. We establish two separate event channels. The first one is used for adding and removing processes, as well as expanding and shrinking the shared memory regions used by processes to communicate with the service domain. We use a request/response protocol on this channel. We use a second event channel for all notifications coming to and from service domains. We use two separate ring buffers, one for each direction. We modified the ring buffer implementation to allow for one-directional communication (i.e., requests with no responses) to support asynchronous requests that do not require a synchronous response. Xen's ring buffers support batching so that only one interdomain interrupt is required when multiple requests or responses occur concurrently.

To ensure consistency, our back-end driver executes process management related requests from the front-end sequentially. For instance, a request to remove a process must be processed after the request to add that process. To avoid having to maintain dependencies between requests, we use a single-threaded implementation, which may create a bottleneck for some workloads.

We use two optimizations to speed up process management. First, we preallocate shared memory regions. When a new process is created, the front-end driver will attempt to use a preallocated region to map into the new process. If successful, it will issue a request to obtain grant references for the next region in anticipation of future processes. Otherwise, it must wait for the previous request to complete before installing the shared region. Second, we avoid waiting for responses where it is not necessary in order to continue, such as when shrinking the shared memory region.

4.4 Recovery Toolkit

The front-end driver exposes a */dev/syscall_service* device interface for system management and recovery. A user-mode *syscall* tool can start, stop, or clean-up after crash system call service for a given service domain. We currently require a user to restart a service domain and load its back-end driver manually. We expect that this functionality can be later integrated into the tool to make recovery easier.

4.5 Lock-free Queues

To support safe concurrency for ready-, request- and other queues, we adapted lock-free FIFO/LIFO queues with ABA tagging [56,72,94], as introduced in Section 2.5.

4.5.1 Memory Management

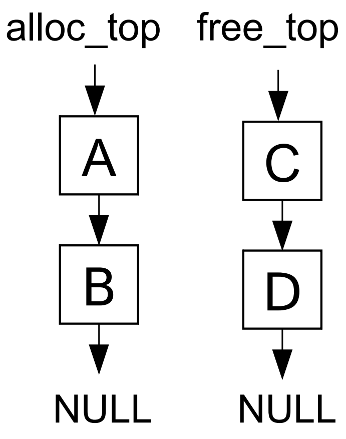
Since queues need to be accessed by both service domains and user processes, queues reside in the memory regions shared between user processes and service domains. To simplify the implementation, shared regions contain per-queue fixed-size arrays of elements. For each queue, we maintain a list of free elements as discussed in Section 2.5.1. Initially, the free list contains all elements from the fixed-size array.

4.5.2 Pointers and Error Validation

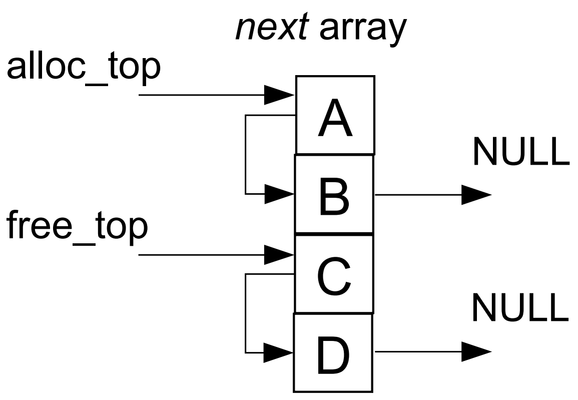
In VirtuOS, service domains are prepared to see queues in inconsistent states if user programs misbehave. Since the queues are mapped to different memory address ranges in service domains and user processes, they use indices into the fixed-size array of elements instead of pointers so that they can easily check their validity. To prevent the ABA problem, each index has an associated tag. For this purpose, we defined a *dptr* structure which consists of an index and tag.

4.5.3 LIFO Queues

Each LIFO queue (stack) consists of a list of allocated elements and a list of free elements. The list of allocated elements uses the *alloc_top* variable to point to the top of the list as shown in Figure 4.1a. Elements are stored in a per-queue fixed-size array of elements (*ents*). Since each element in a singly-linked list needs to point to its successor, LIFO queues also have a fixed-size array of indices (*next*). This array records the indices of successor elements as shown in Figure 4.1b.



(a) Queue lists



(b) Array of indices

Figure 4.1: Lock-free LIFO queue.

To enqueue an item (Algorithm 4.2), an element is taken from the top of the free list (Line 2), initialized with the item's data (Line 5) and inserted at the top of the allocated list (Line 6). To dequeue an item (Algorithm 4.3), an element is taken from the top of the allocated list (Line 2), the element's data is copied (Line 5), and the element is inserted at the top of the free list (Line 6). To check that a queue is non-empty (Algorithm 4.4), we simply verify that the top of the allocated list does not equal NULL.

To insert an element into the allocated or free list, we use the procedure shown in Algorithm 4.5. This procedure reads the current top value (Line 3), initializes the successor's index for the allocated element with it (Line 4), and atomically changes the top value with *index* (Line 5) if top did not change since it was read. If top changed, the entire procedure

Algorithm 4.2 Enqueue operation for LIFO queues.

```
1: function ENQUEUE($q$: queue *, $data$: entry_data *)
2:     $i \leftarrow Remove(next = \&q \rightarrow next; top = \&q \rightarrow free\_top; max = MAX\_ENTRIES)$
3:     if $i = STATUS\_ERROR$ or $i = STATUS\_EMPTY$ then
4:         return $STATUS\_ERROR$
5:     Initialize $q \rightarrow ents[i]$ with $data$
6:     $Insert(next = \&q \rightarrow next; top = \&q \rightarrow alloc\_top; index = i)$
7:     return $STATUS\_SUCCESS$
```

Algorithm 4.3 Dequeue operation for LIFO queues.

```
1: function DEQUEUE($q$: queue *, $data$: entry_data *)
2:     $i \leftarrow Remove(next = \&q \rightarrow next; top = \&q \rightarrow alloc\_top; max = MAX\_ENTRIES)$
3:     if $i = STATUS\_ERROR$ or $i = STATUS\_EMPTY$ then
4:         return $i$
5:     Copy $q \rightarrow ents[i]$ to $data$
6:     $Insert(next = \&q \rightarrow next; top = \&q \rightarrow free\_top; index = i)$
7:     return $STATUS\_SUCCESS$
```

Algorithm 4.4 Checking if a LIFO queue is non-empty.

```
1: function CHECK($q$: queue *)
2:     $cur\_top \leftarrow (q \rightarrow alloc\_top)$
3:     if $cur\_top.index = NULL\_INDEX$ then
4:         return $STATUS\_EMPTY$
5:     else if $cur\_top.index \geq MAX\_ENTRIES$ then
6:         return $STATUS\_ERROR$
7:     else
8:         return $STATUS\_SUCCESS$
```

will be repeated. To remove an element from a list, we use the procedure shown in Algorithm 4.6. This procedure reads the current top value (Line 3), gets the successor's index (Line 8), and atomically updates the top value with the successor's index (Line 9). The method returns the previous top value (i.e., the index of the removed element).

Algorithm 4.5 Inserting an element into a LIFO queue list.

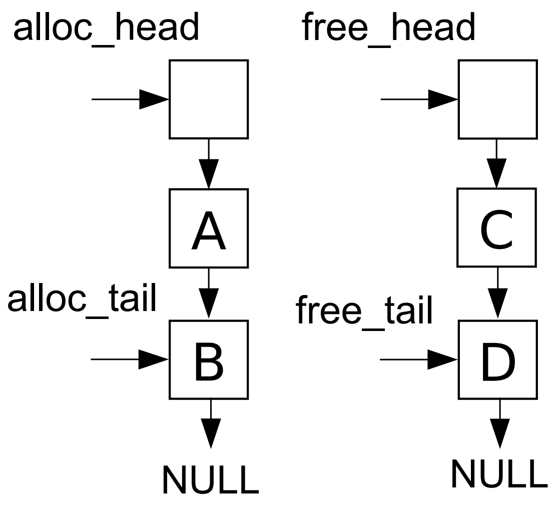
```
1: function INSERT(next: integer[], top: dptr *, index: integer)
2:     repeat
3:         $cur\_top \leftarrow *top$
4:         $next[index] \leftarrow cur\_top.index$
5:     until $CAS(addr = top; old = [cur\_top.index, cur\_top.tag];$
               $new = [index, cur\_top.tag + 1])$
```

Algorithm 4.6 Removing an element from a LIFO queue list.

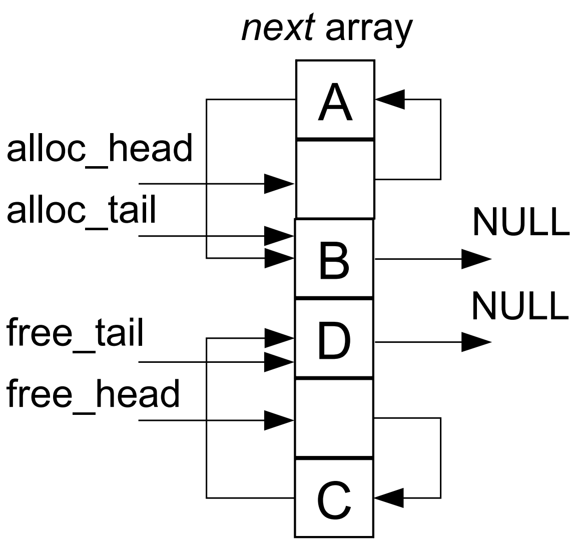
```
1: function REMOVE(next: integer[], top: dptr *, max: integer)
2:     repeat
3:         $cur\_top \leftarrow *top$
4:         if $cur\_top.index = NULL\_INDEX$ then
5:             return $STATUS\_EMPTY$
6:         else if $cur\_top.index \geq max$ then
7:             return $STATUS\_ERROR$
8:         $next\_top \leftarrow next[cur\_top.index]$
9:     until $CAS(addr = top; old = [cur\_top.index, cur\_top.tag];$
               $new = [next\_top, cur\_top.tag + 1])$
10:    return cur_top.index
```

4.5.4 FIFO Queues

FIFO queues are similar to LIFO queues except that they require separate head and tail variables for the allocated and free lists. Each list starts with a sentinel element as shown in Figures 4.2a and 4.2b. FIFO queues also require ABA tags for the array of indices, which are in addition to tags for the *alloc_head*, *alloc_tail*, *free_head* and *free_tail* variables.



(a) Queue lists



(b) Array of indices

Figure 4.2: Lock-free FIFO queue.

To enqueue and dequeue items, we follow the procedures shown in Algorithms 4.7 and 4.8. Since both the head and the tail of FIFO queues need to be updated consistently using only CAS operations, FIFO queues require threads to *help* each other to complete all updates. For all operations that update queues, tails are updated using separate CAS operations.

Algorithm 4.7 Enqueue operation for FIFO queues.

```
1: function ENQUEUE($q$: queue *, $data$: entry_data *)
2:     $ent \leftarrow Remove(next = \&q \rightarrow next; ents = \&q \rightarrow ents; head = \&q \rightarrow free\_head;$
           $tail = \&q \rightarrow free\_tail; max = MAX\_ENTRIES)$
3:     if $ent.index = STATUS\_ERROR$ or $ent.index = STATUS\_EMPTY$ then
4:         return $STATUS\_ERROR$
5:     Initialize $q \rightarrow ents[ent.index]$ with $data$
6:     return $Insert(next = \&q \rightarrow next; tail = \&q \rightarrow alloc\_tail;$
                 $index = ent.index; max = MAX\_ENTRIES)$
```

Algorithm 4.8 Dequeue operation for FIFO queues.

```
1: function DEQUEUE($q$: queue *, $data$: entry_data *)
2:     $ent \leftarrow Remove(next = \&q \rightarrow next; ents = \&q \rightarrow ents; head = \&q \rightarrow alloc\_head;$
           $tail = \&q \rightarrow alloc\_tail; max = MAX\_ENTRIES)$
3:     if $ent.index = STATUS\_ERROR$ or $ent.index = STATUS\_EMPTY$ then
4:         return $ent.index$
5:     Copy $ent.data$ to $data$
6:     return $Insert(next = \&q \rightarrow next; tail = \&q \rightarrow free\_tail;$
                 $index = ent.index; max = MAX\_ENTRIES)$
```

Algorithm 4.9 Checking if a FIFO queue is non-empty.

```
1: function CHECK($q$: queue *)
2:     while True do
3:         $cur\_head \leftarrow (q \rightarrow alloc\_head)$
4:         $cur\_tail \leftarrow (q \rightarrow alloc\_tail)$
5:         if $cur\_head.index \geq MAX\_ENTRIES$ then
6:             return $STATUS\_ERROR$
7:         $suc\_head \leftarrow (q \rightarrow next[cur\_head.index])$
8:         $tmp\_head \leftarrow (q \rightarrow alloc\_head)$
9:         if $cur\_head = tmp\_head$ then
10:            if $cur\_head.index \neq cur\_tail.index$ then
11:                return $STATUS\_SUCCESS$
12:            if $suc\_head.index = NULL\_INDEX$ then
13:                return $STATUS\_EMPTY$
14:            $CAS(addr = \&q \rightarrow alloc\_tail; old = [cur\_tail.index, cur\_tail.tag];$
                   $new = [suc\_head.index, cur\_tail.tag + 1])$
```

To remove an element from a FIFO queue list (Algorithm 4.11), we first read the current head and tail values (Lines 3-4). We then fetch the successor index of the sentinel element, i.e. the first element (Line 7), and make sure that the head has not changed (Line 9). After this, we check if the tail refers to any element behind the sentinel element. If not (Line 10), the tail pointer is advanced (Line 13) and the entire procedure is repeated. Otherwise, we read the data of the first element (Line 17), which then becomes the new sentinel element (Line 18). When the method completes, the old sentinel element can be reclaimed. To check if a queue is non-empty, we follow a very similar procedure (Algorithm 4.9), which does not remove any elements but simply makes sure that the tail points to the sentinel element and the sentinel's successor index is NULL (Lines 10-13).

To insert an element into a queue list (Algorithm 4.10), we initialize the allocated element's successor index to NULL (Lines 2-3). The method reads the tail index (Line 5) and fetches its successor index (Line 8). If the successor index is not NULL, the tail needs to be advanced to the very last element (Line 16). The successor index of an element pointed by the tail is updated to refer to the new element (Line 12). As the last step (Line 13), we attempt to update the tail pointer to refer to the new element. If the corresponding CAS operation fails, it has already been updated by another thread.

Algorithm 4.10 Inserting an element into a FIFO queue list.

```
 1: function INSERT($next$: dptr[], $tail$: dptr *, $index$: integer, $max$: integer)
 2:     $next[index].tag \leftarrow next[index].tag + 1$
 3:     $next[index].index \leftarrow NULL\_INDEX$
 4:     while True do
 5:         $cur\_tail \leftarrow *tail$
 6:         if $cur\_tail.index \geq max$ then
 7:             return $STATUS\_ERROR$
 8:         $suc\_tail \leftarrow next[cur\_tail.index]$
 9:         $tmp\_tail \leftarrow *tail$
10:         if $cur\_tail = tmp\_tail$ then
11:             if $suc\_tail.index = NULL\_INDEX$ then
12:                 if $CAS(addr = \&next[cur\_tail.index]; old = [suc\_tail.index, suc\_tail.tag];$
                                $new = [index, suc\_tail.tag + 1])$ then
13:                     $CAS(addr = tail; old = [cur\_tail.index, cur\_tail.tag];$
                            $new = [index, cur\_tail.tag + 1])$
14:                     return $STATUS\_SUCCESS$
15:             else
16:                 $CAS(addr = tail; old = [cur\_tail.index, cur\_tail.tag];$
                        $new = [suc\_tail.index, cur\_tail.tag + 1])$
```

4.6 Special Case Handling

A number of system calls required special handling in VirtuOS. In this section, we discuss the most relevant cases.

Exec. To load an executable image into a process, POSIX provides the *exec(2)* call. In this call, the entire virtual memory space is reinitialized as specified in the executable image. File descriptors are preserved unless they have the `O_CLOEXEC` flag set, which indicates that they need to be closed on exec. To implement POSIX semantics, we save information about all file descriptors that needs to be preserved in a temporary file. We currently save the file descriptor translation table and other relevant system information in a per-process temporary

Algorithm 4.11 Removing an element from a FIFO queue list.

```
1: function REMOVE($next$: dptr[], $ents$: entry[], $head$: dptr *, $tail$: dptr *, $max$: integer)
2:     while True do
3:         $cur\_head \leftarrow *head$
4:         $cur\_tail \leftarrow *tail$
5:         if $cur\_head.index \geq max$ then
6:             return $\{.index = STATUS\_ERROR, .data = \emptyset\}$
7:         $suc\_head \leftarrow next[cur\_head.index]$
8:         $tmp\_head \leftarrow *head$
9:         if $cur\_head = tmp\_head$ then
10:            if $cur\_head.index = cur\_tail.index$ then
11:                if $suc\_head.index = NULL\_INDEX$ then
12:                    return $\{.index = STATUS\_EMPTY, .data = \emptyset\}$
13:                $CAS(addr = tail; old = [cur\_tail.index, cur\_tail.tag];$
                       $new = [suc\_head.index, cur\_tail.tag + 1])$
14:            else
15:                if $suc\_head.index \geq max$ then
16:                    return $\{.index = STATUS\_ERROR, .data = \emptyset\}$
17:                $result \leftarrow ents[suc\_head.index]$
18:                if $CAS(addr = head; old = [cur\_head.index, cur\_head.tag];$
                               $new = [suc\_head.index, cur\_head.tag + 1])$
19:                    return $\{.index = cur\_head.index, .data = result\}$
```

file. Before the new program is started, the C library is initialized at which point we restore the file descriptor translation table from the temporary file and recreate any system call pages in the new address space.

Ioctl and Fcntl. The *ioctl(2)* and *fcntl(2)* system calls contain a parameter that may point to a memory region of variable size, depending on the request/command code passed. For example, the ioctl(TCGETS) system call passes a pointer to a *struct termios* variable to retrieve a terminal's attributes. We handle fcntl() by treating each command code separately, performing any necessary copies. For ioctl(), we use the `_IOC_SIZE` and `_IOC_DIR` macros to decode memory argument size and direction, and include special handling for those ioctl() calls that do not follow this convention.

Scatter-Gather I/O. Vectored I/O calls such as *readv(2)* and *writev(2)* allow the use of multiple, discontiguous buffers in a single invocation. Because VirtuOS uses an intermediate buffer in process's shared region, we simply allocate a contiguous buffer and consolidate the data copied into or from it. Instead of *readv* and *writev*, we execute *read(2)* and *write(2)* calls in the service domain.

Large Buffers. A system call can potentially transfer large amounts of data. We currently support contiguous buffers as large as 32MB as it is not very common for programs to use larger buffers in system calls. However, we do support the proper semantics by splitting larger requests into multiple system calls. For network-specific calls such as *sendto(2)* we additionally add the `MSG_MORE` flag to indicate that a user has more data to transfer. This flag indicates that the user will send more data, and the OS kernel needs to combine the data to avoid sending partial frames.

Epoll interface. The *epoll(2)* API is used in Linux as an alternative to POSIX's *poll(2)* and *select(2)* functions. In this API, *epoll_ctl(2)* adds or removes file descriptors to and from a watch list, which is maintained by the kernel. The *epoll_wait(2)* operation references a file descriptor that refers to the watch list. To simplify the implementation and avoid additional overhead, our C library does not keep track of the individual file descriptors added or removed in epoll_ctl. Only the service domains are aware of the file descriptors in their respective per-domain subsets. epoll_ctl sets a per-domain flag when any file descriptor pertaining to the domain is added to a watch list. We dispatch epoll_wait requests to all service domains whose corresponding domain flag is set.

Asynchronous I/O. To support asynchronous I/O for data intensive applications such as mySQL, we had to modify Linux's AIO library, which provides access to the AIO system call interface. The AIO interface provides asynchronous system calls to initiate I/O requests through AIO contexts and gathers results from associated completion queues using blocking *io_getevents(2)* call. Since the API allows combinations of operations pertaining to different service domains, io_getevents may need to merge results from multiple domains. Our implementation provides file descriptor translation and necessary integration with our C library for all AIO operations. A single user perceived AIO context is mapped to individual contexts in each domain. Since Linux currently allows the use of AIO only for regular files, we did not include support for network I/O in our prototype.

POSIX signals. POSIX signals interact with system calls because receiving a signal may result in a system call interruption. For most system calls, automatic system call restarting is supported, and control does not return to the user program after a signal is being handled. This is the most transparent model and does not require any additional support in VirtuOS because service domains need not know anything about signal delivery to user processes. However, a user may also request system call interruption if a signal is pending. Calls such as *read(2)* will complete with the `EINTR` error status after delivering the signal to the user program. To support this behavior, we inform service domains of pending signals for in-progress system calls. We mark the signal as pending in the remote worker thread's shadow process control block (PCB), resume the worker thread and let it abort the system call as if a local signal had been produced. In addition, we make note of pending signals through an additional flag in the system call request queue entries; this flag lets the worker thread recognize that a system call should be aborted even if the notification arrived before the request was started. To avoid spurious interruptions, we keep a per-thread counter that is incremented with each system call and store its current value in each system call request. The counter value functions as a nonce so that a service domain can match signal notifications to pending system calls and ignore delayed notifications.

Proc file system. The Linux /proc file system can be used to obtain information about file descriptors and other information pertaining to a process. Some applications inspect the information published via /proc. Our system call library translates accesses to /proc when a user process accesses the file descriptor directory so that programs see translated file descriptors only, hiding the fact that some file descriptors belong to service domains.

4.7 Limitations

The vertical decomposition of a monolithic kernel makes the implementation of calls that intertwine multiple subsystems difficult. For instance, Linux's *sendfile(2)* call, which directly transfers file data onto a network connection, must be implemented via user-level copying if the file and the network connection are serviced by different service domains. Since pipes and user memory are managed exclusively by the primary domain, *vmsplice(2)* which splices user pages into a pipe is not subject to the above limitation.

Our prototype also does not support *mmap(2)* for file descriptors serviced by storage domains. mmap could be supported by granting direct access to the service domain's memory, similar to how the request queue and data pages are shared. User processes would register a handler for the corresponding memory region to redirect faults to the service domain. The service domain would be identified by the mmap file descriptor's tag. When a program accesses memory in the mmap'ed region for the first time, page faults would occur. The primary domain's kernel would then send requests to the service domain to share physical frames corresponding to the page fault addresses. Shared physical frames would be mapped into the user virtual address space by the primary domain. The primary domain would also need to limit the number of shared frames; it could keep a list of shared frames sorted by the timestamp of the mapping. When the limit is exceeded, the primary domain could reclaim the frame with the oldest timestamp to create a new mapping. The service domain would also need to be notified to reuse the physical frame for a different file location. We leave the implementation of mmap() for future work.

Our current prototype also does not provide transparency for file path resolution when a storage domain is involved. To recognize accesses to files stored in a storage domain, the C library keeps track of a process's current working directory and translates all relative paths to absolute ones. This approach provides a different semantics if a process's current working directory is removed.

Chapter 5

Performance Analysis Framework

Prior to our work, performance monitoring in virtualized environments was limited because no existing framework provided support for all modes of virtualization; existing frameworks also had restrictions with respect to the modes of monitoring they supported. In our Perfctr-Xen framework, we addressed these challenges. Perfctr-Xen supports paravirtualized and hardware-assisted modes of virtualization in Xen and it enables support for both a-mode (accumulative mode) and i-mode (interrupt mode) monitoring. Our implementation supports the use of a wide spectrum of higher level tools and frameworks, thereby opening the way for VirtuOS's performance evaluation.

In this chapter, we discuss problems that occur when monitoring performance in virtualized environments such as Xen and how the implementation of our framework addresses them. We test the correctness and evaluate the accuracy of Perfctr-Xen in all modes of monitoring

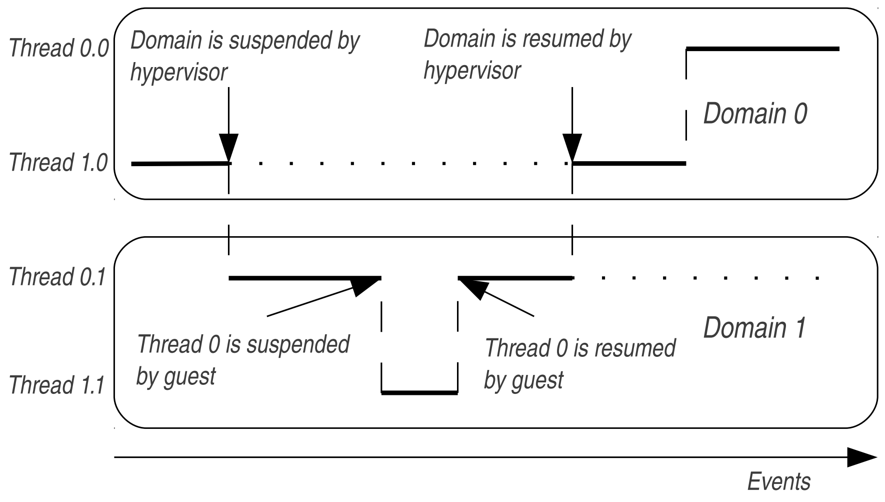

Thread 0.0
Thread 1.0
Thread 0.1
Thread 1.1
Domain is suspended by hypervisor
Domain is resumed by hypervisor
Domain 0
Thread 0 is suspended by guest
Thread 0 is resumed by guest
Domain 1
Events

Figure 5.1: Context switching in a virtualized environment. The guest domains and the hypervisor are both unaware of when a domain or thread switch takes place.

using several benchmarks.

5.1 Virtualizing Hardware Event Counters

The encapsulation of guest domains from the underlying hypervisor poses a difficulty for virtualizing performance counters, because these two components are mutually unaware of their scheduling policies. As shown in Figure 5.1, the guest kernel remains unaware if the hypervisor suspends its domain on the physical CPU on which it runs. Likewise, the hypervisor is unaware of when a guest kernel switches to a different user task on its domain's virtual CPUs.

Both inter- and intra-domain context switches involve the performance monitoring framework and may require updating machine-specific (MSR) registers. First, PMU (Performance

Monitoring Unit) configuration registers (e.g., event selectors) need to be re-programmed to reflect the desired event configuration of the thread to be resumed. Second, if the performance counter register contains the logical value of the thread to be resumed, it must be restored (and the value of the outgoing thread must be saved). Otherwise, its value must be sampled and recorded in the corresponding data structure for the thread or domain.

Since the guest domain kernel runs in a deprivileged environment, its access to the registers during intra-domain context switches must be managed by the hypervisor. For fully-virtualized domains, a trap-and-emulate approach can be used to intercept and emulate the corresponding privileged instructions that write to these registers. Although read accesses could be trapped as well, current architectures allow the hypervisor to grant direct read access to the MSR registers to guest domains. If the guest is adapted to use paravirtualization, the cost of trapping and emulating individual privileged instruction can be reduced by using hypercalls instead, which allows the batching of multiple updates. The guest can also cache previously activated configurations to avoid these hypercalls if possible, thus avoiding unnecessary writes to the MSR configuration registers.

Whereas the counter-related configuration information must be saved and restored on both inter- and intra-domain switches, the values of the registers containing the actual event counts require saving and restoring only if the register physically contains the thread's logical value during execution. Consequently, the cost of writing to these registers can be avoided when register is not required to contain thread's logical value, which holds true in two cases. First, if a counter is used to obtain accumulated event counts (in 'a-mode'), a virtualization-aware guest domain can apply the necessary correction offsets to obtain the logical accumulated value from the physical value. Second, in the case of the timestamp counter (TSC) register, hardware-assisted virtualization via Intel VT or AMD-V allows for transparent, per-domain offsetting, which also avoids the cost of physically updating these registers on a context switch. In addition, on some architecture implementations, it is impossible to safely update the TSC register since it may be shared across cores. Avoiding the save-and-restore cost is beneficial because it can be expensive (66-93 CPU cycles per register; for Pentium 4 as much as 18 registers must be restored).

On the other hand, if a counter is used in interrupt mode (‘i-mode’), the physical register contains a small negative value that will overflow, thus triggering an interrupt, after a desired number of events occurs. In this case, saving and restoring cannot be avoided. The interrupt is handled by the hypervisor, who must forward this interrupt to the affected domain via a virtual interrupt mechanism. Upon receipt of the virtual interrupt, the guest kernel notifies the user-level profiling components via a Unix signal. Previous work [71] claimed that such delivery needs to be synchronous, an assertion repeated in [36]. In Xen, the delivery of virtual interrupts to guest domains is not synchronous, but uses a software interrupt mechanism, thus making it possible that their delivery could be delayed by higher-priority interrupts. In Section 5.2.2, we argue that support for synchronous interrupt delivery is not required if the performance counter registers are restored before the guest domain is resumed, and if the guest domain verifies that a register in fact overflowed before delivering the notification to the user level.

Perfctr-Xen supports performance counter virtualization in Xen in three configurations, which required different and separate implementations: (1) for paravirtualized guest kernels, which use hypercalls to communicate performance counter configuration changes from guest to hypervisor, and in which the guest and hypervisor cooperate to maintain information about the current thread context; (2) for fully-virtualized guest kernels, which use the save-and-restore approach for all registers; and (3) a hybrid approach in which a guest can run in a hardware-assisted, fully-virtualized domain but still enjoy the generality of and the optimizations developed for the paravirtualization case.

5.2 Implementation

Our implementation is based on, and compatible with, the existing *perfctr* [77] implementation. In this section, we describe *perfctr* in detail and outline how we adapted it to enable performance counter virtualization in Xen for paravirtualized and hybrid modes. Section 5.2.2 describes our virtualization strategy for fully virtualized domains.

5.2.1 The *perfctr* library

We chose *perfctr* because it is widely used and provides the foundation for higher-level libraries and frameworks such as PAPI, HPCToolkit, or PerfExplorer, as shown in Figure 5.2. It is efficient, light-weight and allows direct access to performance counters in user mode. *Perfctr* supports a wide range of x86 implementations spanning multiple generations and

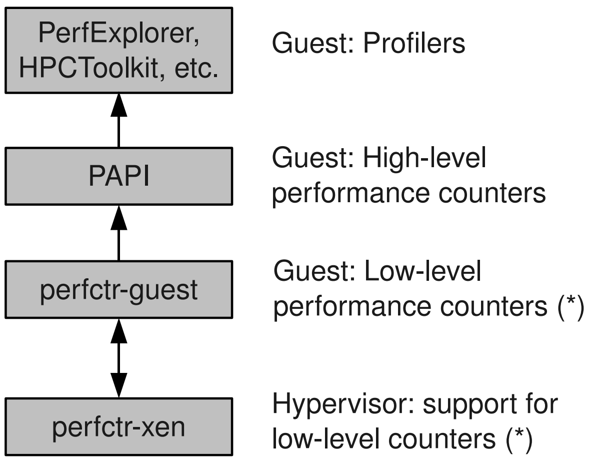

PerfExplorer, HPCToolkit, etc.
Guest: Profilers
PAPI
Guest: High-level performance counters
perfctr-guest
Guest: Low-level performance counters (*)
perfctr-xen
Hypervisor: support for low-level counters (*)

Figure 5.2: Software layers in Perfctr-Xen.

Components marked with an asterisk (*) were adapted from *perfctr*.

different vendors, whose hardware event counter implementation can differ significantly. In addition, *perfctr* works on non-x86 platforms such as PowerPC and ARM and can easily be integrated in any Linux distribution.

Perfctr consists of a kernel driver and a user-level library. The kernel driver maintains performance counter-related per-thread data structures, updates them on each context switch, and makes them available to the user-level library via a read-only mapping. Besides miscellaneous architecture-specific information, this per-thread data structure contains the following information:

- *Control State.* Information about which PMU data registers a thread is actively using, which events these registers count, and to which physical register address they are mapped. Similar information is kept with respect to the use of the time-stamp counter, which is also virtualized.

- *Counter State.* For each PMU data register, as well as the TSC register, two values are kept: Sum_{thread}, which reflects the thread's accumulated logical event count up to including the last suspension point; and $Start_{thread}$, which reflects the sampled value of the counter at the last resumption point.

Perfctr supports two types of counters: *a-mode* and *i-mode* counters. A-mode counters are used by threads to measure the number of events occurring in some region of a program. User code explicitly reads the counter's value when needed. When a thread wants to access the logical value of a counter at time t, a user library function issues a `RDTSC` or `RDPMC` instruction to obtain the register's physical value $Phys(t)$ and computes the logical value $Log_{thread}(t)$ as

$$Log_{thread}(t) = Sum_{thread} + (Phys(t) - Start_{thread}) \tag{5.1}$$

On each context switch, the *perfctr* kernel driver updates the accumulated value of the outgoing thread as $Sum_{thread} \leftarrow Sum_{thread} + (Phys - Start_{thread})$ to account for the events during the last scheduling period. In addition, the $Start_{thread}$ value of the thread to be resumed is reset as $Start_{thread} \leftarrow Phys$. Note that the actual physical register value is not changed on a context switch for a-mode counters.

I-mode counters, which are used for sampling, trigger interrupts after a certain number of events has occurred, which represents the sampling period. Since the value at which an overflow interrupt is triggered is fixed at 0 and cannot be programmed, the physical register must be set to a small negative value whose absolute value represents the desired length of the sampling period. *I-mode* counters are treated differently during a context switch: their physical value is saved on suspend and restored on resume. The Sum_{thread} field maintains the counter's accumulated logical value as for a-mode counters. The $Start_{thread}$ field is used to record the physical value when a thread is suspended. Consequently, the logical value of an i-mode counter can also be obtained using equation (5.1).

When an overflow occurs, *perfctr* handles this interrupt, identifies the register(s) that have overflowed and updates Sum_{thread}, then disables further event counting for these registers. Using the OS's signal delivery mechanism, a signal is sent to the user process. The signal handler is then responsible for recording the sample based on the provided user process's state and it must re-enable event counting. Once re-enabled, the physical value of the register is reset to the sampling period, which is also recorded in the data structure maintained by *perfctr*.

5.2.2 The Perfctr-Xen framework

Perfctr-Xen includes a hypervisor driver, a guest kernel driver, and a modified user-level library, whose functionality we describe in this section.

A-mode counters

The virtualization technique described in Section 5.2.1 requires that the underlying system perform two actions during a context switch: (1) update the counter state of the threads being suspended and resumed, and (2) activate the resumed thread's control state. As discussed in Section 5.1, in a virtualized environment, both intra-domain context switches between threads in a domain and inter-domain context switches between domains can occur. During intra-domain switches, the guest kernel can perform the state updates similar to the native implementation. For inter-domain context switches, the hypervisor must perform these updates.

We first considered having the hypervisor update each thread's counter state directly on the guest kernel's behalf. This approach has the advantage that no changes to the *perfctr* user library are required. However, it would create undesirable coupling between the hypervisor and the guest kernel implementations, because the hypervisor would need to traverse guest kernel data structures. Instead, we decided to split the control and counter state in two parts. At the guest kernel level, a per-thread data structure is maintained. At the hypervisor level, a per-VCPU data structure is maintained for each virtual CPU that is assigned to a guest domain. The hypervisor provides read-only access to this data structure to the guest kernel, who in turn maps it into the address space of each thread using performance counters.

The per-VCPU data structure is modeled after the per-thread data structure used in the native version of *perfctr* (in fact, our implementation uses the same data structure declarations, as discussed in Section 5.2.2). For each PMU data register, as well as for the TSC register, the hypervisor maintains two values per VCPU: $Start_{vcpu}$ and Sum_{vcpu}. $Start_{vcpu}$ represents the sampled value of the counter at the most recent resumption point of the domain or thread (whichever happened last). If the hypervisor resumes a domain, it directly updates

$Start_{vcpu}$ after sampling the counter. If the guest kernel resumes a thread, it requests via a hypercall that the hypervisor record the sampled value in $Start_{vcpu}$. The same hypercall is also used to activate this thread's counter-related control state.

The field Sum_{vcpu} represents the cumulative number of events incurred by this domain since the last intra-domain thread resumption point until the most recent domain suspension point. It is set to zero on each intra-domain switch during the hypercall that notifies the hypervisor that the guest kernel resumed a thread. On each inter-domain context switch, the Perfctr-Xen hypervisor driver updates the accumulated value of the outgoing VCPU as $Sum_{vcpu} \leftarrow Sum_{vcpu} + (Phys - Start_{vcpu})$ to account for the events incurred since the last intra- or inter-domain resumption point.

The Perfctr-Xen guest kernel driver maintains the value Sum_{thread} for each thread as in the native case, which represents the cumulative number of events up to the last thread suspension point. A counter's logical value at time t is computed as

$$Log_{thread}(t) = Sum_{thread} + (Phys^*(t) - Start^*_{thread}) \tag{5.2}$$

$Phys^*(t)$ represents the adjusted physical value that accounts for possible VCPU preemption, which is computed as

$$Phys^*(t) = Sum_{vcpu} + (Phys(t) - Start_{vcpu}) \tag{5.3}$$

Thus, the logical value represents the sum of the cumulative number of events until the last thread suspension point, plus the number of events encountered from there until the last domain resumption point while the domain was active, plus the events encountered since then until t, reduced by an adjusted start value $Start^*_{thread}$.

The adjusted thread start value $Start^*_{thread}$ compensates for the requirement that each intra-domain context switch includes a hypercall. Since this hypercall is introduced by our framework, we wish to exclude any events occurring during its execution. Right before resuming a guest thread, the guest kernel driver computes $Start^*_{thread} = Phys^*(t_r)$ after returning from the hypercall at time t_r

$$Start^*_{thread} = Sum_{vcpu} + (Phys(t_r) - Start_{vcpu}) \tag{5.4}$$

This adjustment excludes any events incurred between when the hypervisor sampled the counter during the hypercall and t_r. The inclusion of the term Sum_{vcpu} ensures that all such events are excluded, even if the domain was suspended and resumed during the hypercall by the preemptive scheduler. Since $Start^*_{thread}$ takes the place of $Start_{thread}$ in equation (5.1), we store its value in the $Start_{thread}$ field of the per-thread structure.

The use of $Start^*_{thread}$ enables an additional optimization. In applications in which multiple threads count the same types of events, the hypercall accompanying the intra-domain context switch does not change any counter's control state. In this case, we skip this hypercall. Consequently, $Start_{vcpu}$ is not reset to the counter's current physical value and Sum_{vcpu} is not reset to 0. Since we still initialize $Start^*_{thread}$ using equation (5.4), we allow the thread being resumed to subtract events incurred by other threads within the same domain since $Start_{vcpu}$ was last initialized. This optimization reduces the frequency with which $Start_{vcpu}$ is updated, which in turn increases the risk that an integer wrap-around leads to incorrect results when computing a thread's logical value. To reduce this risk, we expanded the width of the counters. Whereas *perfctr* uses 32 bits to represent only the lower 32 bits of all counters, our implementation uses their actual width (64 bits for the TSC register, and 48 bits for PMU data registers; 40 bits on older CPUs). We sign-extend based on the physical register width and store the extended values in 64-bit variables.

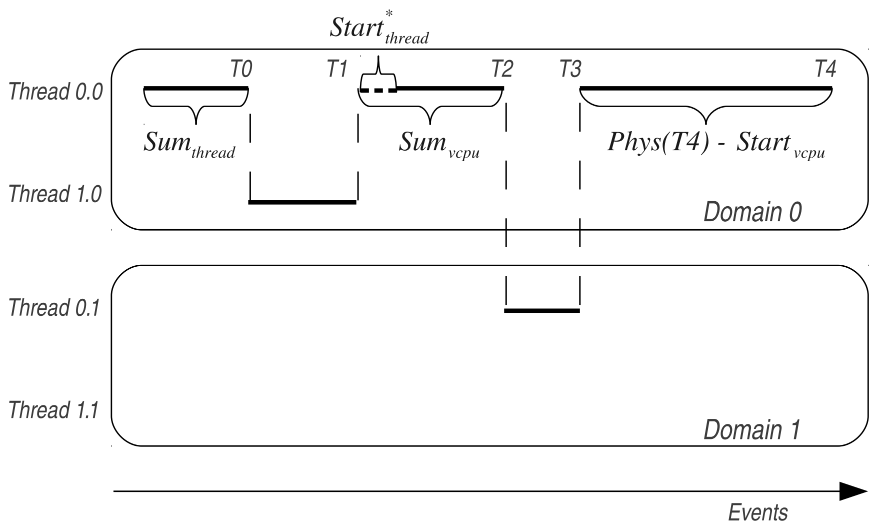


Figure 5.3: Example scenario for virtualized a-mode counters.

Example Scenario. Figure 5.3 shows an example scenario to illustrate equations (5.2) and (5.3). Initially, thread 0.0 in domain 0 is running. At point T0, thread 0.0 is suspended by the guest kernel and its accumulated event count is recorded in Sum_{thread}. At T1, thread 0.0 is resumed. The hypervisor sets $Sum_{vcpu} \leftarrow 0$; upon return from the hypercall, the guest records $Start^*_{thread}$. At point T2, the domain is suspended; the hypervisor records the number of events elapsed in Sum_{vcpu} and later resumes the domain at point T3. At this point, the hypervisor samples $Start_{vcpu}$ as $Phys(T3)$. Finally, the logical value computed at time T4 reflects the sum of the three segments during which the thread was active, while excluding those time periods during which the thread or domain was suspended.

I-mode counters

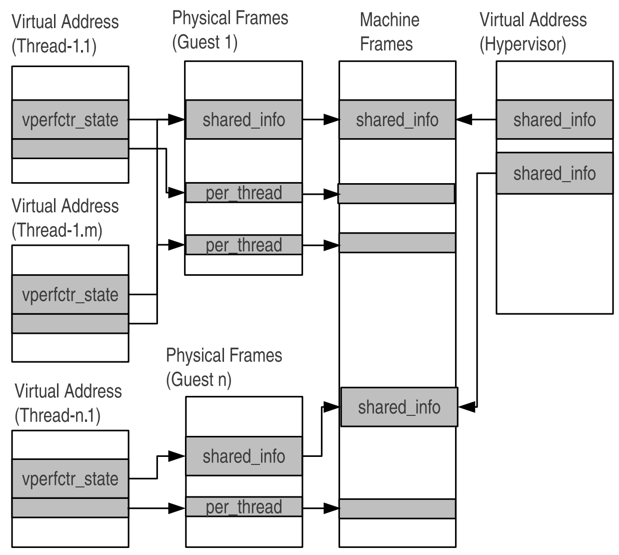
As in the native *perfctr* implementation, i-mode counters require saving and restoring the physical register value on both intra- and inter-domain context switches. In addition, when a thread using i-mode counters is suspended by the guest, the PMU must be reprogrammed to stop triggering interrupts for this counter. Since writes to PMU registers can be performed only by the hypervisor, an additional hypercall is necessary when a guest thread is suspended. Our implementation uses the $Start_{vcpu}$ and Sum_{vcpu} fields in the VCPU structure to maintain the currently active thread's $Start_{thread}$ and Sum_{thread} values at the time the thread is resumed. While a thread is active, we set its per-thread $Start_{thread} \leftarrow 0$ and $Sum_{thread} \leftarrow 0$ in order to be able to use equation (5.2) to compute the logical value (if desired).

When an intra-domain context switch occurs, a guest invokes a suspension hypercall which will update Sum_{vcpu} and store the current physical value in $Start_{vcpu}$. These values are then preserved in the Sum_{thread} and $Start_{thread}$ fields of the outgoing thread. The guest then invokes a resumption hypercall which will restore Sum_{vcpu} and $Start_{vcpu}$ based on the previously saved Sum_{thread} and $Start_{thread}$ fields of the thread to be resumed. The $Start_{vcpu}$ value will be written to the corresponding physical register. When an inter-domain context switch occurs, the hypervisor updates Sum_{vcpu} to account for the events incurred by the domain and preserves the outgoing VCPU's value in its $Start_{vcpu}$ field before restoring the physical register value from the saved $Start_{vcpu}$ field of the VCPU to be scheduled.

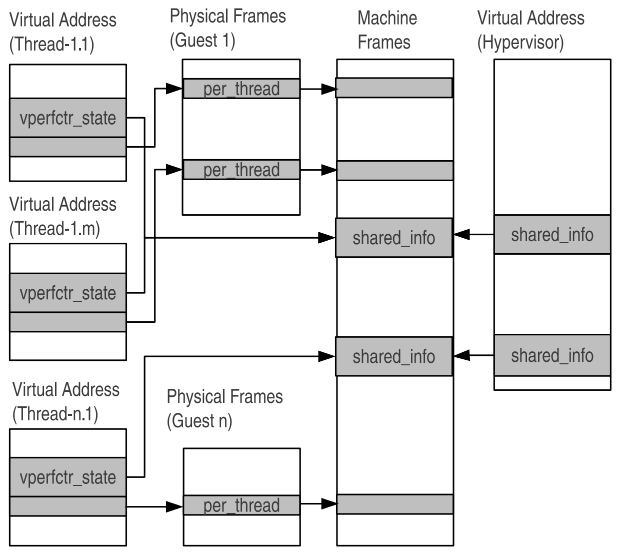
Overflow interrupts are controlled by local APIC (Advanced Programmable Interrupt Controller) on x86 and x86_64 platforms. When an overflow interrupt occurs, the hypervisor reinitializes APIC vector for performance counters and forwards the interrupt to the guest domain using a virtual interrupt we added for this purpose (`VIRQ_PERFCTR`). When the guest receives the virtual interrupt, it performs the same actions as in the native *perfctr* implementation, with three slight nuances: (1) When the guest receives the virtual interrupt, it suspends counting for the interrupted thread, and Sum_{vcpu} will be updated via the suspension hypercall. To prepare for the next sampling period, $Start_{thread}$ is reset with the negative sampling period. (2) When the user thread resumes counting, the resumption hypercall is executed, which restores $Start_{vcpu}$ from $Start_{thread}$ and sets the physical register value from $Start_{vcpu}$. (3) Unlike original *perfctr* implementation, the guest using Perfctr-Xen does not need to do anything with APIC controller, as it is handled by the hypervisor.

The virtual interrupt is not delivered synchronously because a software interrupt mechanism is used. As such, it is possible that the delivery of the interrupt is delayed, for instance, because other, higher-priority interrupts are being handled first. This could have two consequences. First, it could affect the accuracy because the events incurred during those interrupts will be counted as being part of that thread's activity. However, it is already the case that interrupt-related guest kernel activities can perturb the currently running thread's event count. Second, it is possible that a higher-priority interrupt triggers a context switch in the guest. When the virtual interrupt is eventually handled, a different thread may be running on the VCPU. Since we save and restore each thread's counter state on intra-domain interrupts, we can check if the currently running thread indeed encountered an overflow (i.e., if any of its counters have a non-negative value), and prevent the delivery of the overflow notification if this is not the case. When a thread whose counters have overflowed is suspended before the interrupt has been delivered, we mark it by setting an 'interrupt pending' flag, which is checked when the thread is resumed so that the overflow notification is delivered in the correct context.

Memory Management


Virtual Address (Thread-1.1)
Physical Frames (Guest 1)
Machine Frames
Virtual Address (Hypervisor)
vperfctr_state
shared_info
per_thread
Virtual Address (Thread-1.m)
Physical Frames (Guest n)
Virtual Address (Thread-n.1)

(a) Hybrid mode


Virtual Address (Thread-1.1)
Physical Frames (Guest 1)
Machine Frames
Virtual Address (Hypervisor)
vperfctr_state
per_thread
shared_info
Virtual Address (Thread-1.m)
Virtual Address (Thread-n.1)
Physical Frames (Guest n)

(b) Paravirtualized mode

Figure 5.4: Page mappings in Perfctr-Xen.

The virtualization approach described above relies on sharing data structures between hypervisor and guest threads, as well as between guest kernel and guest threads. To expose the hypervisor's per-VCPU data structures, we extended the existing `shared_info` data structure in Xen. This structure is per-domain and contains data that Xen hypervisor shares with a domain such as VCPU, event channel and wall clock information. Our extension required increasing the structure's size from 1 page to 8 pages; as a result, we needed to modify those places in the code where a size of a single page was assumed. The additional information, which is kept in 7 adjacent pages, is also made visible to user threads via a read-only mapping to facilitate the computation of the logical counter value. User threads also have read-only access to per-thread information which is mapped into their user space as in the original implementation.

The way in which the per-VCPU data mapping is established differs between paravirtualized and hardware-assisted mode. In paravirtualized mode, shown in Figure 5.4a, the `shared_info` structure does not appear as physical memory to the guest kernel. Instead, it is allocated by the hypervisor in machine memory and appears at a fixed virtual address in the guest kernel's address space. The corresponding machine address is communicated to the guest kernel through the `xen_start_info` data structure. The guest kernel uses a Xen Guest API (`xen_remap_domain_mfn_range`) to create an additional mapping to these machine frames.

In hardware-assisted mode, shown in Figure 5.4b, the guest has full control over its physical address space. It can allocate the `shared_info` structure in any of its physical page frames. The chosen physical address is communicated from the guest kernel to the hypervisor. Since `shared_info` appears in the guest's physical memory map, Linux's standard mapping API (`vm_insert_page`) can be used to add read-only mappings into the user threads' address spaces.

A domain may use multiple VCPUs, and the guest kernel scheduler may migrate threads between VCPUs based on its scheduling policy. Consequently, we expose the per-VCPU data structures of *all* VCPUs to every user thread. We also added an additional field `smp_id` to the per-thread structure to record the VCPU on which the thread is resumed. The user-level library uses this field as an index to access the correct per-VCPU structure.

Care must be taken to handle thread or domain migrations that may occur while accessing those counters. We implemented an optimistic approach in which we check if the values of the $Start_{thread}$ and $Start_{vcpu}$ fields corresponding to the TSC counter changed between before and after the attempted access. Such a change indicates a domain and/or thread migration, in which case we retry the access until we succeed.

Implementation Effort

In addition to providing compatibility with *perfctr*, we aimed to reuse as much of its codebase as possible. We were able to reuse the architecture-dependent code portions almost entirely, which will allow us to add support for newer CPU families as soon as they are supported by *perfctr*. Both the guest kernel driver and the hypervisor driver are based on *perfctr*. For the guest kernel driver, we replaced the functions that assumed direct access to the hardware with the appropriate hypercalls. For the hypervisor driver, we needed to provide glue code

Table 5.1: New or modified code (Perfctr-Xen).

Component	Number of Lines	Details
perfctr	563	VCPU support, hypervisor communication, etc.
Linux	36	shared_info management, VIRQ_PERFCTR
Xen	3488	perfctr-xen, shared_info management, VIRQ_PERFCTR

so that it could function within the Xen hypervisor rather than the Linux kernel for which it was designed. This glue code was written in the form of preprocessor macros and inlined functions contained in a separate header file, allowing us to avoid changes to most of the *perfctr* code. Table 5.1 summarizes the total amount of added or modified lines of code in the Xen hypervisor, the Linux guest kernel and the imported *perfctr* code. We note that more than half of the number of lines of code added to Xen stems from the addition of the *perfctr* driver.

Counter Virtualization in Fully-Virtualized Domains

The implementation described above requires that the guest kernel includes our Perfctr-Xen implementation so that it can benefit from the optimizations we made to enable direct access to counter values, which avoids the cost associated with save-and-restore for a-mode counters. Fully-virtualized domains do not require any guest kernel changes.

For fully virtualized domains, Xen's VPMU driver already supports counter virtualization for PMU registers on some recent CPUs. This virtualization is achieved by using a save-and-restore mechanism for PMU registers on inter-domain context switches as well as a hardware-assisted trap-and-emulate mechanism for PMU configuration registers. (A similar approach was implemented for KVM in [36].) The hypervisor intercepts the privileged instructions a domain uses when writing to configuration registers. A hardware-supported access bitmap allows the hypervisor to provide exclusive access to dedicated PMU registers for a domain. These mechanisms allow the use of the native *perfctr* implementation in fully-virtualized environments for the PMU registers, but they fail to provide per-VCPU virtualization for the TSC register, which cannot be reliably written to.

Xen exploits the TSC offsetting feature provided in hardware-assisted virtualization, so that each domain can set its own virtual initial value. However, this per-VCPU offset δ is not adjusted during inter-domain switches, hence does not reflect just the cycles during which a particular VCPU was active. To address this problem, we modified the implementation in the following way. When a domain is suspended, we take a sample of the TSC value ($TSC_{last} \leftarrow Phys(t)$). When the domain resumes, we obtain the current value $TSC_{cur} \leftarrow Phys(t)$, and re-calculate the TSC offset as $\delta \leftarrow \delta - (TSC_{cur} - TSC_{last})$. The updated offset is recorded by the CPU and will be reflected when a guest executes the `RDTSC` instruction.

5.3 Experimental Evaluation

Our Perfctr-Xen implementation was able to pass all *perfctr* and PAPI built-in tests. We verified that it functioned correctly with higher-level tools such as the HPCToolkit profiler. In this section, we discuss our experiments to validate the correctness of our implementation using microbenchmarks, we discuss test results obtained for the SPEC CPU2006 benchmarks, and show profiling results obtained using the HPCToolkit running on top of Perfctr-Xen. All experimental results were obtained for Xen 4.0.1, Linux 2.6.32.21-PvOps, perfctr 2.6.41 run on a Intel Xeon E5520 with 2x4 cores and 12 GB of RAM.

5.3.1 Goals and Methodology

In the experimental evaluation, we test the correctness of our hardware performance counter virtualization and verify the accuracy of the obtained results.

We ran series of micro- and macrobenchmarks and compared the results we obtained with Perfctr-Xen running inside a virtualized environment and with the unmodified perfctr library executing in a native (non-virtualized) Linux system. We also present results for the existing VPMU solution for HVM domains as an additional baseline where we want to emphasize measurement accuracy.

5.3.2 Correctness

To validate the correctness of our implementation, we compared the Perfctr-Xen implementation running in Xen with the original *perfctr* implementation running in native mode on the same hardware. For most counters, we expect to obtain the same value. For some counters (e.g., cache misses) we expect to see slight deviations because different domains running in parallel may compete for the same resource.

A-mode Counters

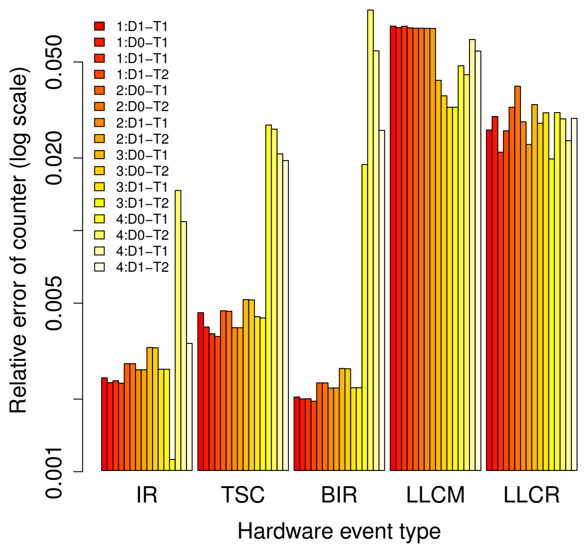


Figure 5.5: Microbenchmark result for a-mode counters.

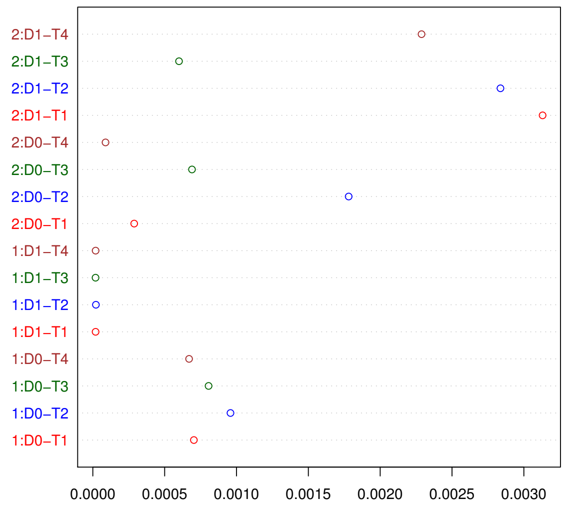
We first ran a specially developed 1-minute long microbenchmark. This synthetic microbenchmark heavily exercises branch instructions, memory accesses, and floating-point operations without performing any useful task. In Figure 5.5, the error relative to the native execution environment is shown for several test scenarios and event types. We considered two domains (Dom0 and Dom1) and two threads (Thread 1 and Thread 2) in each domain, running in parallel. Each test result is denoted as $N : D_x - T_y$, where N is the test case scenario, x is a guest domain and y is a thread. We considered the following test scenarios N, which represent different arrangements of CPU multiplexing: (1) Each domain runs on two dedicated physical cores (PCPUs), and each thread in every domain runs on a dedicated VCPU. (2) Each domain runs on a dedicated PCPU and all threads in every domain run on a shared VCPU. (3) Domains run on a shared PCPU and all threads in each domain run on a shared VCPU. (4) Like (1), except that threads are randomly migrated across VCPUs, and VCPUs are randomly migrated across different PCPUs. We used the Xen `xm` command to pin VCPUs to PCPUs, and we used the Linux `taskset` command to pin threads to VCPUs.

We considered the following counters: TSC (Time Stamp Counter), IR (Instructions Retired), BIR (Branch Instructions Retired), LLCM (L2 cache misses), and LLCR (L2 cache references). The results show some deviations, but the overall relative error (compared to native) remains very small. As expected, the per-thread values for the number of cycles spent, instructions and branch instructions retired match more closely the values obtained in native execution than the values corresponding to cache misses because those values are less affected by resource sharing. The results shown were obtained for paravirtualized domains; we obtained comparable results for hardware-assisted domains using our hybrid mode implementation.

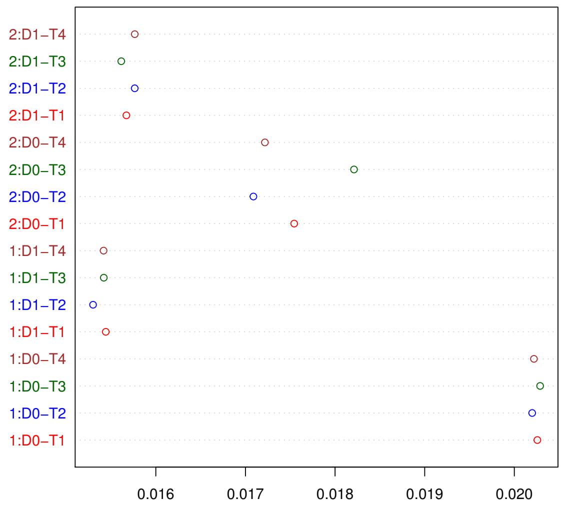
I-mode Counters

To verify the functioning of our i-mode counter implementation, we used PAPI's included tests. We present the results for the *overflow_pthreads* test. The test is a synthetic benchmark that performs a set of floating point operations. We ran the test for 300 sampling periods, and recorded the logical counter values afterwards. The benchmark runs 4 threads for which it uses a random CPU (VCPU in our case) assignment. We present results for two scenarios: (1) Dom0 and Dom1 run on separate PCPUs. (2) Dom0 and Dom1 run on a shared PCPU.

As Figure 5.6a shows, the error relative to the native mode for the number of retired floating-point instructions (PAPI_FP_INS) is negligible. Figure 5.6b shows the number of cycles as measured using the PAPI_TOT_CYC event type, which exhibit a larger relative error. This result is expected because we do not compensate for events occurring in the hypercall events at resumption points when using i-mode counters. The hypercalls consume cycles, but do not perform any floating point operations.



(a) PAPI_FP_INS



(b) PAPI_TOT_CYC

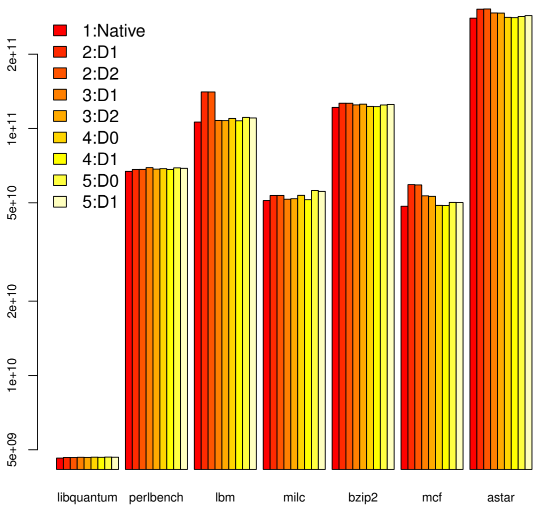
Figure 5.6: PAPI overflow test for i-mode counters.

5.3.3 Benchmarking with A-mode Counters

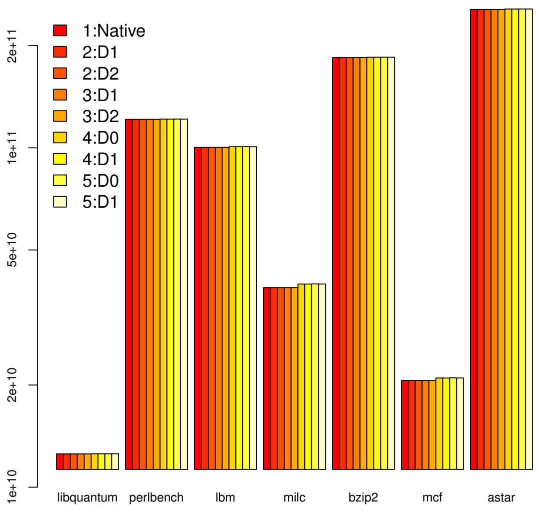
We used the SPEC CPU2006 benchmarks to show error estimates for a-mode counters upon CPU and memory bounded workloads. Native mode execution is again used as reference point. Since Dom0 is a paravirtualized domain in Xen, we used the Dom1 and Dom2 domains for tests that include fully-virtualized domains. (To exclude any possible effect of Dom0, we pinned it to a dedicated core.) We considered 5 scenarios: (1) Native mode execution. (2) Fully-virtualized domains Dom1 and Dom2, each running on a dedicated core (DC). (3) Fully-virtualized domains Dom1 and Dom2 running on the same core (SC). (4) Paravirtualized domains Dom0 and Dom1, each running on a dedicated core (DC). (5) Paravirtualized domains Dom0 and Dom1 running on the same core (SC).

The official SPEC distribution contains a large set of different benchmarks. We ran all of them using the 'train' problem size and recorded the total number of events counted during their execution. Since some benchmarks were executed under different data sets, we calculated the cumulative event counter values for all data sets. We present results for a subset of benchmarks only, choosing those for which both a non-negligible number of events was counted and for which the difference between the scenarios was largest; these represent the relative weakest performance of our framework.

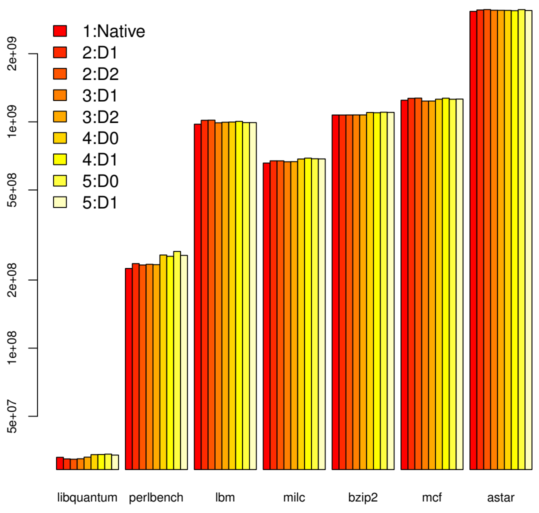
In Figure 5.7a, the results for the cycle counts reported by the virtualized TSC are shown. If the benchmarks execution were unaffected by virtualization, and if our framework achieved the same accuracy as *perfctr* running natively, we would expect to obtain the same results for all test scenarios for a given benchmark. This is true for most benchmarks, although 3 benchmarks (*mcf*, *astar*, and *lbm*) show significant deviations for the fully virtualized configuration. When counting the number of instructions retired (Figure 5.7b), we did not observe any significant differences.



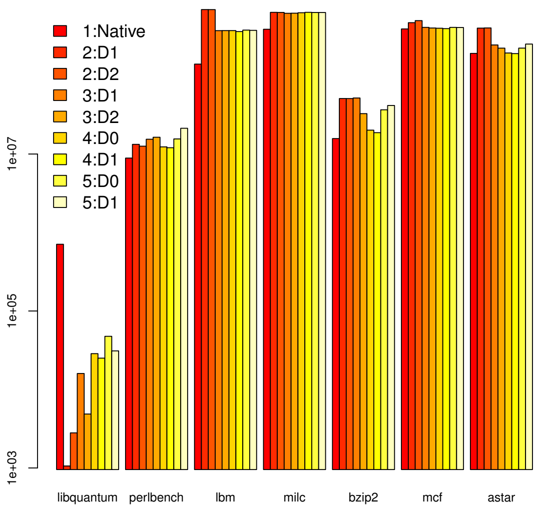
(a) Time Stamp Counter (TSC)



(b) Instructions Retired



(c) L2 Cache References



(d) L2 Cache Misses

Figure 5.7: SPEC CPU2006 test for a-mode counters.

Figures 5.7c and 5.7d display the number of L2 cache references and misses, respectively. Since these events are more strongly influenced by environmental factors inherent to the virtualized environment, they show slightly larger deviations, particularly for the number of cache misses. For example, *libquantum* shows a significant drop in the number of cache misses observed, although the number of cache references is roughly the same. These effect warrant further investigation to ascertain if they indeed reflect environmental circumstances or are caused by inadvertent interactions with the measurement framework.

5.3.4 Benchmarking with I-mode Counters

HPCToolkit's sampling mechanism is based on PAPI, which exploits the i-mode counter capabilities of our framework. As a profiler, HPCToolkit maps sample counts to individual functions. We tested HPCToolkit on the SPEC CPU2006 benchmarks. As an example, we selected the *429.mcf* benchmark, which performs combinatorial analysis. We considered PAPI_TOT_CYC (number of cycles), PAPI_L2_TCM (L2 cache misses), and PAPI_BR_INS (number of branch instructions) events. We considered sampling periods of 40000, 1000, and 500 events, the overall event counts ranged from 10s to 100s of millions of events. Similar to our previous setup, we run 2 concurrent instances of HPCToolkit in two separate domains using the following scenarios that correspond to labels in Table 5.2. (1) Domains Dom0 and Dom1 run on the same PCPU. (2) Domains Dom0 and Dom1 run on different PCPUs. We present results for all top-level functions that accounted for at least 1% of the total number of samples, sorted by decreasing number of samples. The tables show the ratio of the sample counts reported under virtualized vs. native execution. For most functions, similar counts were reported, although one (primal_iminus) shows significant differences which will warrant further investigation. We note that the same set of functions was identified in both execution modes, making the use of HPCToolkit in a virtualized environment a viable tool for identifying bottleneck functions that account for the largest proportion of events.

Table 5.2: HPCToolkit profiling results for 429.mcf. Sample count ratio virtualized/native.

(a) PAPI_TOT_CYC (b) PAPI_L2_TCM

Function	1:D0	1:D1	2:D0	2:D1	1:D0	1:D1	2:D0	2:D1
main	0.98	0.99	0.95	0.98	1.01	1.01	0.99	1.01
global_opt	0.98	1	0.95	0.98	1.01	1.01	0.99	1.01
price_out_impl	0.98	1.01	0.95	0.99	1	1.01	0.98	1.01
primal_net_simplex	0.98	0.98	0.94	0.96	1.03	1.02	1.01	1.01
primal_bea_mpp	0.99	0.99	0.97	0.98	1.04	1.04	1.07	1.05
replace_weaker_arc	0.9	0.97	0.88	0.94	0.97	0.97	1	0.96
refresh_potential	0.96	0.98	0.88	0.9	1.04	1.01	0.9	1.01
update_tree	0.96	0.97	0.9	0.93	0.83	0.72	0.53	0.61
primal_iminus	1.56	1.56	1.36	1.51	4.5	4.75	3.5	4.83
insert_new_arc	1.2	1.13	1.02	1.13	0.27	0.46	0.3	0.22
flow_cost	0.94	0.93	0.94	0.94	0.97	1.03	0.91	0.94
dual_feasible	0.95	0.95	0.95	0.93	0.95	0.95	1.05	1
suspend_impl	0.91	0.9	0.9	0.89	1	0.94	1	1.06

(c) PAPI_BR_INS

Function	1:D0	1:D1	2:D0	2:D1
main	1.01	1.01	0.98	1.01
global_opt	1.01	1.02	0.98	1.02
price_out_impl	1.02	1.04	0.99	1.03
primal_net_simplex	1	1	0.97	1
primal_bea_mpp	1	0.98	0.97	1
replace_weaker_arc	0.99	1.05	1.04	1.09
refresh_potential	1.02	1.28	1.03	1.04
update_tree	0.86	0.75	0.84	0.86
primal_iminus	1.53	1.47	1.23	1.27
insert_new_arc	0.99	0.87	0.95	0.96
flow_cost	0.97	0.97	0.97	0.97
dual_feasible	1.03	1	1	1
suspend_impl	1.07	1	1	1.07

Chapter 6

Experimental Evaluation

Our current prototype implementation uses the Linux 3.2.30 kernel for all domains. We tested it with Alpine Linux 2.3.6, x86_64 (a Linux distribution which uses uClibc 0.9.33 as its standard C library) using a wide range of application binaries packaged with that distribution, including OpenSSH, Apache 2, mySQL, Firefox, links, lynx, and Busybox (which includes ping and other networking utilities). In addition, we tested compilation toolchains including GCC, make and abuild. Our system is sufficiently complete to be self-hosting.

The specification of the system used for evaluation is shown in Table 6.1.

Table 6.1: System specification.

System Parameter	Configuration
Processor	2 x Intel Xeon E5520, 2.27GHz
Number of cores	4 per processor
HyperThreading	OFF (2 per core)
TurboBoost	OFF
L1/L2 cache	64K/256K per core
L3 cache	2 x 8MB
Main Memory	12 GB
Network	Gigabit Ethernet, PCI Express
Storage	SATA, HDD 7200RPM

6.1 Goals and Methodology

Our experimental evaluation comprises of (1) an evaluation of VirtuOS's overhead during system call handling and process coordination, (2) an evaluation of its performance for server workloads, and (3) a verification of its ability to recover from service domain failures.

Our goals are to show that VirtuOS imposes tolerable overhead for general workloads, that it retains the performance advantages of exceptionless system call dispatch for server workloads, and that it can successfully recover from service domain failures.

We run series of micro- and macrobenchmarks to that end. We use a native Linux system or Linux running inside a Xen domain as baselines of comparison, as appropriate. We used the distribution-provided configurations for all programs evaluated in macrobenchmarks. We were unable to compare to FlexSC due to its lack of availability.

6.2 Overhead

Compared to a conventional operating system, VirtuOS imposes multiple sources of overhead, which includes file descriptor translation, spinning, signaling (if necessary), copying, and process coordination. All microbenchmarks were run at least 10 times; we report the average result. We found the results to be highly consistent, with a relative standard deviation of less or equal than 2%.

6.2.1 System Call Dispatch & Spinning.

Our first microbenchmark repeatedly executes the *fcntl(2)* call to read a flag for a file descriptor that is maintained by the storage domain. In native Linux, this system call completes without blocking the calling thread. In VirtuOS, as described in Section 3.3.1, this call is submitted into the service domain's request queue. Our microbenchmark considers the single-threaded case in which there are no other user-level threads to run after the call is submitted. In this case, the calling thread will spin for a fixed number of iterations (the spinning threshold), checking if the request has been processed. If not, it will block the underlying kernel thread, requiring the more expensive, interdomain interrupt-based notification from the service domain.

For a single fcntl(2) call, we found that we needed to iterate at least 45 times for the system call to complete without blocking. If the system call completed without blocking, the achieved throughput was 0.7x that of the native case, which we attribute to the file translation and dispatch overhead, which outweighed any benefit due to exceptionless handling. Otherwise, if notification is necessary, we are experiencing a slowdown of roughly 14x. This result shows that spinning is a beneficial optimization for workloads that do not have sufficient concurrency to benefit from user-level threading. We found, however, that we needed a much larger spinning threshold (1000 iterations) to achieve the best performance for our macrobenchmarks. We use the same value for all benchmarks; on our machine, 1,000 iterations require approximately 26,500 machine cycles.

6.2.2 Process Coordination

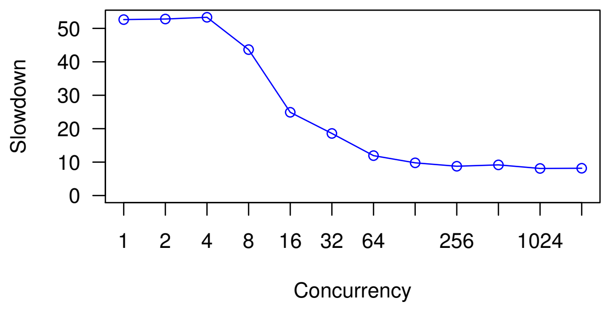


Figure 6.1: Process creation overhead.

As discussed in Section 4.3, the primary and the service domains need to coordinate when processes are created and destroyed. Requests to create and destroy processes are handled sequentially by a single kernel thread in each service domain. We created a microbenchmark that forks N concurrent processes, then waits for all of them. The forked processes simply exit. The case $N = 1$ represents the case of serial execution of single programs, such as in a shell script, albeit without actually executing any commands. Figure 6.1 shows a slowdown of 52x for this benchmark case, which decreases to about 8x as more processes are concurrently forked. This decrease shows the benefit of batching, which reduces the number of interdomain interrupts.

6.2.3 Copying Overhead

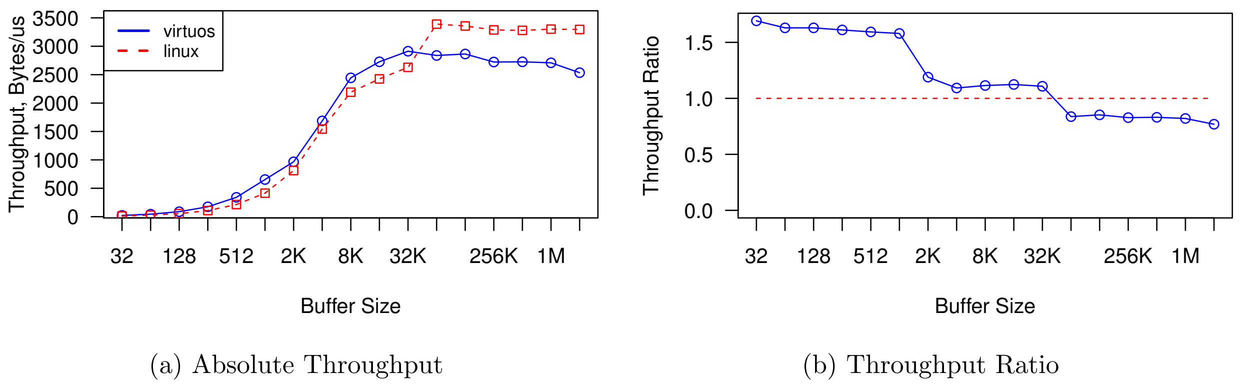


Figure 6.2: Throughput vs Buffer Size for writing to tmpfs.

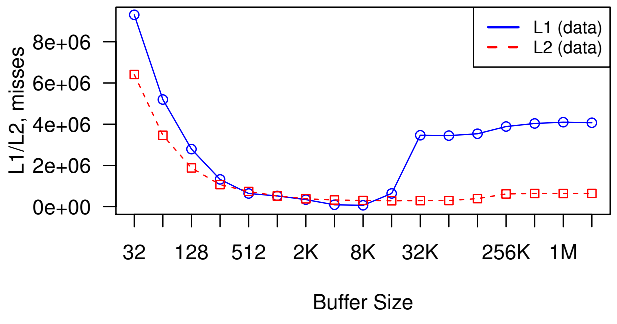
As described in Section 3.2.2, VirtuOS requires an additional copy for system calls that access user memory. Simultaneously, we expect those system calls to also benefit more greatly from exceptionless dispatch. We created a microbenchmark to evaluate these costs and benefits. The benchmark writes 16MB in chunks of 32, 64, up to 2MB to a file created in a *tmpfs* filesystem, which is provided by a native kernel in the baseline case and by a storage domain in VirtuOS. Linux implements this file system type using anonymous paged virtual memory. For the data sizes in our experiments, no paging is necessary. Thus, any differences reflect a combination of the effects of exceptionless system call handling and the additional data copy required by our method of dispatching system calls. Figures 6.2a and 6.2b show absolute throughput for various block sizes and throughput ratio correspondingly.

Since the overall amount of data written remains the same, smaller block sizes indicate a higher system call frequency. For block sizes less than 64K, the savings provided by the exceptionless model in VirtuOS outweigh the additional costs. Such small block sizes are common; as a point of reference, the file utilities included in our distribution's GNU coreutils package (e.g., cp & cat) use a 32K block size. For larger block sizes, the copying overhead becomes more dominant, reducing performance to about 0.8x that of native Linux.

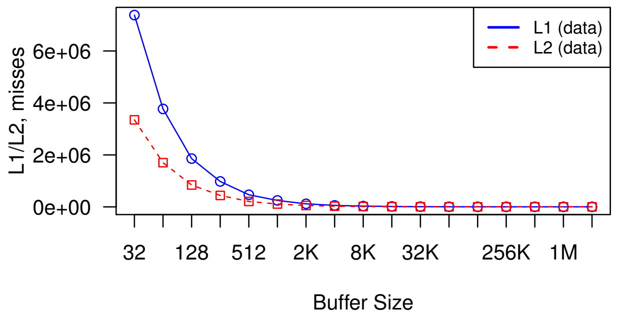
6.2.4 Cache misses

The memory copying overhead identified in Section 6.2.3 prompted us to perform an additional, in-depth analysis using the Perfctr-Xen framework. We repeated the experiment to measure L1/L2 data cache misses. Counted cache misses include the following types: 1) A cold miss when a cache entry is loaded first time from the memory. 2) A capacity miss when some cache entries are evicted to load others due to finite cache size. 3) A conflict miss when same cache entries accommodate different memory addresses. 4) An invalidation miss when an entry is evicted due to concurrent updates by other cores and CPUs.

As shown in Figure 6.3a, the number of L1/L2 cache misses goes down as the chunk size increases for chunks less than 32K. As the chunk size increases, the total number of system calls decreases, and we expect to see fewer invalidation misses. For 32K+ chunks, we also observe capacity misses because the size of the L1 data cache is 32K in our system. To verify that cache misses for smaller chunks are not capacity related, we performed additional experiments. We measured the number of L1->L2 writebacks (Figure 6.4), which increases drastically for blocks greater than 32K+. We also modified the microbenchmark to elide the data transfer for system calls while retaining total number of system calls. As shown in Figure 6.3b, the number of cache misses monotonically decreases even for blocks >32K. Taken together, the experiments demonstrate a relationship of the copying overhead in Section 6.2.3 to the L1 data cache capacity.



(a) Copying Enabled



(b) Copying Disabled

Figure 6.3: L1/L2 Cache Misses (Perfctr-Xen).

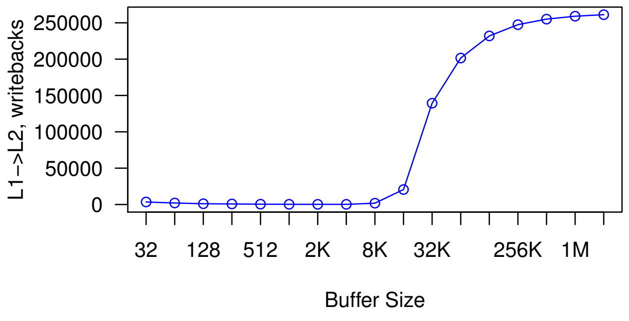


Figure 6.4: L1->L2 writebacks (Perfctr-Xen).

6.3 Performance

6.3.1 TCP Throughput and Latency

We first measured streaming TCP network performance by sending and receiving requests using the TTCP tool [1], using buffer sizes from 512 bytes to 16 KB.

We compared the performance of the following configurations (1) Native, non-virtualized Linux; (2) Linux running in Xen/Dom0; (3) Linux running in Xen/DomU PVHVM with configured PCI passthrough for a network card; (4) Linux running in Xen/DomU using netback drivers in Dom0; (5) VirtuOS. We used configuration (4) as a proxy for the performance we would expect from a Xen driver domain, which we were unable to successfully configure with the most recent version of Xen.

For all configurations, we did not find any noticeable differences; all are able to fully utilize the 1 Gbps link with an achieved throughput of about 112.3 MB/s, independent of the buffer size used. In all configurations, this throughput is achieved with very low CPU utilization (between 1% and 6% for large and small buffer sizes, respectively) in Linux and VirtuOS's primary domain. We observed a CPU utilization of about 20% on the network service domain, due to the polling performed by kernel worker threads described in Section 3.3.2. This relative overhead is expected to decrease as the number of concurrent system call requests from the primary domain increases since more CPU time will be used for handling system call requests than for polling.

We also analyzed TCP latency using lmbench's *lat_tcp* tool [69, 70], which measures the round-trip time for sending 1-byte requests. These results are shown in Figure 6.5. We used two spinning thresholds for VirtuOS: default and long, which correspond to the default settings used in our macrobenchmarks and to an infinite threshold (i.e., spinning until the request completes).

Here, we observed that VirtuOS's latency is slightly higher than Linux's, but significantly less than when Xen's netfront/netback configuration is used. We conclude that VirtuOS performs better than alternative forms of driver isolation using Xen domains. Furthermore, if desired, its latency can be further reduced by choosing longer spinning thresholds, allowing users to trade CPU time for better latency.

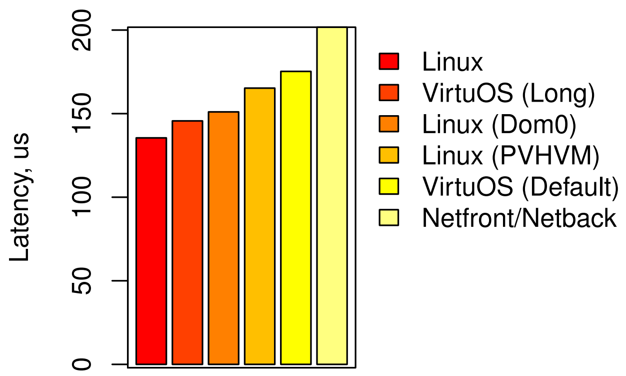
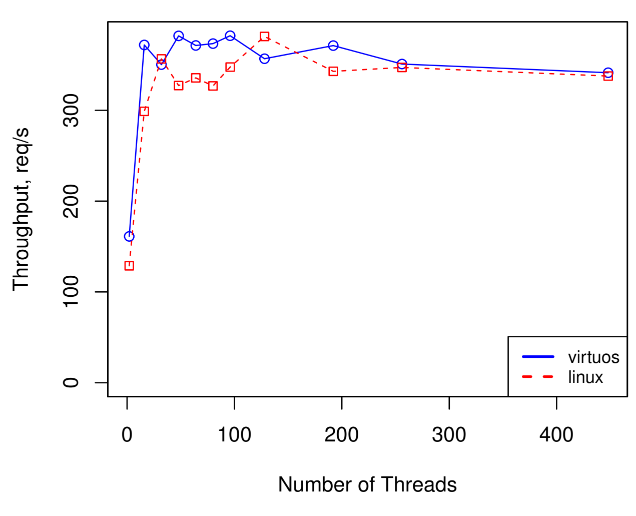


Figure 6.5: TCP 1-byte roundtrip latency (lat_tcp).

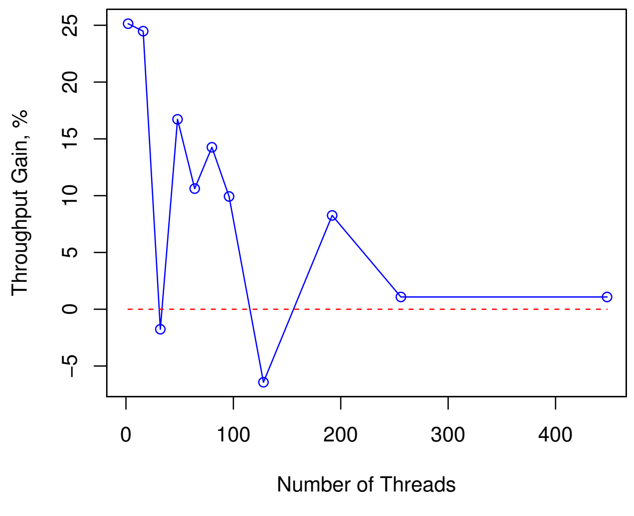
6.3.2 Multithreaded programs

We evaluated the performance of multithreaded programs when using the network and storage domains. We use the OLTP/SysBench macrobenchmark [62] to evaluate the performance of VirtuOS's network domain. In this benchmark, a mySQL server running in VirtuOS receives and responds to 10,000 requests, each comprising of 10 selection queries with output ordering, sent by network clients. The client uses multiple, concurrent threads, each of which issues requests sequentially. The files containing the database are preloaded into the buffer cache to avoid skewing the results by disk I/O. The benchmark records the average response time over all requests; we report throughput computed as *number of threads / average response time.*

We present throughput and the throughput gain of VirtuOS vs Linux in Figures 6.6a and 6.6b respectively. VirtuOS's performance in this benchmark mostly matches or exceeds that of Linux by 1-16%. To evaluate the performance of the storage domain, we used the FileIO/SysBench benchmark [62]. This benchmark generates 128 files with 1GB of total data and performs random reads with a block size of 16KB. We examined two configurations. In the first configuration, shown in Figure 6.7a, we eliminated the influence of actual disk accesses by ensuring that all file data and metadata was kept in the buffer cache. In the resulting memory bound configuration, we observed between 30% and 40% performance loss, which we attribute to the cost of the additional memory copy. Compared to the microbenchmark presented in Section 6.2, the use of many concurrent threads exerts higher pressure on the L1/L2 cache, which increases the copying overhead. Figure 6.7b shows the relative performance for a mixed workload that includes random reads and random writes. Here we allow the Linux kernel and the storage domain to pursue their usual write back policies. Both systems provide roughly similar performance in this case.

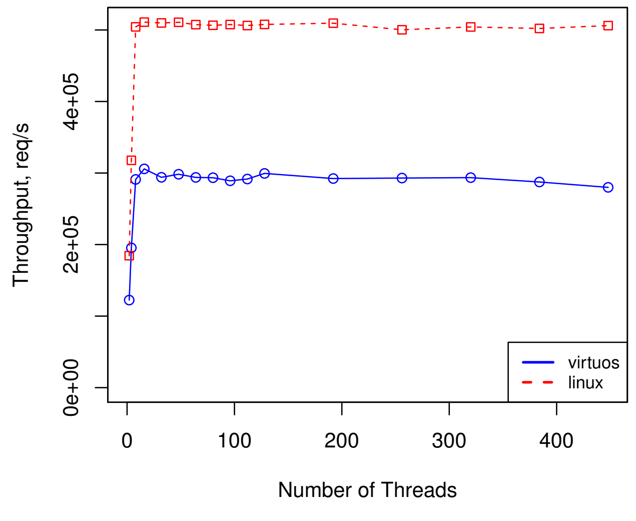


(a) Absolute throughput

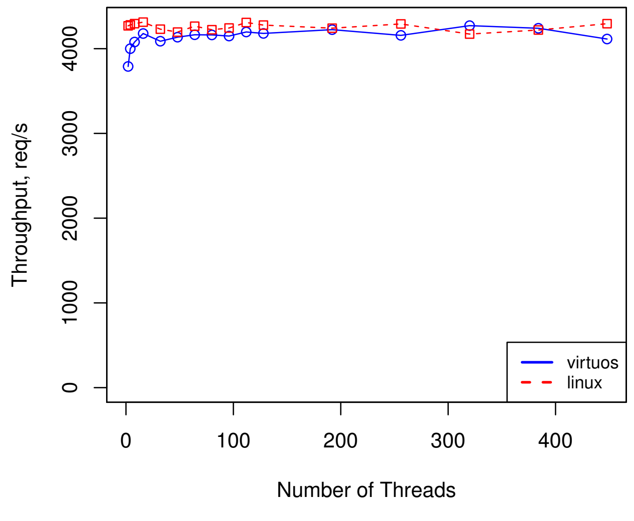


(b) Throughput gain

Figure 6.6: Sysbench/OLTP mySQL throughput.



(a) Without disk accesses



(b) With disk accesses

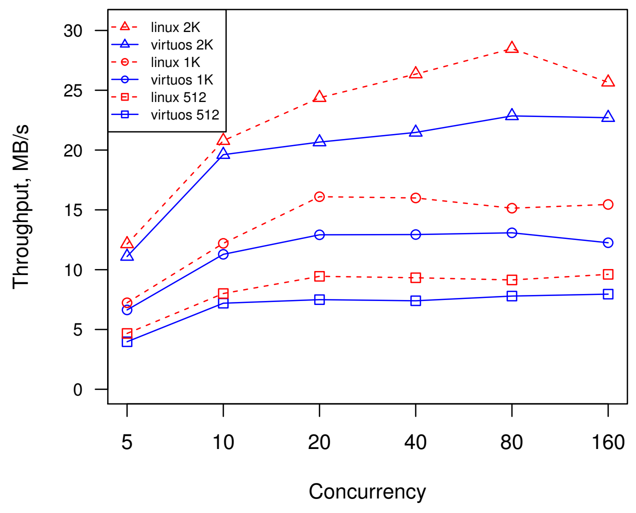
Figure 6.7: FileIO/SysBench throughput.

Taken together, these benchmarks show that for multithreaded workloads which benefit from M:N threading, it is possible to achieve performance that is at least as good as native Linux's.

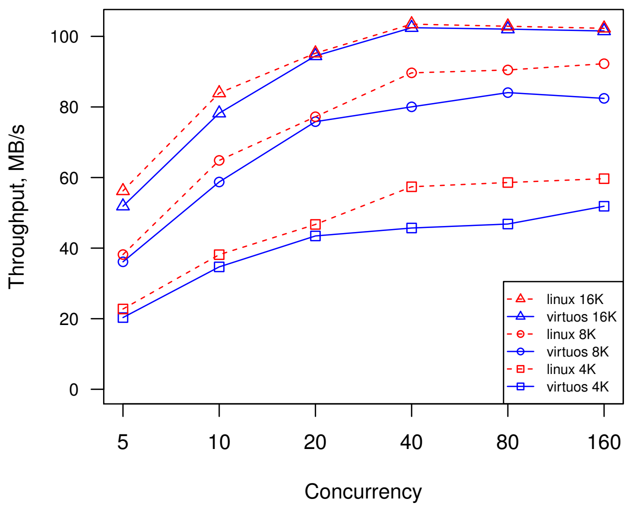
6.3.3 Multiprocess programs

We also tested VirtuOS with single-threaded, multiple process applications such as Apache 2, and compared its performance with native Linux. Single-threaded applications cannot directly utilize benefits of the M:N thread model and, hence, may require notification if system calls do not complete within the spinning threshold.

We used the Apache benchmark utility (*ab*) to record throughput while retrieving objects of various sizes. We present throughput for small and large objects in Figures 6.8a and 6.8b respectively. Both Linux and VirtuOS are able to saturate the outgoing Gigabit link for objects >= 16KB in size; for smaller sizes, VirtuOS's performance lags that of Linux by up to 20%. Adjusting the spinning threshold in either direction did not improve those numbers.



(a) Small requests



(b) Large requests

Figure 6.8: Apache throughput.

Table 6.2: Number of service domain VCPUs and kernel threads.

Program	Primary Domain	Service Domain	M:N
mySQL	8 VCPUs	1 VCPU	M:18
FileIO	4 VCPUs	4 VCPUs	M:4
Apache	8 VCPUs	3-4 VCPUs	N/A

6.3.4 Concurrency Tunables

We found that the results in Sections 6.3.2 to 6.3.3 are sensitive to choosing multiple parameters correctly. Table 6.2 shows the assignment that worked best for our benchmarks. We found that we needed to provide as many VCPUs to the primary domain as there are physical cores (8) for many tests, except for the FileIO benchmarks, where we needed to limit to number of VCPUs available to the primary domain to ensure enough CPU capacity for the service domain. The optimal number of VCPUs assigned to the respective service domain varied by benchmark. We let the Xen scheduler decide on which cores to place those VCPUs because we found that pinning VCPUs to cores did not result in higher performance, except for the FileIO benchmarks, where assigning the VCPUs of the primary domain to the cores of one physical CPU and the VCPUs of the service domain to the other resulted in approx. 15% higher throughput. For our Linux baseline, we used all 8 cores.

In addition, we observed that it is also beneficial to tune the maximum number of kernel threads created by our M:N library for the multi-threaded workloads. We note that such a limit may lead to deadlock if all available kernel threads are blocked in local system calls and the threading library does not create new kernel threads on demand, which our current prototype does not implement.

6.4 Failure Recovery

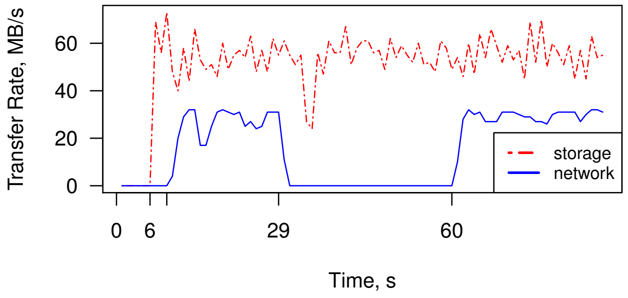


Figure 6.9: Failure recovery scenario.

VirtuOS supports failure recovery for any faults occurring in service domains, including memory access violations, interrupt handling routine failure and deadlocks. Such faults may cause the affected domain to reboot; otherwise, the domain must be terminated and restarted using Xen's toolkit utilities. We provide a cleanup utility to unregister terminated service domains, free any resources the primary domain has associated with them, and unblock any threads waiting for a response from the service domain. We do not currently provide a fault detector to detect when a domain should be restarted. Our recovery time is largely dominated by the amount of time it takes to reboot the service domain (approx. 20 seconds in our current configuration).

We designed an experiment to demonstrate that (1) a failure of one service domain does not affect programs that use another one; (2) the primary domain remains viable, and it is possible to restart affected programs and domains. In this scenario, we run the Apache server which uses the network domain. A remote client connects to the server and continually retrieves objects from it while recording the number of bytes transferred per second. To utilize the storage domain, we launch the Unix `dd` command to sequentially write to a file. We record the number of bytes written per second by observing the increase in used disk space during the same second interval.

Figure 6.9 shows the corresponding transfer rates. At instant 0, the Apache server is launched. At instant 6, the `dd` command starts writing data to the disk. At instant 9, a remote client connects to the Apache server and starts using it. At instant 29, the network domain is abruptly terminated, reducing the client's observed transfer rate to 0, without affecting the ability of `dd` to use the storage domain. At instant 60, the network domain and the Apache server are restarted, and the remote client continues transferring data.

Chapter 7

Related Work

7.1 System Designs

OS fault containment and security have traditionally been both OS design goals as well as the subject of ongoing research. Thus, VirtuOS shares its design goals both with many existing systems and research prototypes.

7.1.1 Kernel Design

A number of approaches decompose kernel functionality to provide better fault containment for system components. Microkernel-based design addresses fault containment by providing better component isolation. Microkernels such as Mach [10] and L4 [66] provide an architecture in which only essential functionality such as task scheduling and message-based interprocess communication is implemented inside the kernel, whereas most other system components, including device drivers, are implemented in separate user processes. Aside from optimizing IPC performance, microkernel-based systems often devote substantial effort to creating compatibility layers for the existing system APIs, e.g. POSIX [53].

Multiserver OS designs such as Sawmill [26,45,91] pursue the opposite approach by attempting to deconstruct a monolithic kernel's functionality into separate servers running on top of a microkernel. Multiserver OS differ from VirtuOS in the methods used for communication and protection. Moreover, VirtuOS does not currently attempt to address all goals multiserver OS address, such as supporting system-wide policies or resource sharing [45].

Recently, microkernels have been used in lieu of hypervisors. Microvisors [51] expose abstractions typical to hypervisors such as VCPUs, memory address space containers, and communication channels to run virtual machines with guest OS on top of a microkernel.

An exokernel-based OS [39] is an alternative OS design which allows applications to directly access hardware resources. Applications in exokernel-based OS do not require a mode switch to access OS services because applications are linked against their own library OS and call corresponding OS service functions directly. When a library OS needs to access global state, a system call to the exokernel is needed.

Dune [18] uses hardware-assisted VMs to isolate applications from each other. Nested paging allows applications to switch their page tables efficiently. Process context switches benefit from TLB tagging available for hardware-assisted VMs. Dune also improves signal delivery latencies by delivering hardware interrupts directly to user processes. However, Dune still uses a monolithic kernel which does not isolate device drivers from each other. Additionally, applications may have to use more expensive hypercalls in lieu of system calls to access OS services.

The nonkernel [19] also runs applications in separate hardware-assisted VMs while allowing them to have direct access to hardware in a manner that is similar to exokernels. To ensure safe access to devices, it relies on IOMMU and self-virtualized I/O. The nonkernel is specifically aimed at infrastructure as a service (IaaS) applications, which allow executing multiple VMs on shared hardware.

7.1.2 Driver Protection Approaches

Because most kernel failures are caused by faulty device drivers, there is a particular focus on making them safer and isolating them from other system components. Nooks [92] introduced hardware protection domains inside a monolithic kernel to isolate device drivers from each other and from the remaining kernel. Such isolation protects against buggy drivers that may perform illegal memory accesses. Nooks demonstrated how to restructure an existing kernel's interaction with its drivers to facilitate the use of intrakernel protection domains, and explored the trade-off between benefits due to isolation and costs imposed by the domain crossings this approach requires. This approach requires drivers to be adapted to the new mechanism, as their interaction with the OS kernel changes. Switching to and from protection domains requires page table switches, along with corresponding TLB flushes, which if done frequently may affect the performance of some applications.

Microdrivers [44] split drivers into parts running inside the kernel and parts running as user processes. In microdrivers, hardware-specific and performance critical code remains in the kernel whereas the remaining code is moved to user space to provide better isolation. Additionally, code running in user space can be written in a higher-level language [80]. Mainstream OS have provided support for writing device drivers that execute in user mode for some time, but these facilities have not been widely used because the added context switches made it difficult to achieve good performance [63]. Some systems provide the ability to run unchanged kernel components such as out-of-box Linux drivers in user mode. DD/OS [64] provides this ability by creating a virtual machine built as a user-level task running on top of L4, whereas SUD [21] provides such an environment inside ordinary Linux user processes. Xen Driver Domains [85] use a mechanism with comparable protection properties by running unchanged drivers in specialized guest OS. The Qubes OS [8] adopted the Xen hypervisor and Xen Driver Domains to enhance security by running separate virtual machines for drivers and applications. The Qubes OS supports a storage domain, a network domain and application virtual machines.

7.1.3 Language and Protocol-based Approaches

Researchers have also proposed to make kernel components safer by using language and protocol-based techniques. SafeDrive [99] uses a type system approach to provide fine-grained isolation of kernel components written in C, which relies upon a combination of compile-time analysis and runtime checking. Carburizer [57] analyzes and modifies driver code to withstand hardware failures by removing the assumption that the underlying hardware behaves according to its specification. Static analysis approaches have been used to find bugs and improve the reliability of systems code [16, 38], as well as approaches derived from model checking [96]. Domain-specific languages can reduce race conditions, deadlocks and protocol violations by formally describing the driver's expected software interface (e.g., Dingo [84]). Despite their advantages and often negligible overhead, the approaches do not aim to prevent all crashes due to kernel component failures. Furthermore, they require co-operation on the part of the developers since they mandate changes to kernel components or the way they interact with the system.

7.1.4 Restartability

A number of systems attempt to minimize the impact of failures of isolated components and to speed up recovery after failures. The microreboot approach [28] advocates designing server applications as loosely coupled, well-isolated, stateless components, which keep important application state in specialized state stores. In doing so, individual components can be quickly restarted with limited loss of data. CuriOS [33] applies similar ideas to a microkernel-based OS. VirtuOS's use of existing kernel code in its service domains prevents us from using this approach since monolithic kernels make free and extensive use of shared data structures. Fine-grained fault tolerance (FGFT [58]) uses device state checkpoints to restore the functionality of drivers after a failure, but it so far has been applied only to individual drivers rather than the state of an entire kernel.

7.1.5 Security

Some solutions are specifically focused on improving the security of existing systems. McAfee DeepSafe [68] proposed a virtualization-based solution which improves OS security by adding security layers underneath the OS kernel, and thus providing better protection from malware. NICKLE [81] focuses on kernel rootkit detection which relies on a trusted hypervisor that checks kernel code and disallows any unauthorized operations. These solutions primarily aim to protect operating systems from intrusion by attackers. They do not address a problem of containing (usually unintentional) device driver bugs.

7.2 Performance Optimizations

Apart from improving the robustness and security of OS, there have also been substantial efforts to improve performance of existing OS, which are related to our work.

7.2.1 Exceptionless techniques

Multiple systems deploy exceptionless techniques: FlexSC [88,89] proposed the use of exceptionless system calls, which we adopted in VirtuOS, for the purposes of optimizing system call performance in a monolithic kernel.

Factored operating systems (fos) [95] use a microkernel-based OS design with servers running on dedicated cores. The design can increase scalability compared to traditional systems due to reduced TLB and cache contention. fos uses proxy libraries to handle system calls coming from user programs. Proxy libraries generate messages and then make calls to a microkernel. The latter delivers a request to a server running on dedicated cores. Corey [20] uses a similar approach; it dedicates kernel cores and handle requests from application cores through shared-memory IPC. Both of these systems are designed for scalability on multicore systems and distribute OS services across cores. VirtuOS shares with these systems the assumption that the increasing availability of cores makes their dedication for systems-related tasks beneficial.

7.2.2 Improving Virtual Machines

VirtuOS relies on an underlying hypervisor, and could benefit from a number of orthogonal ideas that were introduced to improve virtual machine technology. For instance, self-virtualizing hardware [79] makes it easier to safely and efficiently multiplex devices across domains, which would allow multiple service domains to share a device. Spinlock-aware scheduling [98] modifies the hypervisor's scheduler to avoid descheduling VCPUs that execute code inside a critical region protected by a spinlock, which could adversely affect the performance of service domains.

Low inter-domain communication costs improve the performance of any system that uses virtual machines. The Fido system [25] optimizes Xen's interdomain communication facilities by allowing read-only data mappings to enable zero-copy communication. Fido amortizes costs but users need to sacrifice some security and protections guarantees.

The Xoar system [31] addresses the problem of improving the manageability and robustness of complex hypervisors with dedicated control VMs (i.e., Dom0 in Xen) by splitting the control VM into multiple, individually restartable VMs.

7.3 Performance Monitoring

7.3.1 Monitoring in Virtualized Environments

There has been a number of previous efforts to add support for hardware event counters to virtualized environments. XenoProf [71], which is integrated in Xen, allows the use of event counters for system-wide monitoring and profiling. It is an extension of the OProfile Linux system-wide profiler. Each monitored domain runs an instance of OProfile with a Xen-specific driver, which communicates with XenoProf in the hypervisor. XenoProf collects PC samples and puts them into shared buffer. Then it notifies the corresponding domain via a virtual interrupt, so that it can map the PC sample to a specific symbol in the executable. XenoProf does not support performance counter virtualization (i.e., the simultaneous monitoring of multiple domains), works only in paravirtualized mode, is specific to OProfile and cannot be easily adapted to work with other higher-level toolkits such as PAPI or HPCToolkit.

Work concurrent with ours [36, 37] implemented performance counter virtualization for the hardware-assisted KVM virtual machine monitor that is included in recent versions of the Linux kernel. Like Xen's VPMU driver discussed in Section 5.2.2, this implementation uses a save-and-restore mechanism for PMU registers. During interdomain context switches, the hypervisor saves and restores the PMU registers of a domain. The delivery of overflow interrupts to a domain relies on hardware support provided by architectural virtualization extensions. Such full virtualization approaches have the advantage that they do not require any accommodations to the guest kernel or user libraries, and thus allow the use of virtually any framework in the target domain, but they forgo the potential optimizations arising from guest kernel adaptation, such as the offsetting technique for a-mode counters discussed in Section 5.2.2. In addition, each instruction that changes a configuration register requires a separate trap to emulate its effect, whereas the use of hypercalls allows the batching of such changes by combining them into a single call.

The VTSS++ system uses a system-wide global sampling mechanism that records time-stamped event counter values [22]. The global TSC register is used to obtain these time stamps. In addition, the system records the time stamps of all intra- and inter-domain context switches. An off-line post-processing system then reconstructs the events produced by individual threads and domains. This method has the advantage that no guest kernel or user-level provisions are required, but its reliance on post-processing makes analysis more complicated.

The vmkperf utility [4] used by VMware ESX allows the counting of events occurring within given time intervals, similar to a-mode counters. vmkperf does not support the functionality of i-mode counters and therefore cannot easily be used to support high-level profiling toolkits.

7.3.2 Linux Frameworks

The perf_events (previously known as perf_counter) framework [46] provides performance monitoring capabilities similar to those of *perfctr*. This framework has been integrated in recent versions of the Linux kernel. Like *perfctr*, it supports per-thread counters and direct user mode access. Higher-level frameworks such as PAPI include support for perf_events, although the recently added direct access feature [67] is not currently supported in PAPI. The techniques we presented in this work are applicable to perf_events as well; a virtualization of perf_events is possible future work. The tight integration of perf_events with the Linux kernel may require a substantial amount of refactoring in order to create a hypervisor driver. This coupling may make it more difficult to exploit a single code base for the guest and hypervisor driver as done in our Perfctr-Xen implementation (see Section 5.2.2).

The perfmon framework [40] for Linux provides both low-level and high-level features. The framework supports per-thread monitoring and sampling, although it relies exclusively on system calls to access counter data. The Intel VTune performance analyzer [5] and AMD's Code Analyst [2, 35] are proprietary frameworks for precise, low-overhead event sampling. They consist of a kernel driver and high-level infrastructure that provides result analysis capabilities. Event counts can be viewed on per-thread or per-module basis. Our techniques could be applied to a possible virtualization of perfmon, VTune, and CodeAnalyst.

Chapter 8

Conclusion

This work presented VirtuOS, a fault-resilient operating system design which provides isolation for kernel components by running them in virtualized service domains. Service domains are constructed by carving a vertical slice out of an existing Linux kernel for a particular service, such as networking or storage.

VirtuOS allows processes to directly communicate with service domains via exceptionless system call dispatch. Thus, user processes can transparently benefit from isolated service domains without requiring the use of a special API. A special-purpose user-level threading library allows service domains to efficiently resume threads upon system call completion via direct queue manipulation. To the best of our knowledge, VirtuOS is the first system to use virtual machines for system call dispatch and to apply exceptionless communication across virtual machines.

To make the analysis of VirtuOS's performance overhead feasible, we developed Perfctr-Xen, a novel performance counter framework for the Xen hypervisor. Perfctr-Xen extends the existing *perfctr* framework so it can be used in virtual machine environments running under the Xen hypervisor. Perfctr-Xen enables the use of higher-level profiling frameworks such as PAPI or HPCToolkit in those environments, without requiring changes to them. Perfctr-Xen supports both paravirtualized guests and guests using hardware-based virtualization, thus making possible per-thread analysis in primary and service domains of VirtuOS.

We have tested our VirtuOS prototype with several existing applications; our experimental evaluation has shown that our design has the potential to outperform not only existing solutions for driver isolation but can for concurrent workloads that benefit from M:N threading meet and exceed the performance of a traditional, monolithic Linux system using exception-based system call dispatch.

References

[1] TTCP tool, 2007. http://www.netcore.fi/pekkas/linux/ipv6/ttcp.c.

[2] AMD CodeAnalyst, 2011. http://developer.amd.com/cpu/codeanalyst/.

[3] AMD I/O Virtualization Technology (IOMMU) Specification, 2011. http://support.amd.com/TechDocs/48882.pdf.

[4] Vmkperf utility for VMWare ESX 4.0, 2011.

[5] VTune amplifier profiler, 2011. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[6] Kernel Asynchronous I/O (AIO), 2012. http://lse.sourceforge.net/io/aio.html.

[7] Intel Virtualization Technology for Directed I/O, Architecture Specification, 2013. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[8] Qubes OS, 2013. http://qubes-os.org/.

[9] Xen PCI Passthrough, 2013. `http://wiki.xen.org/wiki/Xen_PCI_Passthrough`.

[10] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference* (1986), pp. 93–112.

[11] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, USA, 2006), ASPLOS XII, pp. 2–13.

[12] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701.

[13] Agesen, O., Garthwaite, A., Sheldon, J., and Subrahmanyam, P. The evolution of an x86 virtual machine monitor. *SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 3–18.

[14] Andersen, E. uClibc C library, 2012. `http://uclibc.org/`.

[15] Back, G., and Nikolopoulos, D. S. Application-specific system customization on many-core platforms: The VT-ASOS framework. In *STMCS: Second Workshop on Software Tools for Multi-Core Systems (STMCS)* (San Jose, CA, Mar. 2007).

[16] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems* (Leuven, Belgium, 2006), EuroSys'06, pp. 73–85.

[17] Barham, P., Dragovic, B., Fraser, K., and et al. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP'03, pp. 164–177.

[18] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design & Implementation* (Hollywood, CA, USA, 2012), OSDI'12, pp. 335–348.

[19] Ben-Yehuda, M., Peleg, O., Ben-Yehuda, O. A., Smolyar, I., and Tsafrir, D. The Nonkernel: A Kernel Designed for the Cloud. In *Proceedings of the 4th ACM Asia-Pacific Workshop on Systems* (Singapore, Singapore, 2013), APSys'13, pp. 4:1–4:7.

[20] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design & Implementation* (San Diego, CA, 2008), OSDI'08, pp. 43–57.

[21] Boyd-Wickizer, S., and Zeldovich, N. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Boston, MA, USA, 2010), ATC'10, pp. 117–130.

[22] Bratanov, S., Belenov, R., and Manovich, N. Virtual machines: a whole new world for performance analysis. *SIGOPS Oper. Syst. Rev. 43* (April 2009), 46–55.

[23] Browne, S., Deane, C., Ho, G., and Mucci, P. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference* (June 1999).

[24] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst. 15*, 4 (1997), 412–447.

[25] Burtsev, A., Srinivasan, K., Radhakrishnan, P., Bairavasundaram, L. N., Voruganti, K., and Goodson, G. R. Fido: fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, CA, USA, 2009), ATC'09, pp. 313–326.

[26] Bushnell, T. Towards a new strategy for OS design, 1996. `http://www.gnu.org/software/hurd/hurd-paper.html`.

[27] Camargos, F. L., Girard, G., and Ligneris, B. D. Virtualization of Linux servers: a comparative study. In *2008 Ottawa Linux Symsposium* (July 2008), pp. 63–76.

[28] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation* (San Francisco, CA, USA, 2004), OSDI'04, pp. 31–44.

[29] Chisnall, D. *The definitive guide to the xen hypervisor*, first ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[30] Cohen, W. Multiple architecture characterization of the build process with OProfile, Red Hat, 2003. `http://people.redhat.com/wcohen/wwc2003/`.

[31] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP'11, pp. 189–202.

[32] Creasy, R. The origin of the VM/370 time-sharing system. *Softw. World (UK) 13*, 1 (1982), 4 – 10.

[33] David, F. M., Chan, E. M., Carlyle, J. C., and Campbell, R. H. CuriOS: improving reliability through operating system structure. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design & Implementation* (San Diego, CA, USA, 2008), OSDI'08, pp. 59–72.

[34] Drepper, U., and Molnar, I. The native POSIX thread library for Linux, 2005. `http://www.akkadia.org/drepper/nptl-design.pdf`.

[35] Drongowski, P., Yu, L., Swehosky, F., Suthikulpanit, S., and Richter, R. Incorporating instruction-based sampling into AMD CodeAnalyst. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (Mar. 2010), pp. 119 –120.

[36] Du, J., Sehrawat, N., and Zwaenepoel, W. Performance profiling in a virtualized environment. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (Boston, MA, USA, 2010), HotCloud'10.

[37] Du, J., Sehrawat, N., and Zwaenepoel, W. Performance profiling of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Newport Beach, CA, USA, 2011), VEE'11.

[38] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, 2001), SOSP'01, pp. 57–72.

[39] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, USA, 1995), SOSP'95, pp. 251–266.

[40] Eranian, S. Perfmon2: A flexible performance monitoring interface for linux. In *Ottawa Linux Symposium* (Ottawa, Canada, 2006), pp. 269–288.

[41] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proceedings of the 2th USENIX Symposium on Operating Systems Design & Implementation* (Seattle, WA, USA, 1996), OSDI'96, pp. 137–151.

[42] Fraser, K., Steven, H., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure* (2004), OASIS'04.

[43] Ganapathi, A., Ganapathi, V., and Patterson, D. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX conference on Large Installation System Administration* (Washington, DC, USA, 2006), LISA'06, pp. 149–159.

[44] Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., and Jha, S. The design and implementation of microdrivers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA, 2008), ASPLOS'13, pp. 168–178.

[45] Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J. E., Deller, L., and Reuther, L. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop* (Kolding Denmark, 2000), pp. 109–114.

[46] Gleixner, T. Performance counters for Linux, 2008. `http://lwn.net/Articles/310176/`.

[47] Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, 2009), SOSP '09, pp. 103–116.

[48] Habib, I. Virtualization with KVM. *Linux Journal 2008*, 166 (2008), 8.

[49] Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems* (Santa Fe, NM, 2005), HOTOS'05.

[50] Härtig, H., Hohmuth, M., Liedtke, J., Wolter, J., and Schönberg, S. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint Malo, France, 1997), SOSP'97, pp. 66–77.

[51] Heiser, G., and Leslie, B. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems* (New Delhi, India, 2010), APSys'10, pp. 19–24.

[52] Heiser, G., Uhlig, V., and LeVasseur, J. Are virtual-machine monitors microkernels done right? *SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 95–99.

[53] Helander, J. Unix under Mach: The Lites Server. Master's thesis, Helsinki University of Technology, 1994.

[54] Herder, J., Moolenbroek, D., Appuswamy, R., Wu, B., Gras, B., and Tanenbaum, A. Dealing with driver failures in the storage stack. In *Proceedings of the 4th Latin-American Symposium on Dependable Computing* (Joao Pessoa, Brazil, 2009), LADC'09, pp. 119–126.

[55] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. The architecture of a fault-resilient operating system. In *Proceedings of 12th ASCI Conference* (Lommel, Belgium, 2006), ASCI'06, pp. 74–81.

[56] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[57] Kadav, A., Renzelmann, M. J., and Swift, M. M. Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, 2009), SOSP'09, pp. 59–72.

[58] Kadav, A., Renzelmann, M. J., and Swift, M. M. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, TX, USA, 2013), ASPLOS'13, pp. 473–484.

[59] Karlin, A. R., Li, K., Manasse, M. S., and Owicki, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th*

ACM Symposium on Operating Systems Principles (Pacific Grove, CA, USA, 1991), SOSP'91, pp. 41–55.

[60] King, S. T., Dunlap, G. W., and Chen, P. M. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, USA, 2003), ATEC'03, pp. 71–84.

[61] Kivity, A. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium* (July 2007), pp. 225–230.

[62] Kopytov, A. SysBench 0.4.12 – A System Performance Benchmark, 2013. `http://sysbench.sourceforge.net/`.

[63] Leslie, B., Chubb, P., Fitzroy-Dale, N., Gotz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y. R., Elphinstone, K., and Heiser, G. Userlevel device drivers: Achieved performance. *Journal of Computer Science and Technology 20*, 5 (Sept. 2005), 654–664.

[64] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation* (San Francisco, CA, USA, 2004), OSDI'04, pp. 17–30.

[65] Liedtke, J. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, USA, 1993), SOSP'93, pp. 175–188.

[66] Liedtke, J. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, USA, 1995), SOSP'95, pp. 237–250.

[67] Mackerras, P. Perf_counter direct access support, 2009. `http://lwn.net/Articles/323891/`.

[68] McAfee, I. A new paradigm shift: Comprehensive security beyond the operating system.

[69] McVoy, L., and Staelin, C. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference* (San Diego, CA, USA, 1996), ATEC'96, pp. 279–294.

[70] McVoy, L., and Staelin, C. LMbench – Tools for Performance Analysis, 2013. `http://lmbench.sourceforge.net/`.

[71] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Chicago, IL, USA, 2005), VEE'05, pp. 13–23.

[72] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, PA, USA, 1996), PODC'96, pp. 267–275.

[73] Murphy, B. Automating software failure reporting. *Queue 2*, 8 (Nov. 2004), 42–48.

[74] Nikolaev, R., and Back, G. Perfctr-Xen: a framework for performance counter virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Newport Beach, CA, USA, 2011), VEE'11, pp. 15–26.

[75] Nikolaev, R., and Back, G. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, USA, 2013), SOSP'13, pp. 116–132.

[76] Oglesby, R., and Herold, S. *VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide series)*. The Brian Madden Company, 2005.

[77] Pettersson, M. Perfctr library, 2011. `http://user.it.uu.se/~mikpe/linux/perfctr/`.

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

[79] Raj, H., and Schwan, K. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing* (Monterey, CA, USA, 2007), HPDC'07, pp. 179–188.

[80] Renzelmann, M. J., and Swift, M. M. Decaf: moving device drivers to a modern language. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, CA, USA, 2009), ATC'09, pp. 187–200.

[81] Riley, R., Jiang, X., and Xu, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Cambridge, MA, USA, 2008), RAID'08, Springer-Verlag, pp. 1–20.

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

[83] Rosenblum, M. The reincarnation of virtual machines. *Queue 2*, 5 (2004), 34–40.

[84] Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. Dingo: taming device drivers. In *Proceedings of the 4th ACM SIGOPS European Conference on Computer systems* (Nuremberg, Germany, 2009), EuroSys'09, pp. 275–288.

[85] Santos, J. R., Turner, Y., Janakiraman, G., and Pratt, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, Massachusetts, 2008), ATC'08, pp. 29–42.

[86] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M. S., and Rosenblum, M. Virtual appliances for deploying and maintaining soft-

ware. In *Proceedings of the 17th USENIX conference on Large Installation System Administration* (San Diego, CA, 2003), LISA'03, pp. 181–194.

[87] Shende, S., and Malony, A. D. TAU: The TAU parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[88] Soares, L., and Stumm, M. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (Vancouver, BC, Canada, 2010), OSDI'10, pp. 1–8.

[89] Soares, L., and Stumm, M. Exception-less system calls for event-driven servers. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR, USA, 2011), ATC'11, pp. 131–144.

[90] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Operating Systems Review 41*, 3 (Mar. 2007), 275–287.

[91] Stevenson, J. M., and Julin, D. P. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference* (New Orleans, Louisiana, 1995), TCON'95, pp. 119–130.

[92] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP'03, pp. 207–222.

[93] Torvalds, L. cmpxchg, ll/sc, and portability, 2006. `http://yarchive.net/comp/linux/cmpxchg_ll_sc_portability.html`.

[94] Treiber, R. K. Systems Programming: Coping with Parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center, Apr. 1986.

[95] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review 43*, 2 (Apr. 2009), 76–85.

[96] Yang, J., Sar, C., and Engler, D. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation* (Seattle, WA, USA, 2006), OSDI'06, pp. 131–146.

[97] Zaparanuks, D., Jovic, M., and Hauswirth, M. Accuracy of performance counter measurements. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009).* (Apr. 2009), pp. 23–32.

[98] Zhong, A., Jin, H., Wu, S., Shi, X., and Gen, W. Optimizing Xen hypervisor by using lock-aware scheduling. In *Proceedings of the Second International Conference on Cloud and Green Computing* (2012), CGC'2012, pp. 31–38.

[99] Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. SafeDrive: safe and recoverable extensions using

language-based techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation* (Seattle, WA, USA, 2006), OSDI'06, pp. 45–60.

Appendix A

Annotated List of Figures

Appendix B

Source Code

VirtuOS's and Perfctr-Xen's source code is available at `http://people.cs.vt.edu/~rnikola/` under various open source licenses.