Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Virtual Machine Benchmarking

Kim-Thomas Möller

Diploma Thesis

Advisors:
Prof. Dr. Frank Bellosa
Joshua LeVasseur

17. April 2007

I hereby declare that this thesis is the result of my own work, and that all information sources and literature used are indicated in the thesis. I also certify that the work in this thesis has not previously been submitted as part of requirements for a degree.

Hiermit erkläre ich, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Literatur und Hilfsmittel angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Karlsruhe, den 17. April 2007

Kim-Thomas Möller

Abstract

The resurgence of system virtualization has provoked diverse virtualization techniques targeting different application workloads and requirements. However, a methodology to compare the performance of virtualization techniques at fine granularity has not yet been introduced. VMbench is a novel benchmarking suite that focusses on virtual machine environments. By applying the pre-virtualization approach for hypervisor interoperability, VMbench achieves hypervisor-neutral instrumentation of virtual machines at the instruction level. Measurements of different virtual machine configurations demonstrate how VMbench helps rate and predict virtual machine performance.

Kurzfassung

Das wiedererwachte Interesse an der Systemvirtualisierung hat verschiedenartige Virtualisierungstechniken für unterschiedliche Anwendungslasten und Anforderungen hervorgebracht. Jedoch wurde bislang noch keine Methodik eingeführt, um Virtualisierungstechniken mit hoher Granularität zu vergleichen. VMbench ist eine neuartige Benchmarking-Suite für Virtuelle-Maschinen-Umgebungen. Indem sie den Pre-Virtualisierungs-Ansatz für Hypervisor-Interoperabilität anwendet, erzielt die VMbench-Suite Hypervisor-neutrale Instrumentierung von virtuellen Maschinen auf der Befehlsebene. Messungen verschiedener Konfigurationen von virtuellen Maschinen zeigen, wie VMbench dabei hilft, die Leistung von virtuellen Maschinen zu bewerten und vorherzusagen.

Acknowledgements

Primarily, I would like to thank my supervisors, Prof. Dr. Frank Bellosa and Joshua LeVasseur, for giving me the opportunity to work on a fascinating project. They wisely guided my research, pointed me to interesting issues, and encouraged novel ideas.

Furthermore, I am thankful to my fellow students and to the staff of the system architecture chair for fruitful discussions and for kindly providing me with hardware and software resources.

Special thanks go to Sonja for her continuous love and support. My family encouraged my studies in computer science and also supported me financially. I would like to thank my friends, who motivated me and enriched life outside university in various ways.

Contents

Chapter 1

Introduction

The introductory chapter of this thesis presents the fundamentals of virtualization and justifies the relevance of virtual machine benchmarking.

1.1 Performance of Virtual Machines

Ever-increasing computing power on the server market calls for simplified server management and good hardware utilization. In the recent years, system virtualization [SN05] has become a key technique to improve maintainability and to leverage server consolidation: Rather than on raw hardware, operating systems and application software run in virtual machines, which are controlled by a hypervisor.

The hypervisor flexibly defines which resources are available to the virtual machines, providing extensibility of the virtual machine environment and resource control. Virtual machines are mutually isolated, such that different operating systems and applications can run side-by-side on one physical host.

Unfortunately, flexibility does not come for free. Virtualization adds complexity to the system, reducing performance of certain operations. Making the operating system aware of being virtualized helps avoid costly operations but binds it to one specific hypervisor. The pre-virtualization technique [LUC+05], which was recently developed by Joshua LeVasseur, overcomes the hypervisor lock-in by preparing an operating system kernel for use on raw hardware as well as with different hypervisors.

However, every virtualization technique adds a certain performance overhead, which depends on a number of factors, such as the hardware, the virtualization technique, the hypervisor, and the configuration of the virtual machines [You73].

1.2 Hot Topics in Virtualization Research

The interest in understanding virtual machine performance derives from the wide range of applications for virtualization and the diversity of hypervisors.

The indirection through the hypervisor [WCSG05] enables inspection of the virtual machine from the outside. For example, intrusion detection [DKC$^+$02] and virtual machine migration [CFH$^+$05, NLH05] can be realized on top of a hypervisor. By virtue of indirection, virtual machines are a suitable basis for grid computing [FDF03] and trusted computing [GPC$^+$03, ELM$^+$03].

Isolation of the virtual machines brings forward several improvements to device drivers and resource management. Controlled by a trusted hypervisor, high-performance applications and legacy operating systems coexist on the same physical host [MUKX06]. Virtualization supports dependable [BS95] or even secure [GCGV06] device drivers. For example, the first release of the Xen hypervisor contained all device drivers in its kernel [BDF$^+$03], whereas the second release allowed user-level device drivers to access physical hardware [FHN$^+$04]. Cumbersome porting of device drivers to a new operating system is avoided by reusing unmodified device drivers in their native operating system that runs in a dedicated device driver virtual machine (DD/VM) [LUSG04].

Consolidation of physical servers helps reduce costs, because data centers need to acquire less hardware, and less hardware consumes less electrical power [SN05].

In order to permit hypervisor interoperability, several efforts [VMwb, CI06] define hypervisor-neutral interfaces, that is, general para-virtualization APIs without binding to a specific hypervisor. The pre-virtualization approach [LUC$^+$06] combines the advantages of several virtualization techniques using the soft-layering concept [Coo83]: At build time, the operating system kernel is automatically prepared for being virtualized, but it binds to the actual hypervisor only at boot time of the virtual machine.

1.3 A Case for Virtual Machine Benchmarking

Only a few years after the revival of virtual machines, a large number of hypervisors for different purposes exists: Some emphasize good performance of the virtual machines, others focus on special functionality or take advantage of processor virtualization extensions. Moreover, virtual machines are configurable in many ways, differing in what operating system and applications they run as well as in the amount and type of assigned resources.

Then again, users of virtual machines have different minimum requirements and metrics to characterize their servers [CGS06]. However, as economization is a driving force behind virtualization, users do not tolerate significant performance degradation compared to running on raw hardware. Thus, there is great interest in understanding and comparing performance of virtual machines.

Although most work on virtualization includes a performance evaluation (see for example [WSG02, BDF$^+$03]), the results can hardly be compared, because different hardware, different guest software, and different benchmarking suites are used. Even reproducing performance measurements from previous work when source code and similar hardware are available can turn out to be a nontrivial task in case imitating the original setup is cumbersome [CDD$^+$].

In the recent two years, several approaches have been developed to assess the performance of virtual machines [CGS06, MHS$^+$, AA06, MST$^+$05, GGC]. High-level performance evaluation allows for comparison of virtual machine performance, but it fails at predicting performance for new workload and explaining what contributes to virtual machine performance. Low-level hypervisor-specific performance evaluation enables performance analysis and prediction, but it is not helpful for comprehensive comparison.

Thus, a methodology to investigate the reasons for performance differences and to identify bottlenecks in virtual machine setups is ultimately needed. The methodology must be able to characterize performance of a virtualized system at an adequate level, and to relate low-level and high-level performance.

The goal of this thesis is to develop a benchmarking suite for realistic characterization and rating of virtual machine environments (consisting of hardware, hypervisor, and one or more virtual machines) regarding their overall performance as well as performance in special domains.

1.4 Thesis Structure

Chapter 2 presents background on the pre-virtualization and benchmarking techniques as well as a review of virtual machine benchmarking efforts. Chapter 3 describes the design of VMbench, a benchmarking suite targeting virtual machine environments, and chapter 4 highlights several issues that were solved in its implementation. Chapter 5 contains results and interpretation of the benchmarks. Chapter 6 concludes this thesis with a summary and suggestions for future work.

Chapter 2

Background and Related Work

This chapter gives background information on virtual machine benchmarking. First, I present and compare several techniques for system virtualization. Second, I give an overview of benchmarking methodology. Third, I analyze related work on virtual machine benchmarking, concluding that it does not allow comparison and performance prediction of different virtual machine setups.

2.1 Techniques for System Virtualization

This section gives a short overview over several virtualization paradigms that are currently used, ranging from the original concept of *pure virtualization* to recent software- and hardware-based optimization techniques.

2.1.1 Pure Virtualization

Fundamental research on virtual machines [Gol73] dates back more than thirty years, defining what is known as the *pure virtualization* paradigm: A hypervisor presents a faithful emulation of a hardware interface to one or more guest operating systems running inside virtual machines.

In order to have full control of the hardware, the hypervisor runs at a higher privilege level than the guest operating system kernels. Some processor architectures implement a mode of instruction called hypervisor mode, which surpasses the privileged supervisor or kernel mode [SN05]. On processors that distinguish only two levels of priority the guest kernels run in non-privileged user mode [SN05]. Other processor architectures, such as Intel's IA32 [Int06], have more than two privilege levels, so that the hypervisor deprivileges the guest kernels to an intermediate privilege level [BDF$^+$03].

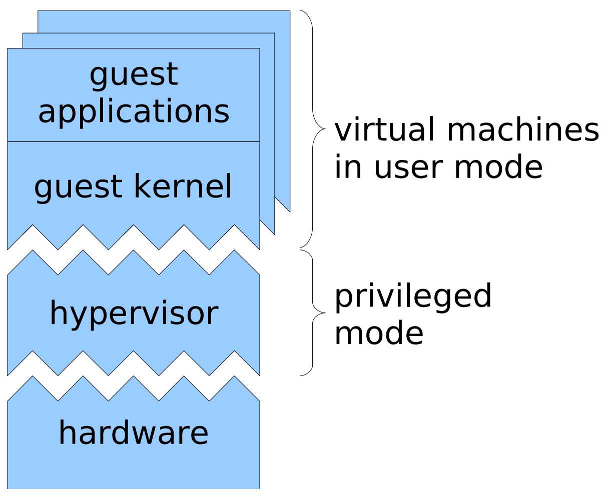

Figure 2.1: A purely virtualized system

In any case, the hypervisor logically detaches the operating system from the physical hardware, such that pure virtualization induces a strictly layered system structure [Dij68] (see Figure 2.1). Strict layering allows for modularization, that is, separation of interface and implementation [PS75]. Thus, pure virtualization facilitates customization of the virtual machine setup such as hypervisor enhancements and recursive construction of VMs [CN01].

Whenever an application executes a system call or experiences a fault, the hypervisor redirects the request to the respective guest operating system. To ensure the containment of the virtual machine, the guest operating system kernel traps into the hypervisor whenever it executes a virtualization-sensitive instruction, that is, when it tries to detect or modify external state of the virtual machine, such as the amount of available resources. In response, the hypervisor emulates the effect of the instruction and modifies physical machine state where necessary. The *VM assist* optimization [Mac79] allows the processor to delay a trap into the hypervisor by caching state changes in the virtual machine's protection domain. Popek and Goldberg [PG74] prove that a computer architecture is virtualizable if all virtualization-sensitive instructions are privileged and trap into the hypervisor (see Figure 2.2). Hypervisors are also known as virtual machine monitors (VMMs) [SN05]. The notion *hypervisor* emphasizes that the virtualizing software has the highest privilege level in the system, whereas the notion *virtual machine monitor* stresses that the virtualizing software gives guest software the illusion of running on a physical machine. Henceforth, the notion *virtual machine environment* refers to the combination of hardware, hypervisor or VMM, and virtual machines.

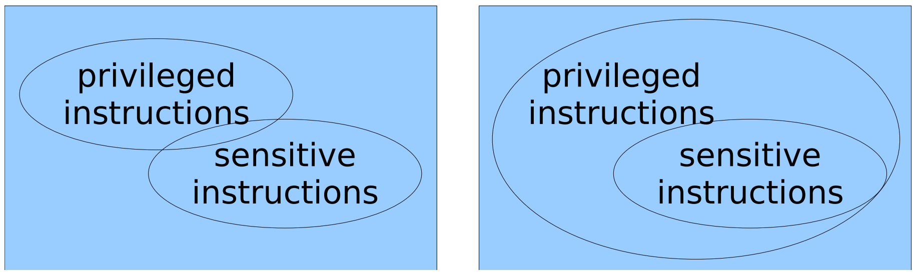

Figure 2.2: Non-virtualizable (left) and virtualizable (right) instruction set

Pure virtualization is a clean concept: It does not require modification of the guest operating system kernel, minimizing engineering effort for virtualization and permitting virtualization of operating systems whose source code is not available.

Unfortunately, pure virtualization has two major drawbacks: First, pure virtualization assumes a virtualizable processor architecture where every virtualization-sensitive operation that a guest kernel executes faults and traps into the hypervisor. Many modern processors architectures such as IA32 and IA64 are not virtualizable [RI00]. Instead, several workarounds have evolved that contend with non-virtualizable hardware [AA06]. Second, pure virtualization imposes a relatively high virtualization overhead on the system, since all virtualization-sensitive activity within the virtual machines must be authorized by the hypervisor [You73, MST$^+$05]. Particularly, switching between privilege and protection domains takes significant time on modern processors. The operating system is not aware of being virtualized, thus it does not avoid frequent virtualization-sensitive operations such as enabling and disabling interrupt delivery.

2.1.2 Para-Virtualization

The para-virtualization concept [WSG02] was developed to overcome pure virtualization's drawbacks. Para-virtualization is based on the idea of modifying the guest operating system kernel to make it virtualization-friendly: First, on non-virtualizable processor architectures, the guest kernel must be aware of being virtualized and correctly handle virtualization-sensitive but non-privileged instructions. Second, the kernel should avoid frequent traps into the hypervisor. Thus, a high-level API replaces virtualization-sensitive parts of the guest operating system kernel, allowing to cache virtual machine state in the guest kernel's protection domain (see Figure 2.3). Virtual machine state is synchronized with the hypervisor by means of hypercalls.

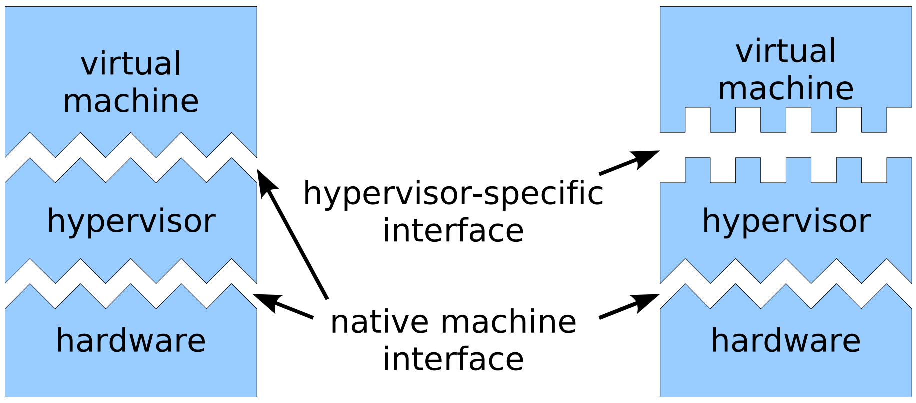

Figure 2.3: Comparison of pure virtualization (left) and para-virtualization (right)

Para-virtualization represents co-design of hypervisor and operating system: The operating system is adapted to a specific hypervisor, which is considered a special hardware architecture. In contrast to pure virtualization, running the operating system on raw hardware or on another hypervisor fails, because the para-virtualized operating system does not obey the native machine interface.

In summary, para-virtualization achieves better performance at the cost of higher engineering overhead and restricted binary compatibility. Composition of virtual machine environments is less flexible and less modular than with pure virtualization. Furthermore, the source code of the operating system must be modified by hand, which is cumbersome or even impossible in case of closed-source operating systems.

2.1.3 Virtualization by Binary Translation

Some hypervisors, such as VMware's Server, ESX Server, and Player [AA06], use binary translation to virtualize the operating system kernel's code at load and run time: Innocuous instructions are left unmodified, except for corrections of location-dependent code. Privileged instructions are replaced by *hypercalls*, downcalls to the hypervisor. An improved form of binary translation, *adaptive* binary translation, detects and translates virtualization-sensitive but non-privileged memory operations online by identifying data locations that are frequently involved in traps. In contrast to para-virtualization, the kernel is not subject to structural modifications.

Adaptive binary translation hides virtualization from the guest operating system kernel. Thus, adaptive binary translation can virtualize unmodified kernels and reduce virtualization overhead even on non-virtualizable architectures. However, the translation process itself is rather intricate, because virtual machine equivalence [PG74] requires the virtual machine to behave in a manner that is indistinguishable from a physical machine, set aside timing and resource availability.

2.1.4 Hardware-Assisted Virtualization

Recent extensions of the legacy processor architectures such as Intel's Vanderpool Technology [UNR$^+$05] or AMD's Pacifica [Adv06c] remedy non-virtualizability at the cost of more complex hardware. Introduction of a new processor mode permits virtualization of unmodified operating systems, but performance can decrease, caused by lack of optimization [AA06].

Progressive para-virtualization [Fra06] takes advantage of virtualizable processor architectures by selectively adding para-virtualization extensions to a previously unmodified guest operating system. These para-virtualization extensions can for example optimize I/O and MMU operation, avoid idle time and support device hotplugging.

2.1.5 Pre-Virtualization

Pre-Virtualization [LUC$^+$06] is a novel concept for virtualization that uses the soft-layering principle [Coo83] to improve hypervisor interoperability.

Applying the Soft-Layering Principle to Virtualization

The soft-layering principle [Coo83] was primarily developed to retain modularity in network protocol design: Performance of layered network protocols is often enhanced by increasing the flow of information between the layers through layer co-design. The disadvantage of co-design is that lower layers' interfaces are defined to match higher layers' requirements, such that lower layers are fixed and not exchangeable. However, to make lower layers exchangeable, the soft-layering principle requires that a layer offers by default a neutral interface to superior layers. In addition, the layer may provide superior layers with hints how to use it efficiently. Thus, soft-layering helps avoid co-design.

Para-virtualization is another instance of co-design, namely hypervisor / operating system co-design: Each para-virtualization hypervisor defines a specific hypercall API that differs from the native instruction set of the processor. Therefore, an operating system has to be ported to a hypervisor's API to make it compatible with the hypervisor.

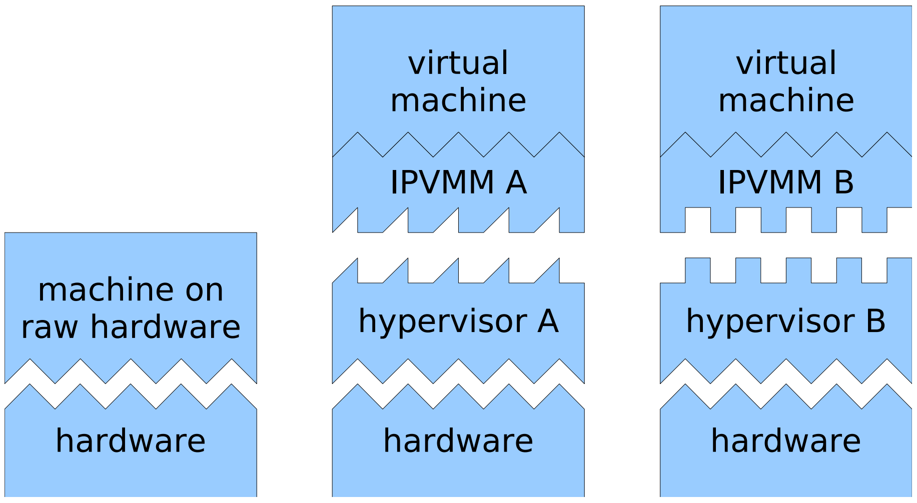

Figure 2.4: Several configurations for a pre-virtualized system

The pre-virtualization concept [LUC+06] applies soft layering to virtualization: The guest operating system kernel is prepared for virtualization, but it is still based on the native machine. At boot time of the virtual machine, a hypervisor takes advantage of the virtualization preparations and adapts the kernel to make it cooperate efficiently with the hypervisor. Thus, the same kernel binary image is able to run on raw hardware as well as on different hypervisors (see Figure 2.4).

The In-Place Virtual Machine Monitor

A hypervisor on a non-virtualizable processor architecture offers a hypercall interface that differs from the native machine interface. Therefore, an in-place virtual machine monitor (IPVMM) is introduced that represents the internal state of the virtual machine and mediates between guest kernel and hypervisor. For efficiency reasons, the IPVMM resides in the guest operating system kernel's protection domain. It maps sensitive instructions to hypercalls and forwards incoming interrupts and exceptions to the guest kernel.

Furthermore, the IPVMM is responsible for rewriting the guest kernel such that the sensitive instructions that the binary kernel image specifies are replaced by calls to emulation code in the IPVMM.

Advantages of Pre-Virtualization

Pre-virtualization is an improved form of para-virtualization where the virtualized interface agrees with the native machine interface. Compared to para-virtualization, it offers several advantages: First, the process of virtualization can be automated, squeezing down engineering effort near zero. Second, pre-virtualization leverages flexibility of the virtual machine environment, because it is hypervisor-neutral and preserves the native machine interface. Third, virtualization is more structured with pre-virtualization, given that all hypervisor-specific adaptations to a guest operating system kernel are encapsulated in the IPVMM.

Performance of a pre-virtualized machine depends on how efficient the IPVMM maps native machine interface to hypervisor interface. Comparison of pre- and para-virtualized operating system kernels [LUC^{+}06] have shown only slight differences in performance between both approaches.

Virtual Machine Environment Configurations

Pre-virtualization enables a single binary image of an operating system kernel to be used in a variety of models for hardware access:

- Raw: The kernel runs in non-virtualized mode on raw hardware (see Figure 2.4, left).
- Pass-through: The kernel runs on a hypervisor, but still has access to most physical devices. Only CPU and memory are virtualized by the hypervisor (see Figure 2.5, left).
- Pass-through with virtual PCI: The kernel runs on a hypervisor which offers real device access through a virtual PCI bus. This scheme enables partitioning resources between virtual machines.
- Hypervisor device drivers: Hardware is fully virtualized, so the kernel interacts with device drivers that are part of hypervisor (see Figure 2.5, middle). This scheme enables full system virtualization.
- Reused device drivers: Hardware is fully virtualized, but with this model, the guest kernel interacts with a device driver virtual machine (DD/VM) that has access to physical devices (see Figure 2.5, right). Reused device drivers enable fine-grained resource control and improve system dependability [LUSG04].

To summarize, the same guest kernel image can be used with different hypervisors. Furthermore, the hypervisor can create any number of virtual machines, each with memory and other resources assigned to it.

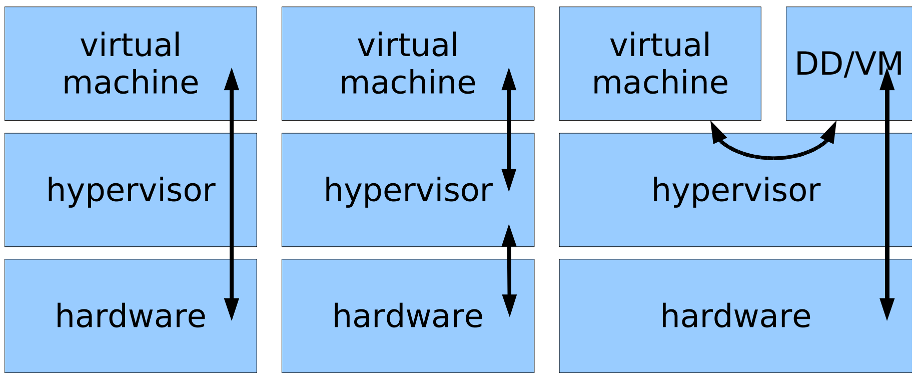

Figure 2.5: Several configurations for device access: pass-through (left), hypervisor device drivers (middle), reused device drivers (right)

2.2 Benchmarking Techniques

Generally, a benchmark is a standardized problem or test that serves as a basis for evaluation or comparison [mer06]. The notion *benchmark* originally designated a point of reference for topographic surveys, but nowadays the meaning has devolved to benchmarking programs and benchmarking results in computer science as well. Thus, benchmarks try to answer questions about how well a system performs a certain task [SKSZ99].

2.2.1 Requirements for Significant Benchmarks

Benchmarks serve a twofold goal [EWCS96, LIJ$^+$98]: First, they are used to identify performance problems and improve system design. Second, they provide a basis for comparison between systems.

Mogul [Mog99] gives a detailed analysis of requirements and properties of benchmarks. He states that significant benchmarks must fulfil three requirements:

- Benchmarks must be *repeatable*, such that other researchers can verify and perpetuate the tests.
- Benchmarks must be *comparable*, to permit valuation of suitability for a certain task.
- Benchmarks must be *relevant* to important applications, so that benchmark results predicts real-life behavior.

These requirements imply that the metrics used for benchmarking must have the following properties: First, a proper scientific procedure is indispensable for obtaining correct results. Second, the researchers must agree on a common set of metrics that are used. Third, the metric must be realistic and widely used, which means that it must be applicable both in the research domain and in the real-world domain.

However, Mogul argues that the most difficult part in benchmarking operating systems (as opposed to benchmarking computer architectures) is to develop reliable techniques to transfer insights from artificial, oversimplified test cases to complex, unpredictable environments. The "tension between realism and reproducibility" [Mog99] implicates that no benchmarking technique addresses both these requirements equally well.

2.2.2 Workloads and Metrics

Benchmarks are classified based on the type of workload that they impose on the system under test, ranging from homogeneous to heterogeneous, or based on the type of metrics that they use to characterize the system. From the wide variety of possible metrics, the following are most commonly used [EWCS96, TvS02]:

- *Latency* is the interval between stimulation and response.
- *Throughput* is the amount of data or events per time.
- *Dilation* is the extension in length.
- *Utilization* is the percentage of utilized computing time on a processor.

A metric represents only certain aspects of performance, some of which may be more important than others for a given combination of workload and user requirements. For example, throughput may not be an appropriate metric to characterize performance of an interactive system [EWCS96].

In benchmarking context, the notion *system under test* designates the combination of hardware and software that a benchmark evaluates.

Macro-Benchmarks

Macro-benchmarks [Ous90] measure overall performance of actual applications under loosely defined conditions. The results are realistic, but as they are tied to specific applications, the behavior of other applications can only vaguely be inferred [BS97].

Typical examples for macro-benchmark workload are kernel build, web server, or database operations.

Micro-Benchmarks

Micro-benchmarks measure the performance of individual primitive operations under well-defined conditions. The alternative notion narrow-spectrum benchmarks [SBSM89] emphasizes that the variation of the workload is small. Micro-benchmark measurements are quite exact but rather unrealistic [Mog99].

The lmbench suite [MS96] is a typical instance of an operating system micro-benchmark. Its design was motivated by the need for a simple, portable, and realistic benchmark suite that accurately measures a wide variety of individual operations.

However, when writing the accompanying paper, the authors already predicted that advances in computer architecture might obsolete the suite. In the course of about ten years, this prediction has probably come true. The timing mechanisms in lmbench are based on the *gettimeofday* system call, which is much more imprecise than performance monitoring hardware that is built in modern processors. Multi-stage instruction pipelines, out-of-order execution, and deeper memory hierarchies have invalidated lmbench's assumption of relatively simple and predictable hardware.

The lmbench suite has been criticized for lack of statistical rigor and inconsistent measurements [BS97], resulting in improved versions of lmbench [SM,Sta02] and in the development of hbench:OS [BS97].

Profiling

Profiling means benchmarking the behavior of a program, for example the frequency and duration of function calls or basic blocks. Examples for profilers are gprof [GKM82], ATOM [SE94] and DCPI [ABD$^+$97]. OProfile [MST$^+$05] is a profiling toolkit for the Linux kernel and applications that uses the CPU's performance monitoring counters and non-maskable interrupts to correlate hardware events with activity. The two major mechanisms to generate profiles are tracing and statistical profiling.

Tracing [GKM82] denotes the approach where the program under examination outputs a trace, a stream of events that pertains to the control flow. For that purpose, the program must be instrumented, that is, augmented by event registration code. Depending on the frequency of events and on the cost of trace generation, tracing can incur high overhead.

Statistical profiling [ABD$^+$97] trades profiling overhead with accuracy: Either in regular intervals (time-based sampling) or at certain events (tracing), samples of the processor state are stored. Each sample contains instruction pointer, address space identifier, and optionally additional information. A device driver usually aggregates samples and passes them to a user-space daemon, who in turn adds information derived from corresponding executable files, such as which functions pertain to the recorded instruction pointers. Thus, a statistical profiler outputs a statistical summary of the events observed.

The advantage of statistical profiling is that it works transparently to the code under examination, that is, the code does not need to be modified. Runtime overhead of statistical profiling is rather low. However, the measured distribution of executed code is only approximated.

Hardware Performance Counters

Many modern processors include low-level performance monitoring mechanisms that allow to keep track of hardware events such as clock ticks, retired instructions, cache misses, branches, and branch predictions. Intel's Pentium 4 and Core processors distinguish non-retirement and at-retirement events [Int06]:
Non-retirement events represent all events that occur during instruction execution, whereas at-retirement events are generated only for committed work. For example, a mispredicted branch contributes solely to non-retirement events. These processors provide three usage models of performance monitoring:

Event Counting The benchmarking software reads and writes the event counters as registers. This requires the hardware counters to be configured such that they automatically keep track of certain events. In regular intervals, for example at timer interrupts, the software must poll the registers.

The time stamp counter is a special event counter that measures time relative to processor startup. On modern Pentium 4 and Core processors, the time stamp counter increments at a constant rate, independent of processor clock speed changes. Having a width of 64 Bits, it will not wrap around for at least ten years after processor reset. The time stamp counter register is accessible via the *rdtsc* instruction. Unfortunately, the *rdtsc* instruction does not serialize the instruction stream, so it has to be prefixed with a serializing cpuid instruction if exact measurements are required. Thereby measurements affect performance, making vertical benchmarking at fine granularity impossible. Recent AMD processors provide a *rdtscp* instruction that serializes the processor pipeline and reads the time stamp counter atomically [Adv06b].

Non-precise Event-based Sampling The software defines a limit for the event counters in question. When the counters overflow, a performance monitoring interrupt is generated, so the interrupt service routine can record the return instruction pointer for later distribution analysis. The out-of-order execution of the Pentium 4 NetBurst and Core micro-architectures is problematic for precise accounting of costly instructions, resulting in wrong correlation of events and instructions.

Precise Event-based Sampling (PEBS) This usage model is similar to the previous, except that on overflow the processor state is stored in the debug store, a dedicated memory buffer.

2.2.3 Relating Micro- and Macro-Performance

Useful benchmarks must allow to understand and predict performance [Mog99]. Therefore, a benchmarking methodology needs a performance model that correlates micro- and macro-benchmarking results [SS96]. The model must be simple enough to enable efficient calculation, yet it must be powerful to explain a wide range of observations.

The Abstract Machine Model

Saavedra-Barrera's abstract machine model [SBSM89] assumes that a program consists of primitive operations whose individual execution times sum up to the total execution time of the program. More formally, the *system characterization vector* $\overrightarrow{v}$ represents the individual primitive operation's execution times, whereas the *primitive usage vector* $\overrightarrow{a}$ describes how often each primitive operation is executed by the program. Thus, the total execution time p of a program on an abstract machine is calculated as

$$p = \overrightarrow{v} \cdot \overrightarrow{a}.$$

A suite of micro-benchmarks, called machine characterizer, determines the system characterization vector. The primitive usage vector derives from an execution trace of the respective application. Curve fitting is never involved, so that the model directly explains execution time.

The model is simple, but quite limited for practical application. It ignores that execution on real processors is non-linear in time due to effects such as caching, pipelining, and interaction with hardware. In addition, the model assumes that a typical system characterization vector exists and can be accurately measured, which is not always the case [SS96]. Assuming a constant cache miss rate, memory hierarchies can be modeled by adding an explicit term.

Application-specific Benchmarking

As a refinement of the abstract machine model, Seltzer et. al. [SKSZ99] propose application-specific benchmarking, consisting of vector-based methodology and trace-based methodology.

The vector-based methodology is a generalization of Saavedra-Barrera's linear model to the affine performance metric

$$p = \overrightarrow{v} \cdot \overrightarrow{a} + gc(\overrightarrow{a}),$$

where $\overrightarrow{v}$ is the system characterization vector, $\overrightarrow{a}$ is the application performance vector, and $gc(\overrightarrow{a})$ is a possibly non-linear function to characterize asynchronous overhead. In their case, the asynchronous overhead was caused by garbage collection in Java virtual machines, which runs non-deterministically with respect to applications.

Trace-based methodology means dynamically adapting the application performance vector to the respective application. The base component is a cache simulator that provides a more detailed view of the application's operations by distinguishing cache hits and cache misses. From a mathematical point of view, the simulation procedure conducts piecewise linearization of a non-linear performance function.

In a subsequent work [ZS00], one of the authors achieved quite accurate performance predictions for Java virtual machines by tracing about hundred primitive operations that he considered decisive. However, it turned out that identifying significant primitive operations is nontrivial.

2.3 Virtual Machine Benchmarking

Previous work on virtual machine environments has typically used standard operating system benchmarking suites such as lmbench [MS96] to evaluate the impact of virtualization on performance. For example, the original paper on the Xen hypervisor [BDF+03] compares Xen's performance to several other hypervisors and to execution on raw hardware.

However, operating system benchmarking suites are not aware of virtualization and thus fail to answer virtualization-specific questions:

- Effect of server consolidation: How to describe the overall performance of a system of consolidated servers?
- Virtual machine interference: To what extent does the virtual machine environment impact performance of individual machines?
- Generalization of micro-benchmark results: How does application-level performance of a virtual machine relate to virtualization-level performance?

In the recent two years, several works have been published that address virtualization-specific benchmarking: Two papers [CGS06, MHS+] examine how to aggregate performance of multiple virtual machines into one single metric. One paper [AA06] compares the impact of hardware virtualization assistance on performance of virtual machines on the VMware Player hypervisor. The other papers [GGC,CG05,GCGV06,MST+05] describe two performance evaluation toolkits that are specific to the Xen hypervisor.

2.3.1 Performance of Consolidated Servers

Server Performance Characterization

Casazza et. al. [CGS06] suggest a high-level approach to characterize the performance of a system of consolidated servers. The authors observe that there are mainly two virtualization usage models in enterprises: *Server consolidation* typically results in a mixture of servers (web, email, database etc.). *Flexible provisioning* improves hardware utilization and accelerates installation of new servers by eliminating the procurement cycle. Both these virtualization usage models imply a mixture of different workloads, each having different requirements and metrics, with varying relative priorities.

Thus, the suggested methodology aggregates the overall performance of a physical server as the weighted sum of individual metrics that were normalized to non-virtualized performance. Iterative tuning of workload should identify bottlenecks in the virtual machine setup.

The aggregation metric applies to a given instance of consolidated servers. However, the methodology fails at two important benchmark requirements: Although performance bottlenecks may be identified experimentally, it does not help examine the causes for differences in performance among distinct setups. Also, the methodology cannot predict the performance of other workloads or virtual machine setups. Furthermore, the comparison of different virtualization environments is based on a single number, the aggregated metric, which certainly oversimplifies the impact of virtualization[1].

Nonetheless, the authors confirm a number of implementation challenges for virtual machine benchmarking, including the following:

- Virtual machine clock accuracy and precision differ from hardware.
- Hypervisors provide only basic performance monitoring capabilities.
- Virtualization introduces additional abstraction, additional overhead, and limits resources.
- Virtualization adds variation, particularly when resources are rare.

In the final section of the paper, the authors come up with the demand for industry-standard virtualization benchmarks. In addition to the necessary attributes of benchmarks mentioned in subsection 2.2.1, Casazza et. al. identify two attributes

[1] A recent whitepaper by VMware [VMwa], the employer of Casazza et. al., does not adhere to the server performance characterization methodology, a fact which has surprised its contender Xensource [Xen].

specific to virtualization: First, the benchmark should be agnostic in terms of platforms, hypervisors, and virtual machine setup to the extent possible. Second, the benchmark should apply to a wide range of systems, times, and components. The authors estimate that the most difficult decision will be to choose which workload and aggregation strategy matches user requirements.

VMMark

A recent paper by Makhija et. al. [MHS$^+$] discusses an alternative approach for benchmarking consolidated servers. To represent the diverse workload of virtualized servers on a physical host, the authors define a *tile* as "a collection of virtual machines executing a set of diverse workloads". In periodical intervals, the individual benchmark results are normalized to a reference system and then aggregated into a single metric. Each individual workload is throttled such that it executes at less than full utilization. Thus increasing the number of tiles running on a single physical machine allows to judge whether the system is overcommitted.

The authors combined five typical data center workloads and a standby server in their exemplary tile. Reusing pre-existing benchmarks reduced engineering effort and provided a reliable basis for the metrics. The virtualized setup required some adaptation, mainly to support aggregation of benchmark results and to restrict resource consumption.

It is questionable whether throttled workloads are realistic and interesting test cases for virtual machine environments. The paper leaves open the question how to decide on a well-defined reference score and on representative workload.

2.3.2 Comparison of Virtualization Techniques

Adams and Agesen [AA06] compare techniques for x86 virtualization, namely software virtualization via adaptive binary translation versus hardware-assisted virtualization.

In a qualitative comparison of virtualization overhead, the authors find that neither technique is superior to the other. Adaptive binary translation tends to win in trap elimination, emulation speed, and callout avoidance, whereas hardware-assisted virtualization wins in code density, precise exceptions, and system call performance.

In a series of experiments, the authors investigate virtual machine performance under several workloads. Under benign workloads, which consist mostly of user-level computations, execution time is close to native speed with either virtualization technique. Under more operating-system intensive workloads, such as an HTTP server, the PassMark desktop-oriented benchmark, and large compile jobs, performance degrades to about 20% to 70% of native performance. The *fork-wait* micro-benchmark stresses virtualization-sensitive operations, carrying performance degradation to extremes. In order to examine performance differences more in-depth, the authors wrote a series of virtualization nano-benchmarks, each measuring performance of a single sensitive operation from within an artificial operating system kernel.

The results from the publication give valuable insights in virtual machine performance. However, the approach does not directly correlate nano-performance with micro- and macro-performance. The proposed nano-benchmarking technique is incompatible with para-virtualization's structural modifications, preventing performance evaluation of the popular Xen hypervisor. Instead, nano-benchmarks run in the non-realistic environment of a purpose-built operating system.

2.3.3 Xen Profiling Suites

XenMon

XenMon [GGC] is a toolkit for QoS monitoring and performance profiling for Xen-based VM environments, consisting of three components:

- The *xentrace* component resides in the hypervisor and generates events at arbitrary control points.
- The *xenbaked* component resides in the privileged domain Dom-0. It catches and processes the event stream.
- The *xenmon* frontend displays and logs *xenbaked*'s output.

XenMon supports several metrics, such as different time measures (CPU usage, blocked time, waiting time), execution counts, and I/O operation counts. Xen 2 implements I/O operations by means of page flipping in order to avoid copying data between virtual machines: The client virtual machine exchanges a page containing I/O data with an unused page from the driver virtual machine, an operation which is simple to instrument. The virtual machine scheduler was modified to keep track of lapsed time and count virtual machine executions.

When using XenMon to study web server performance [CG05], the authors found that small file sizes result in notable CPU processing overhead, because every HTTP request/response pair results in the transfer of at least 5 TCP/IP packets, each requiring a page flip. XenMon has also been used for measuring disk I/O. Page flipping caused higher CPU overhead than expected: for each raw disk block of 512 Bytes, a 4 KB memory page was flipped. In a sensitivity study on Apache web server performance [CG05], the impact of CPU allocation on web server performance was analyzed. The study revealed that increased CPU allocation to the driver domain sometimes resulted in frequent protection domain switches and led to significantly decreasing performance.

In a recent work [GCGV06], a new virtual machine scheduler was built that accepts feedback from XenMon. A mechanism named ShareGuard, similar to Lazy Receiver Processing [DB96] and Resource Containers [BDM99], attributes a driver virtual machine's CPU time to the virtual machine that initiated the I/O operation.

Although XenMon helps understand performance of different virtual machine configurations, it is specific to the Xen hypervisor. Comparing results under different hypervisors would require porting the XenMon infrastructure to each hypervisor under examination. Different abstractions, such as how I/O operation are realized, might render the results incomparable between different hypervisors.

Xenoprof

The Xenoprof project [MST$^+$05] adds support for OProfile (see Subsection 2.2.2) to the Xen hypervisor. Virtualizing the hardware performance counters would require a different privilege setup and incur a high overhead, so the hardware performance counters are managed by Xen on behalf of the guest kernels. Xen-specific drivers residing in the guest kernels receive notifications about performance monitoring events by means of virtual interrupts, and they access the data via shared memory.

Using Xenoprof the authors discovered that, in the original Xen performance measurements [BDF$^+$03], not the hypervisor but the network interface card was the component which limited throughput. Unfortunately, given that hardware performance counter virtualization would be costly, Xenoprof is tied to paravirtualized XenoLinux. Using OProfile with other hypervisors would require porting the drivers and modifying the user-level tools and the hypervisor.

2.3.4 Shortcomings of Existing Virtual Machine Benchmarks

As the preceding discussion has shown, work on virtual machine benchmarking treats three areas of interest. The VConsolidate and VMMark projects address high-level performance characterization of colocated virtual machines. Another work conducts a low-level comparison of hardware and software techniques for virtualization under VMware. The Xenoprof and XenMon projects investigate virtualization performance specific to the Xen hypervisor.

However, research on virtual machine performance has neither addressed how to characterize different hypervisors at a low level of abstraction nor how to relate performance of different granularity. Both accurate hypervisor characterization and decompositional analysis of virtualization's performance impact are necessary to understand performance. Users of virtual machine environments need an impartial basis for their purchasing decisions, developers wish to avoid performance bottlenecks in their virtualization solutions, and researchers want to learn how to build efficient virtual machine environments. In contrast to related work on virtual machine benchmarking, this thesis focuses on low-level cross-hypervisor comparison and prediction of virtual machine performance.

Chapter 3

Design

First, this chapter analyzes the impact of virtualization on performance. Then, it states the requirements for a virtual machine benchmarking suite and presents a three-stage methodology for virtual machine benchmarking. Subsequently, it describes the design of VMbench, a virtual machine benchmarking suite, in relation to the three stages.

3.1 The Impact of Virtualization on Performance

The starting point for a performance evaluation is to find out which aspects of performance are of interest. Therefore, a virtual machine benchmarking suite that should be able to compare different virtualization techniques must model peculiarities of particular virtualization techniques and hypervisors in a generic manner. Current system virtualization paradigms impact several aspects of performance:

- Virtualization-sensitive instructions: As sensitive instructions alter external state of the virtual machine, they must not execute within the virtual machine (see Section 2.1). Thus, either the virtual machine itself, the hypervisor or special virtualization hardware has to emulate the behavior of sensitive instructions.

- Virtual processor mode and context switch, interrupt delivery, and access across user-kernel boundary: Depending on which virtualization paradigm is used, the hypervisor must intercept voluntary or non-voluntary privilege switches of the virtual processor. For example, with hardware-assisted virtualization, the hypervisor must intercept exceptions, but it does not need to intercept system calls. In contrast, with virtualization using binary translation, the hypervisor must intercept system calls too [AA06]. Also, most virtualization paradigms require that the hypervisor intercepts hardware interrupts. If the hypervisor switches between guest kernel and user address spaces by means of paging [LUC$^+$06], kernel access to user memory is delayed, too, because the user address space must be emulated for the kernel.

- Interference of virtual machines sharing one physical host: Concurrent virtual machines have restricted access to physical resources. Performance of competing virtual machines decreases if resources are insufficiently available. A related problem are sequencing issues: Hypervisors sometimes lack knowledge about optimal scheduling of virtual machines. Thus, they often take suboptimal decisions about the order in which virtual machines are granted resource access.

Considering the diversity of virtualization techniques, there may exist other aspects of performance that are susceptible to virtualization. However, most contemporary system virtualization techniques encounter the aforementioned aspects.

3.2 Requirements for a Virtual Machine Benchmarking Suite

In order to yield meaningful results as defined by Mogul [Mog99] and detailed by Casazza et. al. [CGS06], a virtual machine benchmarking suite must meet the following requirements:

- Comparability: The benchmarking suite must allow to compare different hypervisors and virtual machine environment setups. Thus, the benchmarking suite should run on any hypervisor without requiring major adaptations. Moreover, the benchmarking suite should be easily portable to other architectures and operating systems.

- Generality: The benchmarking suite must predict performance of different types of applications. Therefore, the suite must incorporate different benchmarking applications, each stressing another aspect of performance.

3.3 Methodology

VMbench uses a three-stage approach to characterize performance of a virtual machine environment. The stages build upon each other, increasingly tolerating complexity and non-determinism of the environment (see Figure 3.1).

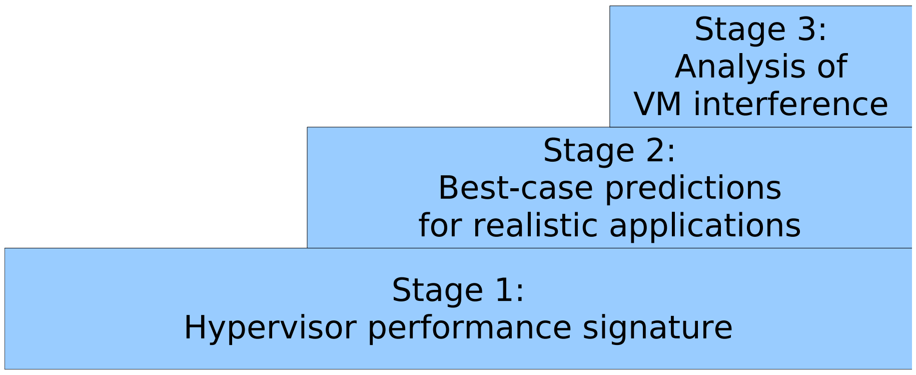

Figure 3.1: Stages of analysis with VMbench

1. Hypervisor performance signature: In the first stage, micro- and nano-benchmarks determine the hypervisor performance signature, that is, the best-case performance of a virtual machine's primitive operations for a given combination of hardware, hypervisor, operating system and workload. Therefore, a single virtual machine exercises well-defined operations, such that the performance of virtualization-specific functional primitives can be accurately measured. To determine the best-case performance, VMbench minimizes side effects and interprets the results optimistically.

2. Virtual machine performance: The second stage combines the outcome of the first stage using a linear model to predict best-case results for realistic applications.

3. Virtual machine interference: The third stage examines how the prediction from the second stage varies under non-optimal conditions caused by concurrent virtual machines.

VMbench pursues a latency-oriented approach. Although some application areas for virtual machines exist where data throughput is more important than latency [CGS06], several arguments justify the focus on latency: First, latency of primitive operations allows to apply a linear performance model. Second, according to Endo et. al. [EWCS96], "throughput measures provide an indirect rather than direct measure of latency" by delivering only end-to-end information and ignoring event handling latency. Third, VMbench considers throughput reduction caused by virtualization to be a consequence of dilated primitive operations.

3.4 Issues of Virtual Machine Benchmarking

Benchmarking in a virtual machine environment faces a number of issues. These issues are related to the way a system is virtualized, to the additional layer that virtualization introduces, and to unforeseeable effects caused by the complexity of the virtualized system. Each of the following subsections addresses a benchmarking-sensitive aspect of virtualization and discusses how to solve it.

3.4.1 Different Virtualization Techniques and Hypervisors

Within only a couple of years, the resurgence of system virtualization has stimulated the development of diverse virtualization paradigms and hypervisors. Even when limiting the comparison to IA32, the most popular processor architecture for server computers today, competing hypervisors are based upon hardware-assisted virtualization, para-virtualization, virtualization via binary translation, or combinations thereof. Some hypervisors support hardware partitioning, some support full system virtualization (see Section 2.1).

The requirement for comparability (see Section 3.2) postulates that a virtual machine benchmarking suite should be able to compare performance of virtual machines on hypervisors adhering to different virtualization paradigms. In order to yield comparable results, the benchmarking suite must measure performance with respect to a common interface. Moreover, the suite must not be biased towards one hypervisor.

Hardware-assisted virtualization and virtualization by binary translation apply to the native machine interface, such that virtual machines are directly comparable at a machine's lowest abstraction level. Para-virtualization's hypervisor-specific adaptation prevent instruction-level comparison of virtual machines. However, pre-virtualization, which leaves the native machine interface intact (see Subsection 2.1.5), is capable to substitute para-virtualization [LUC$^+$06]. In addition, it is compatible with hardware-assisted virtualization and with virtualization by binary translation, in the sense that these techniques can virtualize a pre-virtualized operating system kernel, too. Therefore, pre-virtualization enables instruction-level comparison of different hypervisors, provided that the respective IPVMMs perform comparably well.

For comparing different hypervisors at higher levels of abstraction, the benchmarking process must establish a unified interface to wrap hypervisor-specific functionality. For example, the Xenoprof infrastructure, which supports statistical profiling on the Xen hypervisor (see Subsection 2.3.3), could be ported to each hypervisor under examination. Given that each hypervisor has its own specialties, porting requires a certain engineering effort, and the resulting profiler might be biased.

3.4.2 Implications of the Layered System Structure

Benchmarks are initiated by one virtual machine, which I will call *primary virtual machine*. In order to yield meaningful results, the primary virtual machine must be aware of the hypervisor and of all other virtual machines in the system (*secondary virtual machines*). However, virtual machines are strictly isolated. The consequences of isolation are virtualization transparency, virtual machine opacity, and emergent misbehavior.

Virtualization Transparency

With the exception of para-virtualization, virtualization is transparent, that is, the virtual machines are not aware of being virtualized. In any case, virtual machines have no means to retrieve information from the hypervisor, and they do not know anything about the other virtual machines with whom they share a physical host.

An example for effects of virtualization transparency is the address space layout in traditional L4-based virtualization: As the guest operating system kernel runs in user mode, user address spaces are not mapped into the kernel address space. Instead, temporary mappings are constructed whenever the guest kernel faults on user memory. These shadow page faults, which are transparent to the guest kernel, cause notable degradation of performance.

The solution to circumvent virtualization transparency is straightforward: Virtualization is made opaque by extending the virtualizing software such that it exposes information about itself and about all virtual machines to the primary virtual machine. For example, the hypervisor or the IPVMM must report the number of transparent guest kernel page faults. The primary virtual machine is augmented with facilities to gather information from all components of the virtual machine environment.

VMbench defines a common approach for benchmarking applications to identify the IPVMM (if one is present), configuration of all virtual machines on the physical host, and name and version of the hypervisor. To enable coordinating of measurements, each component that actively participates in benchmarking offers an interface to control benchmarking and access measurement results. Wherever possible, performance monitoring hardware is not virtualized, such that data from the various components is directly comparable and additional measurement overhead is avoided. However, if an instruction requires supervisor privilege, such as IA32's *wrmsr* instruction, a virtualized operating system kernel will cause a fault, and the hypervisor must emulate the instruction. In order to make shadow page faults and other transparent events visible to the primary virtual machine, all hypervisors under examination must be modified to expose an equivalent benchmarking interface.

Given that benchmarking takes place in an artificial environment rather than in a production environment, relaxing virtual machine isolation does not impact system security. Influence of the benchmarking extensions on performance is a more serious problem: The benchmarking suite must be designed such that performance in a benchmarking-aware environment does not deviate from performance in a production environment. As exact measurements with IA32's *rdtsc* instruction are relatively expensive (see Subsection 2.2.2), simultaneous benchmarking at different levels of the virtual machine environment is impossible.

Opacity of Virtual Machines

Some benchmarking techniques, such as statistical profiling (see Subsection 2.2.2), require accessing virtual machine state that resides within other protection domains. For these investigations, cutting down virtualization transparency is insufficient. The benchmarking suite needs a means to retrieve and interpret internal virtual machine state. However, facilities to access internal virtual machine state differ among virtual machine environments.

Moreover, in some virtual machine environments, the resource monitor, which emulates or multiplexes hardware devices, runs in user mode. The L4Ka virtualization project [LUC$^+$06] is an example for fine-grained protection domains. The modularity created thereby enhances safety and configurability, but it makes attributing processor state to virtual machines more difficult: The hypervisor does not associate virtual machine semantics with address spaces, and the resource monitor cannot access processor state of threads residing in other address spaces.

In contrast to a distributed system, a common clock exists within every virtual machine environment, namely the non-virtualized time stamp counter which is built into every modern processor. In order not to virtualize the time stamp counter, the hypervisor must simply clear a processor flag during initialization. Given that the time stamp counters of the individual processors in a multiprocessor machine may diverge, each component that accesses the time stamp counter must be physically multiprocessor-aware.

To support statistical profiling, VMbench includes a hypervisor-neutral library for logging events and associated information about protection domains. Hypervisor-specific glue code binds the library to the hypervisor's facilities for interrupt handling and debug output.

Non-Normality and Emergent Misbehavior

Conventional micro-benchmarks usually expect that their measurements are normally distributed, and that measurements of different systems are distributed similarly. The underlying assumption is that measurement errors result from the addition of many small and independent effects, in which case the central limit theorem applies [Was04]. Given that the affine transformation of normally distributed, independent random variables is itself normally distributed, normally distributed measurements can be composed linearly.

A normality assumption can be tested, for example using the Kolmogorov-Smirnov test, or in case of small populations using the Lilliefors test. In case a normality assumption cannot be justified, there are unnoticed effects that need more precise explanation.

On the one hand, performance of a virtual machine environment heavily depends on effects of locality (that is, caching and congestion), and on random timing effects in the various layers. Therefore, the normality assumption is rarely justified. On the other hand, even if the primitive operations are not normally distributed, a linear model might apply.

The problem of non-predictability gets even worse if complexity of the virtualized system causes emergent misbehavior, that is, unforeseeable effects that result from the interaction of several components [Mog06]. To understand the adverse impact of non-normality and emergent misbehavior, consider the following three examples from different levels of the virtual machine environment:

- To avoid switching the protection mode, a virtual machine often buffers page table modifications and flushes them at once to the hypervisor.
- The Pentium IV NetBurst microarchitecture is heavily pipelined and supports out-of-order execution [Int06]. On the one hand, ignoring the pipeline for instruction-level measurements yields imprecise results. On the other hand, controlling the pipeline by means of serialization leads to worst-case measurements.
- Scalability of multiprocessor virtual machines is susceptible to suboptimal scheduling decisions by the hypervisor. If a virtual machine holding a spin lock is preempted, concurrent virtual processors that contend for the lock waste time for reasons that are invisible to the guest operating system. Uhlig et. al. indicate that avoiding lock-holder preemption can improve virtual machine performance by up to 28% [ULSD04].

Application-specific benchmarking handles these effects by refining its system model based on knowledge about the underlying architecture. Using application-specific benchmarking in virtual machine context is hindered by underlying virtualized architecture behaving different for each hypervisor. The benchmarking suite must not make assumptions about hypervisor implementation.

For these reasons, VMbench sets aside automatic application-specific benchmarking and contends with analyzing the distribution of measurements, such that application-specific benchmarking can be applied manually for each hypervisor.

3.4.3 Performance Accounting and Aggregation

In the case of virtualized devices, virtual machines can offload workload to other virtual machines, for example to ones running a device driver operating system [LUSG04]. Resource containers as proposed by Banga et.al. [BDM99] might be helpful to correctly account execution time spent on behalf of other virtual machines. However, to keep track of workload offloading, a convention among all hypervisors and virtual machines would be necessary, so the initial version of VMbench does not include a concept for resource containers.

The question how to aggregate the performance of multiple virtual machines has recently been treated by the VMMark and VConsolidate projects [MHS$^+$, CGS06]. Although VMbench was designed with a focus on low-level performance aspects, it is compatible with such high-level benchmarks: Its output can be normalized to non-virtualized performance and included in an aggregate metric.

3.5 Hypervisor Performance Signature

VMbench is motivated by an application-centric point of view: Application software and operating system kernel run in a virtual machine, so a virtual machine performance evaluation can be similar to a hardware performance evaluation. In the style of the abstract machine model (see Subsection 2.2.3) the virtual machine characterizer outputs a performance signature of virtualization-specific primitive operations.

However, a virtual machine is subject to more side effects than a physical machine. For example, the hypervisor or other virtual machines can steal time from it, which means that the virtual processor neither sleeps nor runs, because the hypervisor performs housekeeping work, or another virtual machine occupies the physical processor. If the guest operating system kernel executes in user mode, a virtual processor mode switch causes additional overhead by physical context switching. In order to yield a best-case performance signature, stolen time must be minimized.

Figure 3.2 depicts the kinds of activity that occur in a system virtual machine:

- An application executes innocuous instructions.
- A fault, system call, or interrupt causes a transition to the guest operating system kernel. The transition proceeds either directly or via indirection through the hypervisor.
- The guest kernel executes a mix of innocuous and sensitive instructions. Sensitive instructions are emulated by the IPVMM or a comparable entity

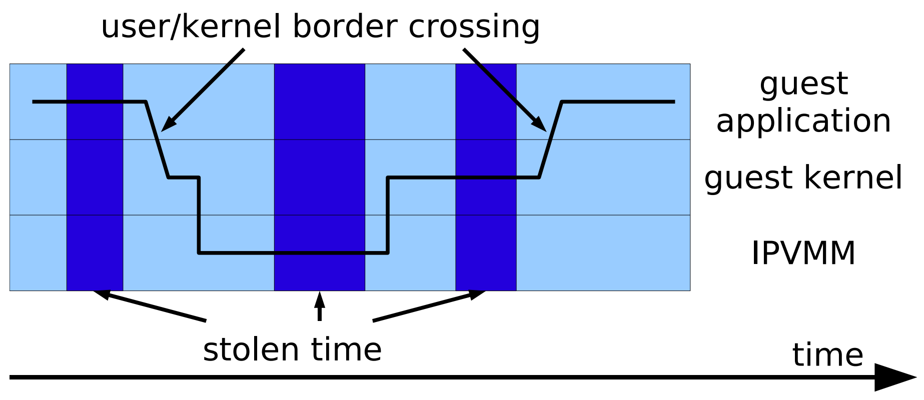

Figure 3.2: Activity in a virtual machine

and may cause a hypercall that transfers control to the hypervisor. The emulation executes transparently with respect to the guest operating system.

- The guest kernel reactivates an application, possibly after a guest context switch. Again, the privilege switch proceeds either directly or via the hypervisor.

- At any point in time, the hypervisor or another virtual machine can steal time from the virtual machine.

VMbench assumes that innocuous instructions executed by an application or by the guest kernel have no time dilation effect.

3.5.1 Choosing an Adequate Level of Abstraction

The level of abstraction for hypervisor benchmarks must be chosen carefully: On the one hand, only low-level hypervisor benchmarks allow precise characterization of the hypervisor. On the other hand, portability between hypervisors requires that the micro-benchmarks must no be affected by para-virtualization-related structural modifications of the guest operating system kernel. Furthermore, the level of abstraction should define an interface that is narrow enough to avoid collecting huge amounts of data. Representative measurements should deviate little from their respective mean value. If the measurements are not normally distributed, the model should probably be refined [SKSZ99].

An examination of the diverse levels in a virtual machine environment indicates which primitive operations are suitable for hypervisor benchmarks.

- For hypervisor comparison, processor microarchitecture events are very low-level. Measuring microarchitecture events requires each hypervisor to provide a way to configure and access measurements in a similar way.

- System calls are too high-level to allow conclusions about how the virtualization will react on them. For example, RAM-resident files and files on hard disk cannot be distinguished at the system call level, such that overhead of accessing a file on hard disk cannot be predicted.

- Hypercalls are hardly comparable, because para-virtualization results in incompatible hypercall interfaces.

- Operating system kernel functions are not suited for performance prediction with a linear model, because they are iterative, nested, and usually have non-constant execution time. However, kernel function profiling can be helpful for analyzing which functions are susceptible to virtualization and therefore targets for optimization through para-virtualization. Principal component analysis [The07] is an adequate statistical method to identify these functions: It simplifies a multidimensional data set using an orthogonal linear transformation such that the resulting subspace has largest variance.

- Basic blocks are sequences of instructions, where each instruction always executes directly before the succeeding instruction. In contrast to functions, basic blocks have exactly one entry and one exit, and do not contain any jumps. Therefore, the performance of basic blocks is easier to describe than the performance of functions. However, basic blocks are not a syntactical construct of high-level programming languages. In case a basic blocks contains virtualization-sensitive instructions or is interrupted by concurrent activity, its execution time is not constant.

All of the levels mentioned above represent only effects that pertain to the synchronous stream of execution. They do not contain effects of transparent or asynchronous execution such as address space switches and interrupts.

As outlined in subsection 3.4.1, I suggest comparing hypervisors on a common instruction-level interface. First, with pre-virtualization (see Subsection 2.1.5), performance of raw hardware and different virtualization techniques is directly comparable. Measurements at the neutral platform interface are orthogonal to virtualization, which is carried out by the IPVMM below this interface. Second, pre-virtualization allows automatic instrumentation of the guest operating system kernel by modifying the virtualization preparation phase during kernel build. Third, the instruction level is the most natural interface to record transparent or asynchronous events.

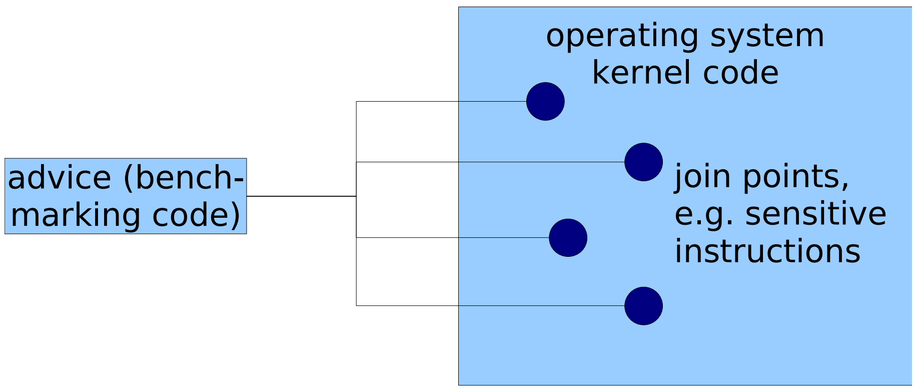
operating system kernel code
advice (benchmarking code)
join points, e.g. sensitive instructions

Figure 3.3: Instrumentation according to aspect-oriented programming

3.5.2 Nano-Benchmarks

Instrumentation for nano-benchmarks at the instruction level is modelled conveniently using the aspect-oriented programming paradigm [MSSP02]: The crosscutting-concern *benchmarking* inserts instrumentation code, which is specified as an advice, at certain code locations that are indicated by join points such as instances of sensitive instructions or function calls (see Figure 3.3).

VMbench allows to instrument primitive operations of different granularity: functions, basic blocks, and sensitive instructions. When recording the execution of sensitive instructions, the instructions are identified either by their code address or by their instruction type. Alternatively, VMbench supports manual instrumentation, such as performance measurements of system calls or at specific code locations within the IPVMM.

Each class of primitive operations can be combined with a metric. In this context, the notion *metric* designates the benchmarking code that is executed before and after a primitive operation. Table 3.1 presents an overview of VMbench's nano-benchmark metrics with respect to their information content and their overhead.

The *Count* Metric

Among the three metrics, the *count* metric has the lowest memory and runtime overhead. Having minimal side effects, it is best suited for extracting a usage vector of the primitive operations. However, it does not give any information about the sequence or execution latency of primitive operations.

Table 3.1: VMbench's nano-benchmark metrics on the IA32 architecture

Metric	**Measured data**	**Memory overhead for n measurements and m primitives**	**Runtime overhead instr./ser./ D-cache misses**
count	counter	1×m words	1 / 0 / up to 1
mean latency	counter cumulated time	3×m words	34 / 2 / up to 6
latency samples	counter time event type	1+2×n words	42 / 2 / up to 6

The *Mean Latency* and Related Metrics

The *mean latency* metric cumulates execution latency and execution count of primitive operations, the information necessary to calculate the arithmetic mean of latencies. It serves well to accumulate the total execution time of primitive operations, and to characterize primitive operations that are normally distributed. Non-normally distributed operations are not characterized well by this metric. This metric has only modest space requirements, but exact latency measurement on the IA32 architecture requires two serializations of the instruction stream for each primitive operation that is executed. Moreover, serialization clobbers the four general-purpose registers, such that the metric code must temporarily save them on the stack. The double-word (64 bit) wide accumulator can hold time intervals up to several years. *Minimum latency* or *maximum latency* metrics can be implemented similarly to *mean latency*.

The *Latency Samples* Metric

Latency samples are the most fine grained metric. Memory overhead is proportional to the maximum number of samples that can be recorded. Given that dynamic memory allocation causes bookkeeping overhead and non-predictable timing of memory accesses, samples are saved to statically allocated memory, such that only a fixed number of samples can be saved. Runtime overhead of *latency samples* is only a little higher than with *mean latency*, considering that measuring samples of latency requires two serializations per measurement as well. In order to save space, time intervals are saved as 32 bit wide words, such that latencies up to one second can be measured, which has been found to suffice even for delayed instructions, basic blocks, and functions.

3.5.3 Micro-Benchmarks

The instruction- and function-level benchmarks discussed in the previous section only represent the activity in the guest operating system kernel that is synchronous to the virtual processor. As discussed in section 3.1, most virtualization techniques also reduce performance of asynchronous events or events related to processor mode switching. Examples for these types of events are page faults, exceptions, system calls, and interrupts. With virtualization, switching the processor mode or the current address space can take considerably more time.

VMbench includes micro-benchmarks to characterize these events in a manner similar to the lmbench suite [MS96].

3.5.4 Storing and Exporting Recorded Data

To yield realistic results, a benchmark suite must collect data with lowest overhead possible. Thus, not only the metric code, but also the storage of recorded data must be optimized to be minimally intrusive.

Memory Allocation

Nano-benchmarks, which are transparent to the surrounding code, must neither modify processor registers nor the stack. Therefore, all nano-benchmarking data is stored in dedicated memory. In case of latency measurements, the code that is inserted before the primitive operation being measured fetches the respective memory into the processor cache and saves the negated current time to it. The code that is inserted after the primitive operation adds the current time to the memory. Thus, the benchmarking code calculates the time difference between start and end of the primitive operation.

Coordinating Measurements

A user initiates a benchmark run by executing a user-mode application. This application consecutively identifies the system it runs on, starts measuring in all benchmarking-aware components of the virtual machine environment, imposes some workload on the system, finishes measuring, gathers the results, and outputs them in human-readable form. Controlling benchmarks from user-level provides end-to-end measurement with respect to the user and simplifies saving data to files. The interesting topic for virtual machine benchmark coordination is how to enforce benchmark control and data aggregation across diverse system layers and configurations.

The lower layers of the virtual machine environment, more precisely guest operating system kernel, IPVMM, and hypervisor, must provide means to read and reset their counters, either via an interface akin to function calls or via shared memory. Shared memory is the preferred method to transfer large chunks of contiguous data. Call semantics are suited for short notifications, or in case shared memory cannot be established without major modifications.

- Benchmarking applications access data from the operating system kernel via shared memory.
- The IPVMM offers *wedge calls*, which are basically system calls that are handled by the IPVMM itself. All IPVMMs under examination must implement a common set of wedge calls.
- Communication with hypervisors is implemented via their specific hypercall interface.

Non-Atomic Measurements

Both nano- and micro-measurements are non-atomic: Other activities in the virtual machine system, such as stolen time or a world switch to another virtual machine, can disjoin start and end of a measurement. Disabling stolen time during measurements by means of locking would cause results to be non-realistic. Therefore, the mean latency metric should be interpreted with care. Latency samples should be checked for excessive outliers. In most cases, interleaved measurements are rare, such that the median is an adequate measure for the central tendency.

VMbench tries to enforce virtually synchronous start and stop of measurements by pre-paging memory mappings where possible. Hypercalls and wedge calls are synchronous with respect to the virtual machine anyway.

Measurements on Multiprocessor Machines

In a multiprocessor environment, inter-processor synchronization to arbitrate storage for measurements would disrupt realistic time behavior. With all virtual machine environments I am aware of, sensitive instructions are emulated on the same processor where they were issued. Thus it suffices to allocate distinct storage for each processor. VMbench does not yet handle threads migrating between virtual processors.

3.6 Virtual Machine Performance Under Optimal Conditions

Similarity between the abstract machine model (see Subsection 2.2.3) and the virtual machine model suggests to apply a linear performance model for predicting virtual machine performance based on the hypervisor performance signature. The abstract machine model maps well to execution of the same sequence of primitive operations on different types of virtual machines. However, the second analysis stage which determines the baseline cost of virtualization must avoid virtual machine interference. Therefore, virtual machine performance prediction starts with the best-case performance of a virtualized system.

3.6.1 Applying the Linear Model

In order to contend with non-normally distributed latencies of primitive operations, the best-case system characterization vector is defined to be the median execution time of primitive operations under optimal conditions. If the execution time is skewed but outliers are infrequent, the mean execution time may be more realistic. In most observed cases, the 5%-trimmed median is sufficiently realistic to represent the central tendency. The decision which measure for central tendency is most representative must be made on a case-by-case basis. For example, the user should verify that the mean latency coincides with the calculated mean of the latency samples.

In analogy to vertical profiling as described by Hauswirth [HSDH04], *vertical benchmarking* means that benchmarks at different levels of abstraction run simultaneously. The high cost of instruction-level benchmarking on the IA32 architecture, which is mainly caused by instruction stream serializations, makes vertical benchmarking impossible. Instead, VMbench runs the same workload on differently instrumented operating system kernels to determine minimum run times at each level.

3.6.2 Selecting Relevant Workload

The requirement for generality (see Section 3.2) demands a virtual machine benchmarking suite to incorporate different types of workload. Thus, VMbench entails a selection of diverse workloads. In accordance with related work [AA06,MHS^{+}], VMbench distinguishes three types of application based on how they are affected by virtualization:

- Computing-intensive applications run mostly in user mode. Thus, system virtualization affects user mode execution mainly through interference of concurrent virtual machines. Thus, computing-intensive applications are helpful for characterizing performance of concurrent virtual machines.
- Operating system intensive applications frequently exercise the kernel, for example by accessing RAM-resident filesystems or by manipulating tasks or threads. However, I/O operations that block the virtual machine are rare, so dilation of virtualized operations is the most prevailing aspect of performance.
- I/O intensive applications use the operating system mostly for accessing external hardware, such as disc or network. The virtual machine is often idle while waiting for I/O operations to complete. Therefore, data throughput is more important than dilation. In some cases, virtualization-specific bottlenecks limit data throughput.

Several smaller programs represent a mix of computing-intensive and operating system intensive workload. To evaluate OS-intensive workload, VMbench runs the Linux kernel build benchmark. To evaluate I/O-intensive workload, the netperf and httperf benchmarks are suitable, because they are well maintained and easily configurable. The netperf and httperf benchmarks are not yet integrated with VMbench.

3.7 Virtual Machine Interference

The previous section described how to establish a lower bound for the best-case performance of virtual machines. The best-case performance analysis uses a linear model based on the primitive operations that a virtual machine executes. As a consequence, it ignores all independent activity by other virtual machines or by the hypervisor. For example, it cannot identify transparent activity which increases stolen time (see Section 3.5) or delays of asynchronous events such as interrupt delivery.

In addition to nano- and micro-benchmarks, the benchmarking suite needs a benchmark that gives a global view of the physical machine. Two types of benchmarks are suitable to retrieve a global view: statistical profiling or idle-loop profiling.

3.7.1 Statistical Profiling

Statistical profiling (see Subsection 2.2.2) requires to access the processor state either in regular intervals or at certain events such as performance monitoring interrupts. A virtual machine benchmarking suite must do this with low overhead in a portable manner.

A meaningful statistical profile attributes consumed resources (e.g. processor time) to the entity consuming it (e.g. the thread). However, in a virtual machine environment, this information is distributed among several components: Statistical profiling must track low-level hardware events, such as hardware performance counters or timer interrupts. Most of these tasks require supervisor privileges, such that the hypervisor has to carry them out. In case of fine-grained privilege separation, as with the L4Ka virtualization project [LUC$^+$06], the hypervisor associates no virtual machine semantics with the profiling information. Only the virtual machine monitor, who may not have supervisor privileges, knows which virtual machine an address space is assigned to. Unless the virtual machine monitor imposes extensive restrictions on its virtual machines, it cannot interpret profiled state. Thus, the virtual machine itself must find out which process the profiled state describes.

VMbench supports statistical profiling in collaboration with the hypervisor. However, hypervisor-specific bindings to performance monitoring interrupts are necessary to make this work.

3.7.2 Idle-Loop Profiling

Idle-loop profiling has been suggested by Endo et. al. [EWCS96] to determine event handling latency. The technique is even less intrusive than statistical profiling, requiring only a user-mode process at low priority to time the execution of an idle loop. Dilation of the loop's execution time indicates activity within the virtual machine or, if the idle loop is the only process running in the virtual machine, time stolen by the hypervisor or by other virtual machines.

The drawback of idle-loop profiling as opposed to statistical profiling is that it cannot determine what caused the dilation. The user must either guess the causative event or impose synthetic workload.

To implement idle-loop profiling, neither hypervisor nor operating system needs to be modified. Therefore, this technique is well suited to identify stolen time in a virtual machine environment.

Chapter 4

Implementation

This chapter describes the implementation of VMbench for the Linux kernel running on the IA32 architecture.

4.1 The Structure of VMbench

The overall design of the VMbench suite is displayed in Figure 4.1. The benchmarking system consists of the following components:

- VMbench tools, which are written in the Perl script language, control benchmarks and export data to MATLAB [The07].
- The VMbench user-level library encapsulates access to the diverse components of the virtual machine environment. The library binds to the respective components at load time.
- In order to take advantage of virtualization-specific benchmarks, user-level macro- and micro-benchmarks link to the VMbench user-level library.
- The guest kernel may be instrumented to run nano-benchmarks. However, benchmarks to determine the application-level performance run on a non-instrumented kernel.
- Similarly, if an IPVMM is present, it may or may not be prepared for nano-benchmarks.

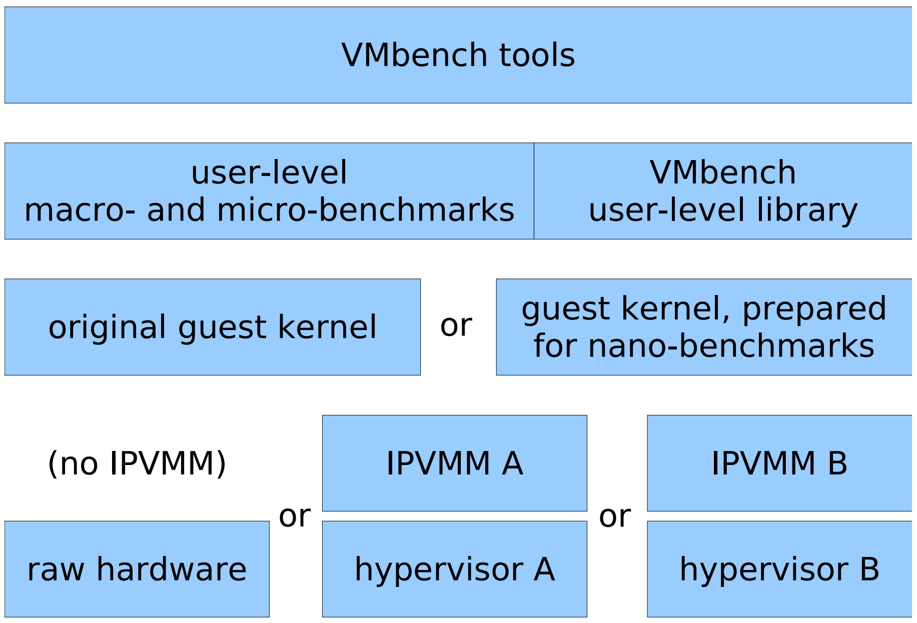

Figure 4.1: Hierarchical structure of VMbench

4.2 Kernel Instrumentation for Nano-Benchmarks

For low-level measurements, VMbench supports automatic instrumentation of functions, basic blocks, and sensitive instructions. Additionally, it includes facilities for manual instrumentation and reconfiguration at run time. The following subsections describe VMbench's instrumentation mechanisms and its benchmarking metrics.

4.2.1 Benchmarking Functions and Basic Blocks

The Linux kernel is usually compiled by GCC, the GNU C compiler. Triggered by command line arguments, GCC instruments functions or basic blocks with calls to profiling functions. Table 4.1 specifies GCC's profiling mechanisms. On the IA32 architecture, function calls incur relatively high overhead compared to most other instructions. For realistic and precise timekeeping, instrumentation must avoid inducing any additional overhead. Therefore, VMbench postprocesses GCC's assembler-language output with the afterburner assembler parser. The assembler parser detects the profiling function calls, which GCC inserts in the assembler code, and replaces them with low-overhead benchmarking code.

It was worth to note that, at the assembler level, a function has one entry but may have multiple exits. The metric code at a function entry and all corresponding exits must refer to the same counter, so allocation of new counters must take

Table 4.1: GCC's profiling mechanisms

Functional primitive	**Mechanism**	**GCC flag**	**Profiling function**
Function entries	gprof	-pg	mcount
Basic blocks	gcov	-fprofile-arcs	__gcov_merge_add
Function entry and exit	cyg_profile	-finstrument-functions	__cyg_profile_func_enter, __cyg_profile_func_exit

place when parsing a function entry. Function exits refer to the counter that was allocated when parsing the preceding function entry. In contrast, jump labels must be unique within each assembler file, such that new label names are generated on each function entry and exit.

4.2.2 Benchmarking Sensitive Instructions

The afterburner assembler parser was originally developed to pre-virtualize operating system kernel code. Specifically, it detects and instruments virtualization-sensitive instructions. VMbench uses the assembler parser to insert benchmarking code for sensitive instructions. The build infrastructure may activate pre-virtualization and benchmarking independently at file granularity, allowing to compare latency of original and pre-virtualized sensitive instructions.

Unfortunately, not all sensitive instructions can be identified at the assembler level: Memory-sensitive instructions are merely sensitive in some contexts. For example, a *mov* instruction, which is virtualization-insensitive in most cases, becomes sensitive when it modifies a page table. To cope with memory-sensitive instructions, VMbench expects memory-sensitive instructions to be marked in the assembler code with pseudo-instructions. These pseudo-instructions are replaced by benchmarking code.

Analogous to how memory-sensitive instructions are identified, VMbench provides the facility to manually instrument arbitrary places in the kernel code. This enables benchmarking specific control flow such as overall I/O latency or subsets of sensitive instructions. For example, VMbench records system call invocations using manual instrumentation.

4.2.3 Rewriting the Assembler Code

Assembler-level rewriting is relatively convenient, because symbolic names are used to name code and data locations. In contrast to higher-level languages, side effects are easier to control.

Figure 4.2 displays the ELF file sections related to benchmarking. The location of the benchmarking code, which is needed to reconfigure the metric, is

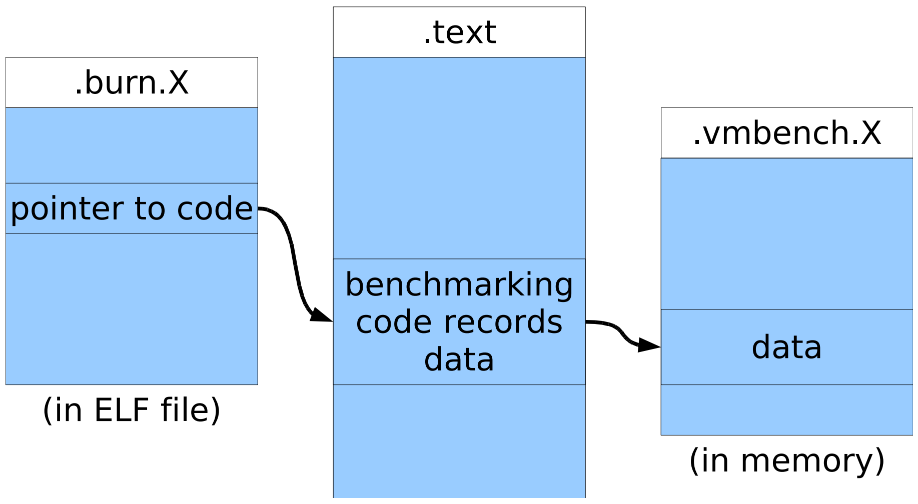

Figure 4.2: ELF file sections that are relevant to low-level benchmarking
X stands for the metric name.

specified in a non-loadable file section to minimize data overhead during runtime. I extended the afterburner assembler parser to emit benchmarking code as previously discussed. The separation between syntax and semantics was only tainted for instrumenting functions, where the assembler parser identifies calls to benchmarking functions by the name of the called function.

VMbench's nano-benchmarks allow to choose among several metrics. Additional metrics, such as timestamp samples, access to the processor's performance monitoring counters, and calls to high-level C or C++ functions respective methods, are already prepared but currently not in use.

4.3 Implementation of Micro-Benchmarks

As indicated in subsection 3.5.3, VMbench's micro-benchmarks are similar to those from the lmbench suite [MS96]. In contrast to lmbench, VMbench accounts time more precisely using the time stamp counter, and it leaves all statistical analysis to dedicated statistics software.

As yet, VMbench includes micro-benchmarks to determine the latency of system call, page fault, exception, sending and ignoring a signal, sending and handling a signal, and installing a signal handler.

- The *getppid* system call, which retrieves the identification of the parent process, represents the base cost of two context switches, from the application to the operating system kernel and back to the application. The kernel obtains the identification of the parent process easily from the process control

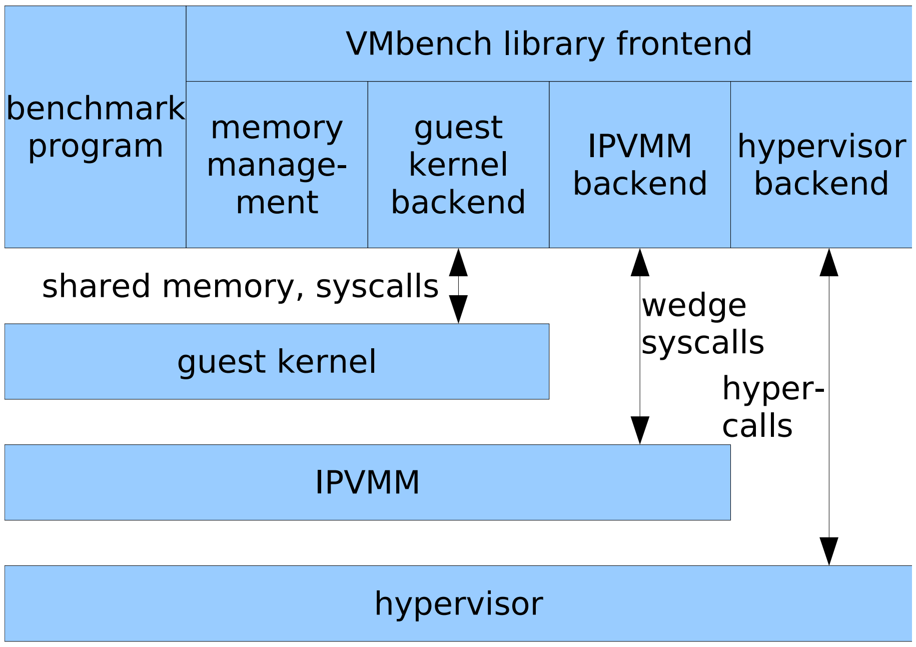

Figure 4.3: Components of the VMbench user-level library

block, such that *getppid* is one of the fastest system calls. A *getppid* system call typically does not execute any virtualization-sensitive instructions.

- A page fault entails at least one kernel entry and exit. In addition, the kernel executes several sensitive instructions, and, if necessary, initiates I/O. A page fault results in a sequence of at least eight sensitive instructions: *mov, sti, pushf, cli, popf, pgd_read, pgd_read, pte_set*.

- A handled signal has comparable overhead to two system calls: *kill* and *sigreturn*. The *kill* system call or a hardware exception enters the kernel and activates the signal handler at user mode. The *sigreturn* system call returns from the signal handler and reactivates the regular instruction stream. The whole procedure comprises 23 sensitive instructions, most of them being related to synchronization by clearing and setting the interrupt flag.

- An unhandled signal results in a sequence of three sensitive instructions: *pushf, cli, popf*.

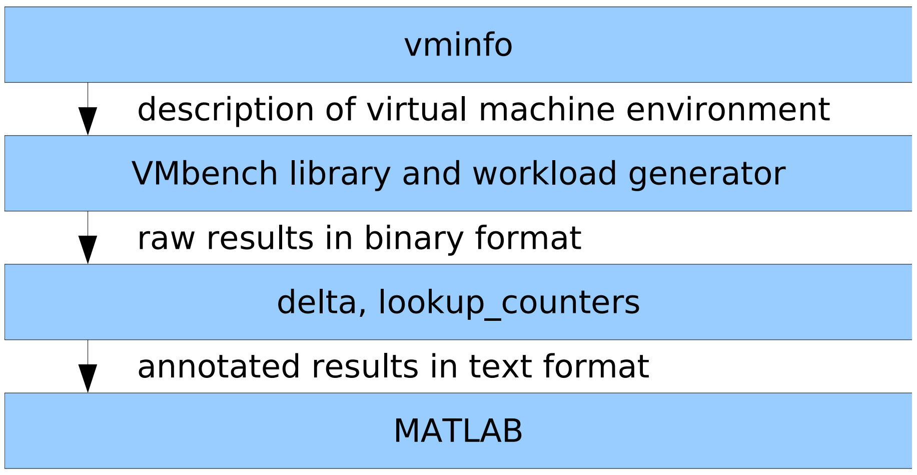
vminfo
description of virtual machine environment
VMbench library and workload generator
raw results in binary format
delta, lookup_counters
annotated results in text format
MATLAB

Figure 4.4: Data flow in VMbench

4.4 Data Export

The VMbench user-level library gathers data from the various components of a virtual machine environment. Each component offers different interfaces for communication. Thus, the library includes several backends, each communicating with one virtual machine environment component. Figure 4.3 shows VMbench's backends and how each backend interacts with a distinct component.

Communication with the guest operating system kernel is implemented via shared memory. In Linux kernels before release 2.6.12, the */dev/kmem* device was nonfunctional, so the necessary code was backported from release 2.6.17 in straightforward manner. In addition to shared memory, Linux offers profiling system calls such as *profil* and *getrusage*. Adding a special system call for VMbench was not necessary.

Counter access must be atomic, that is, it must not be interleaved with any other activity that changes the counters. To avoid page faults, VMbench pages all buffers in before using them.

4.5 Benchmarking Control

The benchmarking process is based on several tools (see Figure 4.4):

- The *vminfo* tool provides a uniform way to access information about virtual machines.
- Workload generators, such as the *extract_counters* tool, use the VMbench library to conduct benchmarks in a uniform way.

- The *delta* tool computes the difference between two files containing binary counter values, according to the counter's metric.
- The *lookup_counters* tool associates the binary counter values with external information about what the counters represent.

The tools *vminfo*, *delta*, and *lookup_counters* are not performance-critical, in that they run either before or after measuring. Implemented in the Perl language, they share some of their code in a Perl library.

4.5.1 The *vminfo* Tool

The benchmarking process needs two sorts of information about the virtual machine environment: the presence of a particular component and what measurements a component supports.

Presence detection is specific to the respective component: The *uname* command extracts the current version of the Linux kernel from the *proc* filesystem. An IPVMM should intercept the *cpuid* instruction and specify its own name in the processor model field. Generally, timing skew analysis allows to detect a hypervisor. Detection of the L4 hypervisor is more convenient, because a special invalid instruction triggers publishing the kernel interface page.

VMbench stores metric and counter names in object file sections that are not loaded upon execution. To avoid transferring non-relevant information, VMbench supports caching object file information. The Perl library extracts metric and counter names from the object files or from the cache, and instructs the VMbench user-level library how to access the counters.

4.5.2 Workload Generators

Workload generators link to the VMbench user-level library in order to record virtualization-specific events.

The *extract_counters* workload generator, for example, starts measurements, runs an arbitrary program by means of the *execve* system call, and finishes measurements when the program terminates. Thus, it serves as a wrapper for arbitrary programs which need not be instrumented.

The *forkwait* program [AA06] stresses process creation and termination. These operations are particularly susceptible to virtualization.

VMbench's micro-benchmarks (see Section 4.3) are also linked to the VMbench user-level library, allowing to record virtualization-sensitive events during synthesized activity.

4.5.3 The *lookup_counters* Tool

Depending on which metric was used, the data specifies either implicitly or explicitly the primitive operation that produced it. In case of tracing, the counters are allocated in the same order as the pertaining primitive operations. In case of sampling, each data sample specifies the unique identifier of the instruction that produced the sample. The *lookup_counter* Perl script reads a file containing binary counter data. Then it looks up the human-readable names of the primitive operations, for example function names, sensitive instructions, or memory addresses.

4.6 Relating Micro- and Macro-Benchmarks

A number of statistic software suites exists, among them GNU R, MATLAB with its Statistic Toolbox, and the IT++ library. I chose MATLAB, because it combines both vector mathematics and statistical analysis. It has a graphical user interface with built-in support for common tasks such as data import and plotting. The freely available GNU Octave language is compatible with most MATLAB code. It only lacks the convenient graphical environment and more sophisticated statistics functions.

Chapter 5

Results

This chapter presents initial results that were produced using the VMbench suite. After introducing the benchmarking environment and the configurations under examination, the overhead of kernel-level nano-benchmarks is evaluated. The main focus of this chapter is to exemplify how to characterize virtual machine environments using VMbench's three-stage methodology.

5.1 Benchmarking Environment

I ran the experiments on an Intel Pentium 4 CPU model 4 stepping R0. The processor, built using 90 nm technology, ran at 3.8 GHz core speed and 800 MHz bus speed. It had a 64-entry instruction TLB and a 64-entry data TLB. In all configurations, the RAM available to Linux was restricted to 256MB.

The system under test booted a stripped-down version of Debian 3.1 from a 12 MB-sized ramdisk image. Additional storage was allocated on temporary ramdisks. The kernelbuild benchmark ran on a DMA-capable IDE hard drive in a *chroot* environment. To avoid interferences by the network interface card, benchmarks that do not need networking were run with the network interface taken down.

Using GCC version 3.3.5 and the afterburner assembler rewriter for pre-virtualization, I built the Linux kernel version 2.6.9 in unmodified form and in different micro-benchmarking configurations. Hyperthreading and multiprocessing were disabled in the kernel. Fast system calls were enabled, but they are active only if the respective hypervisor supports them.

5.2 Configurations Under Examination

I compared performance on raw hardware with performance on two hypervisors, L4 and Xen. In addition, I have verified that VMbench is compatible with a popular closed-source virtual machine monitor. However, the license agreement does not permit publishing the results. Given that the virtual machine monitor virtualizes performance monitoring hardware, benchmarking results are less precise.

Xen is a native hypervisor for the IA32 and AMD64 architectures. It supports para-virtualized adaptations of Linux and several other operating systems as guests. With hardware-assisted virtualization, it can also support unmodified Linux and Windows. Xen runs guest kernels in privilege ring 1, whereas guest applications run in privilege ring 3. Thus, it takes advantage of segmentation support built in IA32 processors for switching between guest kernel and applications.

L4 is a mature second-generation microkernel [Lie96]. Given that the microkernel concept allows only basic security-related functionality to run in kernel mode, user-mode L4 applications can implement diverse personalities at liberty. Taking advantage of this flexibility, one popular option is to run L4 with a user-mode virtual machine monitor that hosts Linux or other guest operating systems [LUSG04]. The ensemble is called *L4Ka virtualization environment*. However, L4 has not been designed as a specialized hypervisor, such that it restricts the virtual machine monitor's view on the virtual processor in some aspects. Work is underway to make L4 support virtualization in a more straightforward manner.

Xen and L4 were configured for pass-through device access, which has been described in subsection 2.1.5. Benchmarking other configurations of the virtual machine environment is left for future work.

5.3 Nano-Benchmark Overhead

The Linux kernel under examination contains 5571 virtualization-sensitive instructions. For example, during a kernel build, about 57,591,000 sensitive instructions are executed. The overhead of nano-benchmarking was approximately determined by comparing the overall run time of a Linux kernel build on a non-instrumented kernel and on several instrumented instrumented kernels (see Figure 5.1). On the one hand, the measurements indicate that the overhead of the *count* metric for sensitive instructions is negligible. Considering that the *count* metric has minimal impact on memory and cache (see Table 3.1), too, it is well suited to record the usage count of sensitive instructions. On the other hand, the higher overhead of *latency samples* and *average latency* support the decision to abandon vertical benchmarking (see Subsection 3.6.1). Storage for latency samples is limited, such that the *latency samples* metric looses significance for long benchmarking runs.

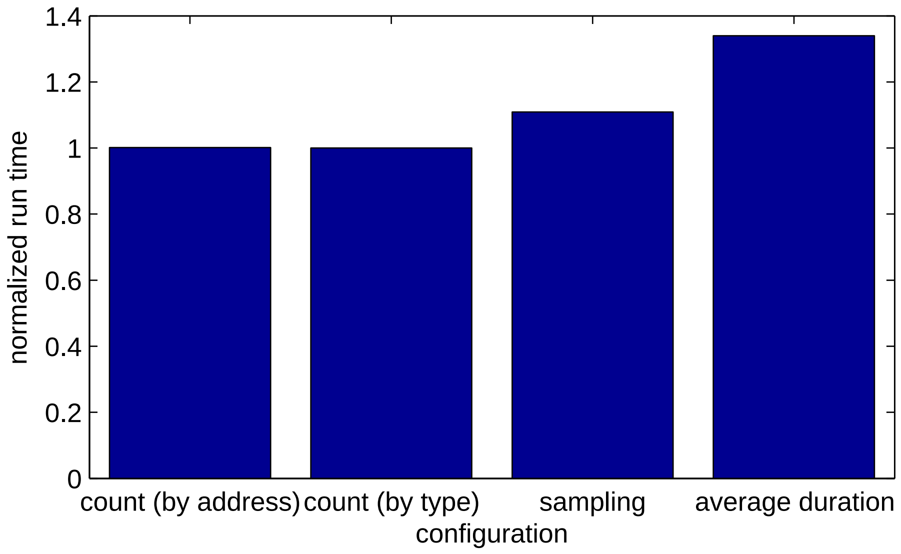

Figure 5.1: Nano-benchmarking effect on run time of kernelbuild, normalized to non-instrumented run time

5.4 Hypervisor Performance Signature

This section describes the first stage of the suggested benchmarking methodology: the determination of the hypervisor performance signature.

5.4.1 Sensitive Instruction Signature

Histograms allow to judge how appropriate the assumption of a normal distribution is. The histograms 5.2, 5.3, and 5.4 demonstrate the empirical distribution of sensitive instructions under forkwait workload on raw hardware, on L4, and on Xen 2. For several of the sensitive instructions, the histograms rarely justify a normality assumption in the strict sense. However, most histograms resemble a degenerate distribution, in which case the median of the values represents their central tendency, too.

Most sensitive instructions take up to 200 cycles. Infrequent outliers are due to cache misses or due to non-atomic latency measurements.

- On raw hardware, the *in* and *out* instructions, which access external devices, take considerably longer than in virtualized environments, because they operate on real hardware. The *invlpg* instruction, which flushes the TLB, takes 550 to 700 cycles. The *mov* instruction takes up to 600 cycles in case of a cache miss.

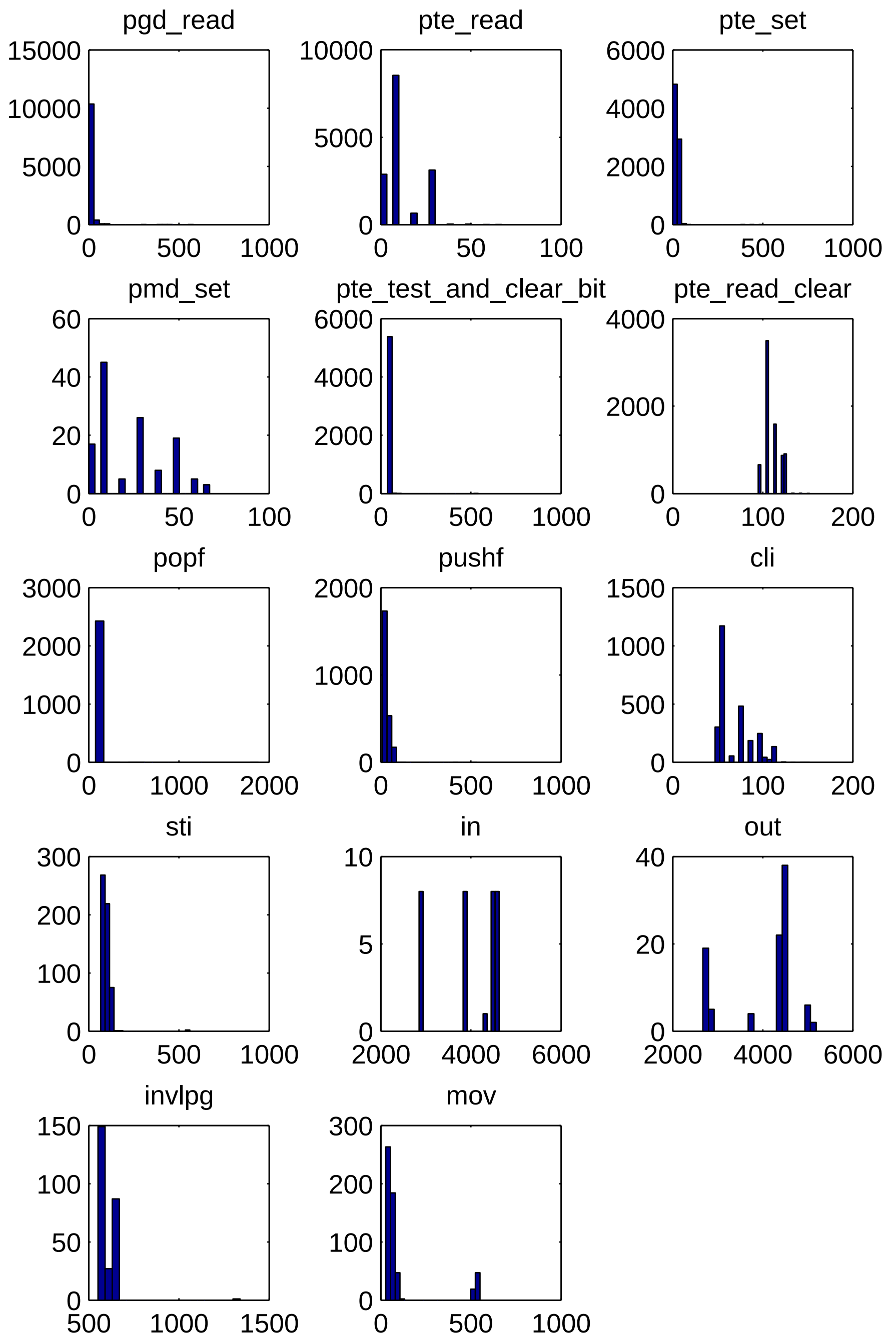

Figure 5.2: Sensitive instruction histogram of pre-virtualization on raw hardware
The X-axes reflect the range of values in 20 classes, whereas the Y-axes specify the frequency of sensitive instructions in each class.

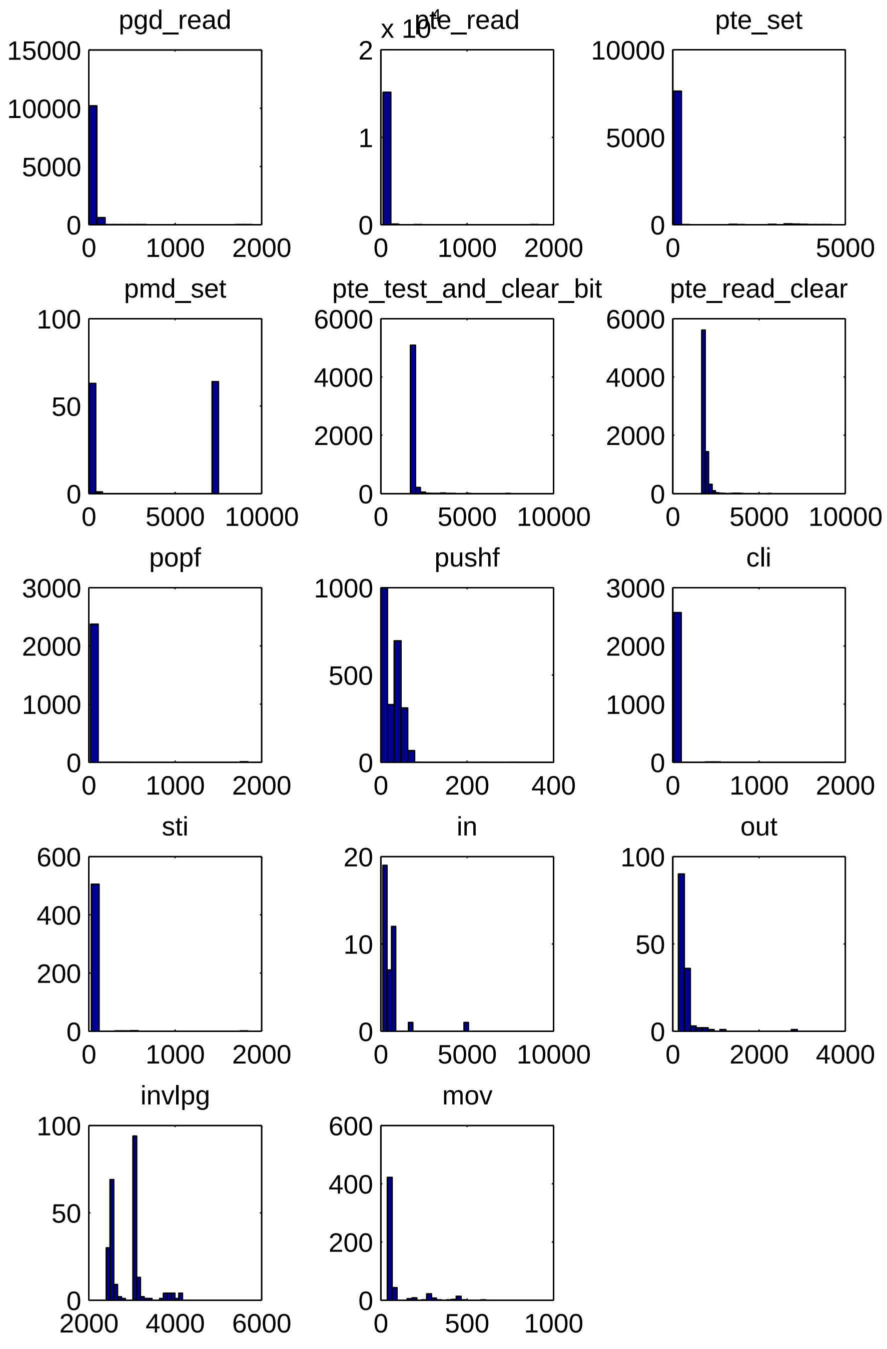

Figure 5.3: Sensitive instruction histogram of pre-virtualization on L4
The X-axes reflect the range of values in 20 classes, whereas the Y-axes specify the frequency of sensitive instructions in each class.

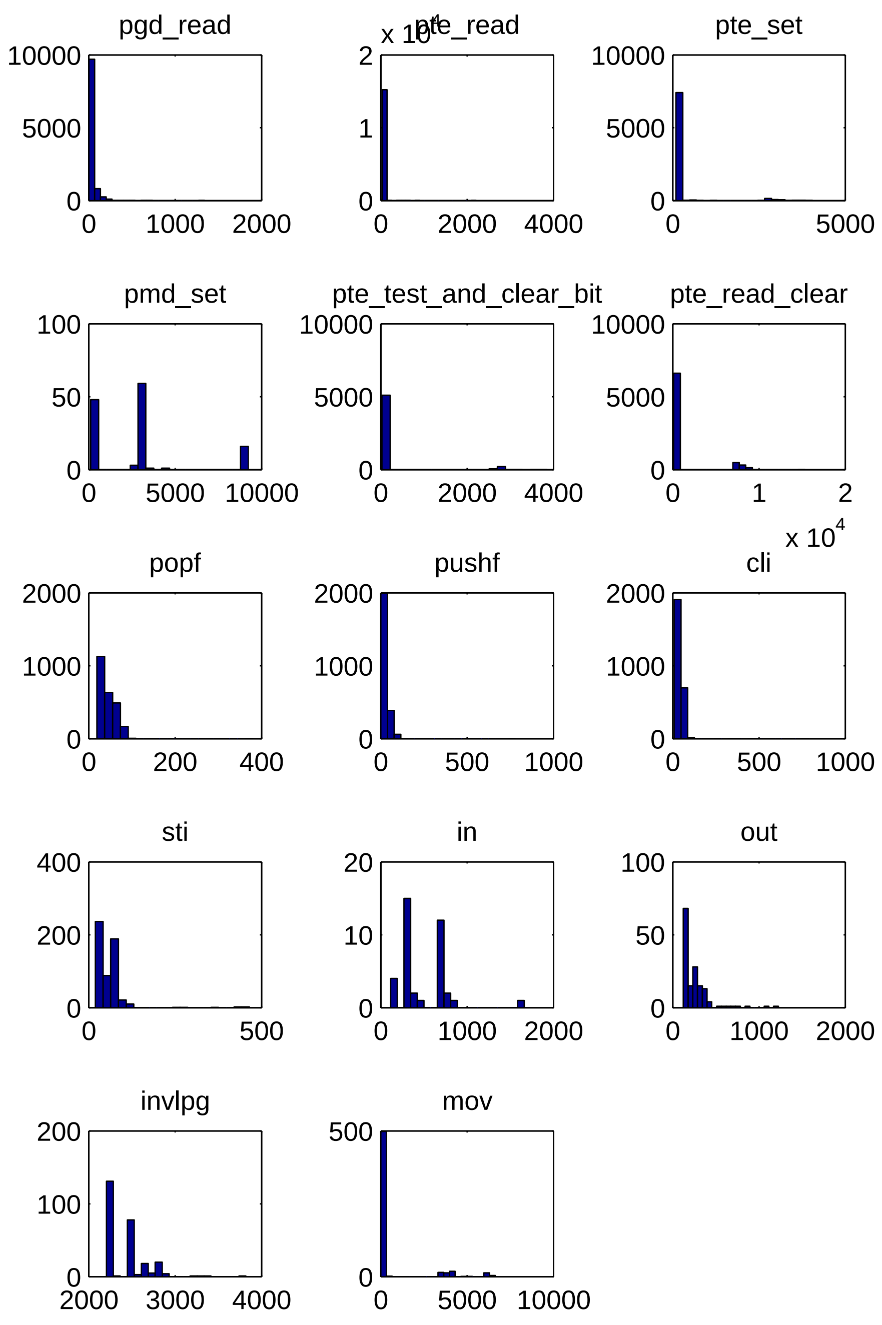

Figure 5.4: Sensitive instruction histogram of pre-virtualization on Xen 2
The X-axes reflect the range of values in 20 classes, whereas the Y-axes specify the frequency of sensitive instructions in each class.

- With the L4Ka virtualization environment, half of the *pmd_set* pseudo instruction take about 8000 cycles. The *pte_test_and_clear_bit* pseudo instruction takes 2000 cycles. The *pte_read_clear* pseudo instruction takes more than 1500 cycles. The *in* and *out* instructions take less than on raw hardware, because they access virtual devices. The *invlpg* instruction takes between 2500 and 6000 cycles, with major peaks at 2700, 3400, and 4200 cycles.

- On the Xen 2 hypervisor, several histograms show minor peaks at 2400 cycles, which are possibly due to timer interrupts arriving during non-atomic measurement and instruction emulation. The *pmd_set* instruction has three distinct peaks at 200, 3100, and 9200 cycles. The first peaks is formed by about 40 percent of the data, the second by about 50 percent.

Assuming normal distributions, the sensitive instruction signature reduces a sensitive instruction's latency to its median value. The boxplot diagrams 5.5, 5.6, and 5.7 display quartiles, outliers, and medians of the sensitive instructions. Each box ranges from the lower quartile to the upper quartile. Thus, it includes fifty percent of the data. The whiskers extend to include the rest of the data minus outliers. Given that outliers below the lower quartile are infrequent, the boxplot diagrams confirm that the median values of latencies are acceptable for usage in a best-case hypervisor signature.

5.4.2 Mode and Context Switching Latency

Figure 5.8 displays the latency of different events related to processor mode or context switching. The 95% confidence intervals for the data at 99 degrees of freedom (that is, 100 independent benchmarking runs) are less than 0.5% from the mean value.

It is interesting to note that pre-virtualized Linux on Xen 2 handles signals faster than it does on raw hardware. During a single signal handling operation, the virtual interrupt flag is cleared or set at least fourteen times. I suppose this is the reason for the unexpected performance enhancement.

5.4.3 Baseline Virtualization Overhead

The time series diagrams 5.9, 5.10, and 5.12 display the dilation of an idle loop (see Subsection 3.7.2). The inner loop was calibrated to run for at least one millisecond. The benchmarking application, which executes at minimum priority, repeats the inner loop several hundred times. Interrupts and activity of the operating system or other applications dilate the execution time of the inner loop. On

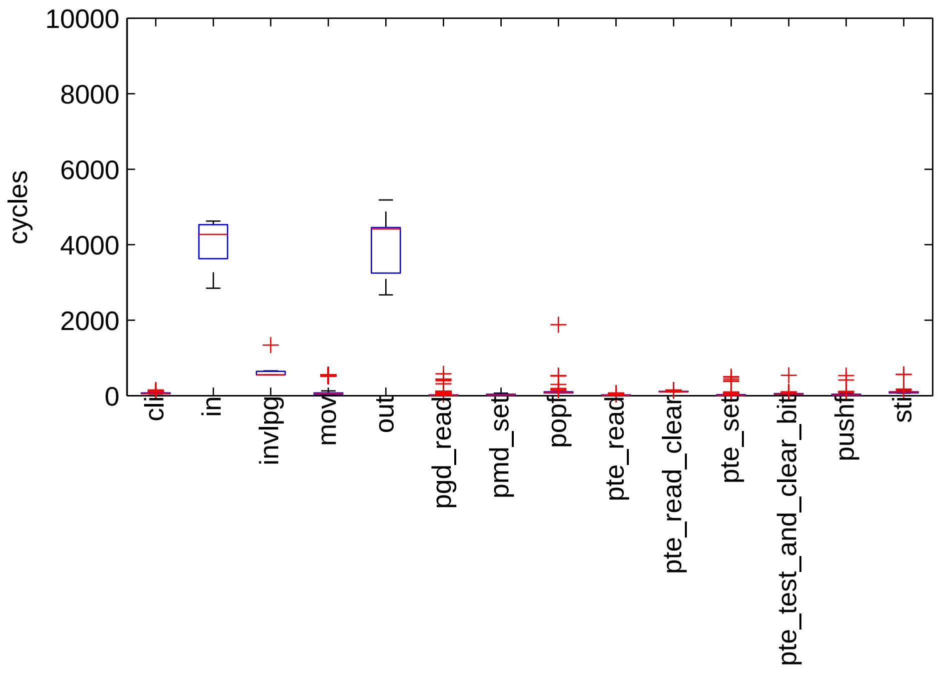

Figure 5.5: Sensitive instruction signature of pre-virtualization on raw hardware

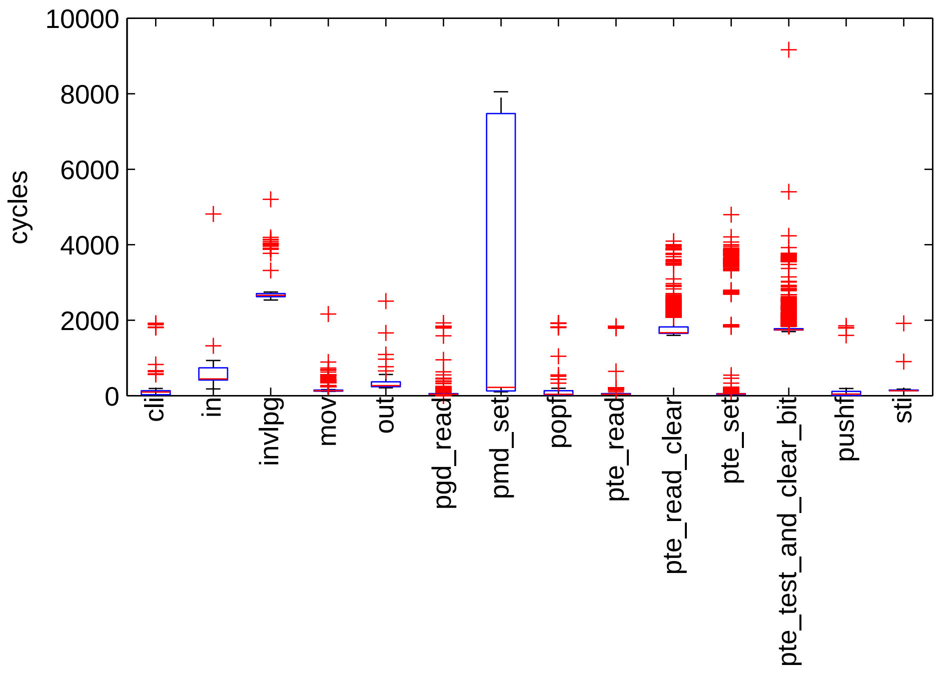

Figure 5.6: Sensitive instruction signature of pre-virtualization on L4

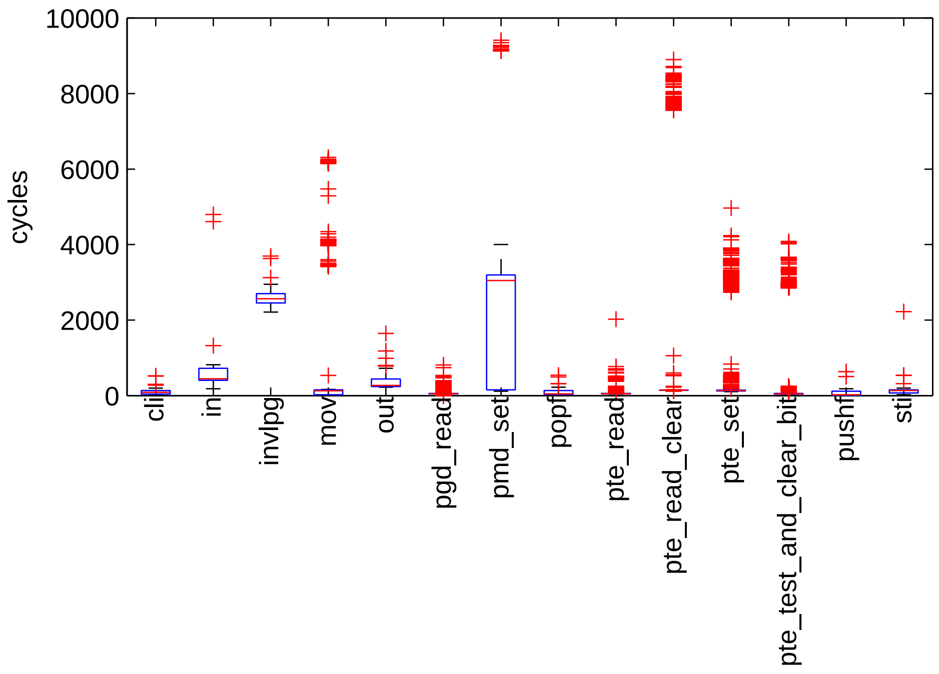

Figure 5.7: Sensitive instruction signature of pre-virtualization on Xen 2

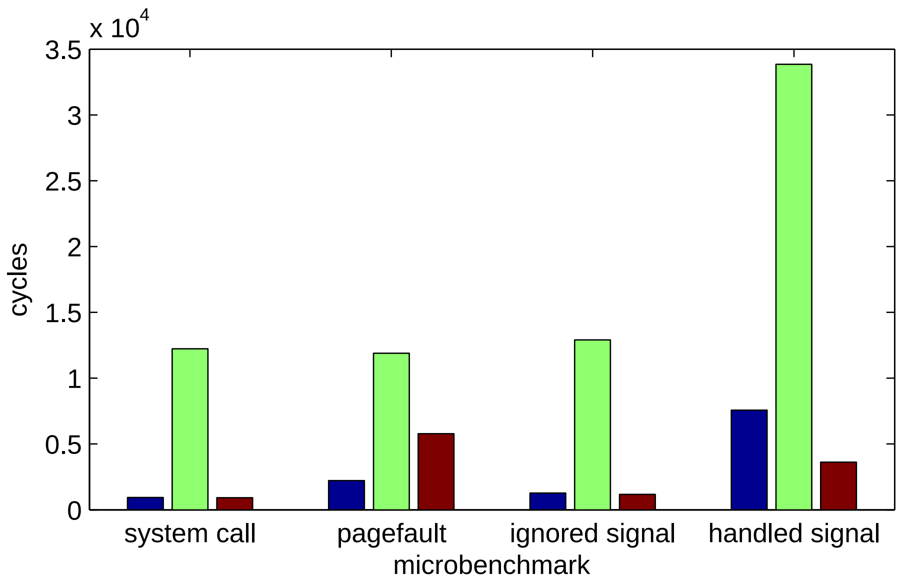

Figure 5.8: Mode and context switching signature of pre-virtualization on raw hardware (left), L4 (center) and Xen 2 (right)

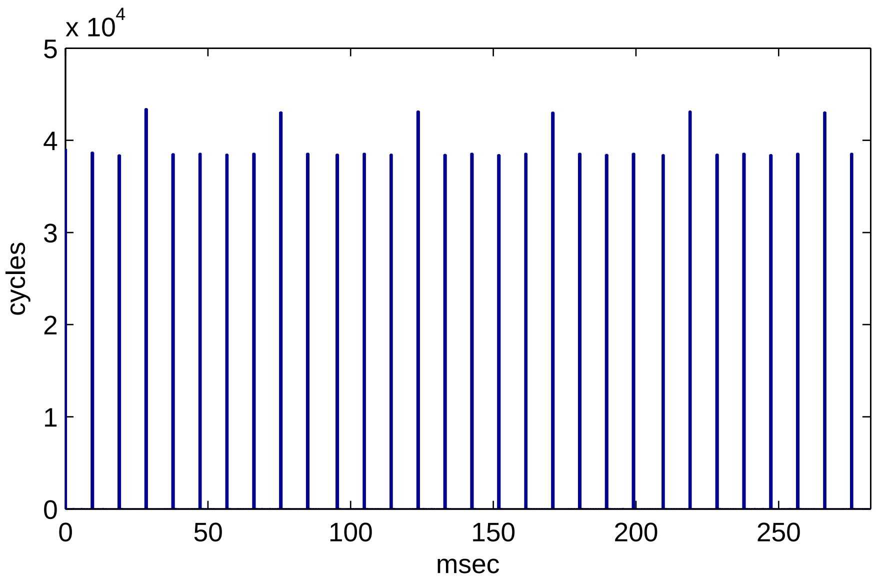

Figure 5.9: Idle profile of pre-virtualized Linux on raw hardware

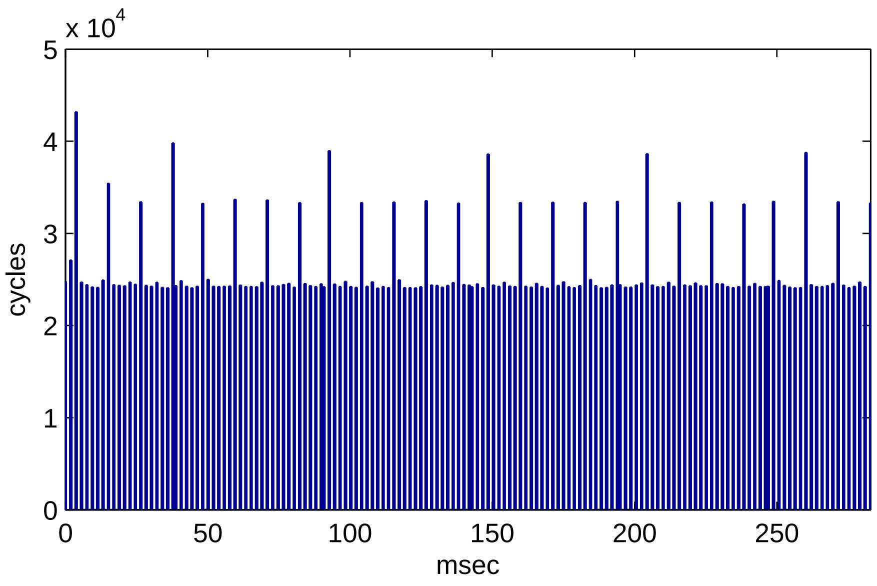

Figure 5.10: Idle profile of pre-virtualized Linux on L4

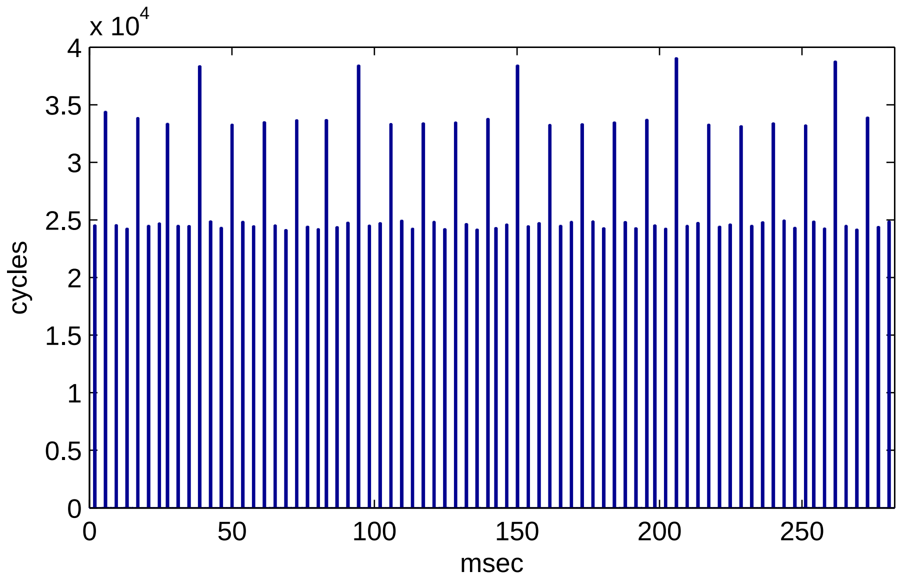

Figure 5.11: Idle profile of pre-virtualized Linux on L4 with reduced timer interrupt frequency

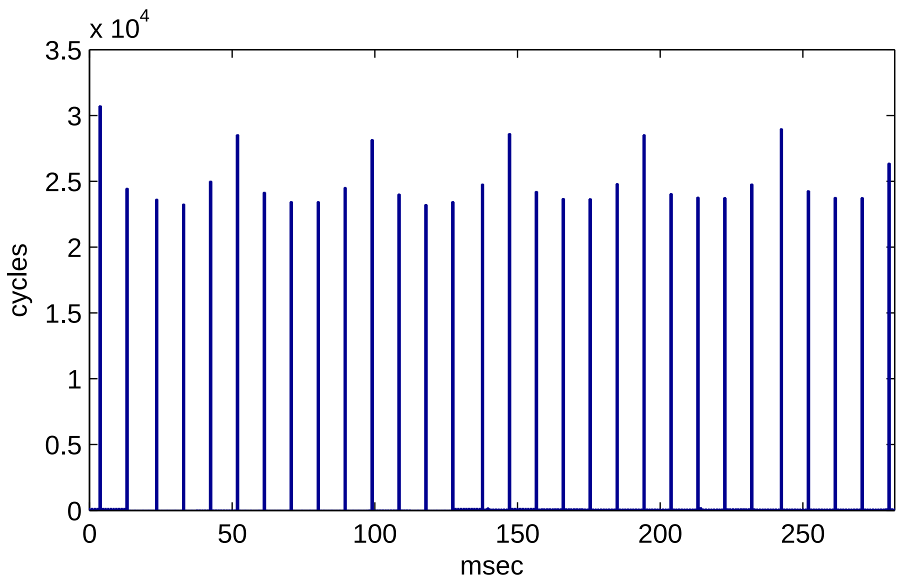

Figure 5.12: Idle profile of pre-virtualized Linux on Xen

raw hardware (Figure 5.9), incoming timer interrupts cause delays of about 38,500 cycles every ten milliseconds. Every fifth timer interrupt activates the scheduler, such that the idle loop takes 4,600 cycles longer.

On L4, virtual timer interrupts are modelled through an IPC timeout for the IRQ handler thread [LUC$^+$06]. The microkernel handles more timer interrupts than it passes to the guest operating system (Figure 5.10). In a virtual machine environment, the additional interrupts are useless, since they do not trigger any activity. Changing the L4 kernel to generate less timer interrupts was trivial and reduced the timer overhead (see Figure 5.11).

The profile on Xen (Figure 5.12) looks very similar to raw hardware, however, timer interrupts show more variation. Analysis of the absolute numbers reveals that timer interrupt handling takes longer on raw hardware than on L4 or Xen, probably due to better cache utilization.

5.5 Virtual Machine Performance Under Optimal Conditions

As suggested in Section 3.6, the second stage of the virtual machine benchmarking methodology predicts lower bounds for execution time of realistic workload. In Figures 5.13, 5.14, 5.15, and 5.16, baseline execution time is shown right at the bottom, latency of selected sensitive instructions is above, is latency of context switches is yet above, and latency caused by stolen time is upmost.

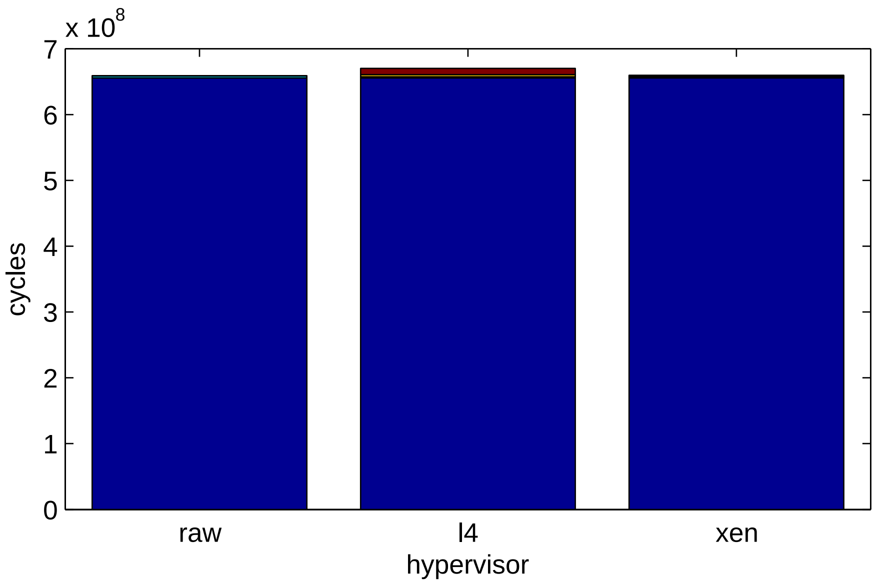

Figure 5.13: Performance analysis of *gzip* under different hypervisors

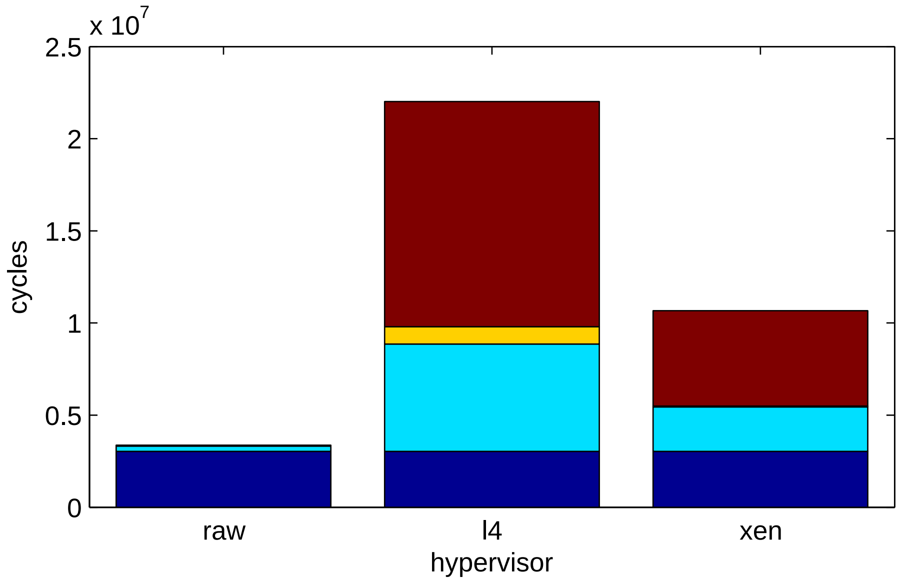

Figure 5.14: Performance analysis of *forkwait* under different hypervisors

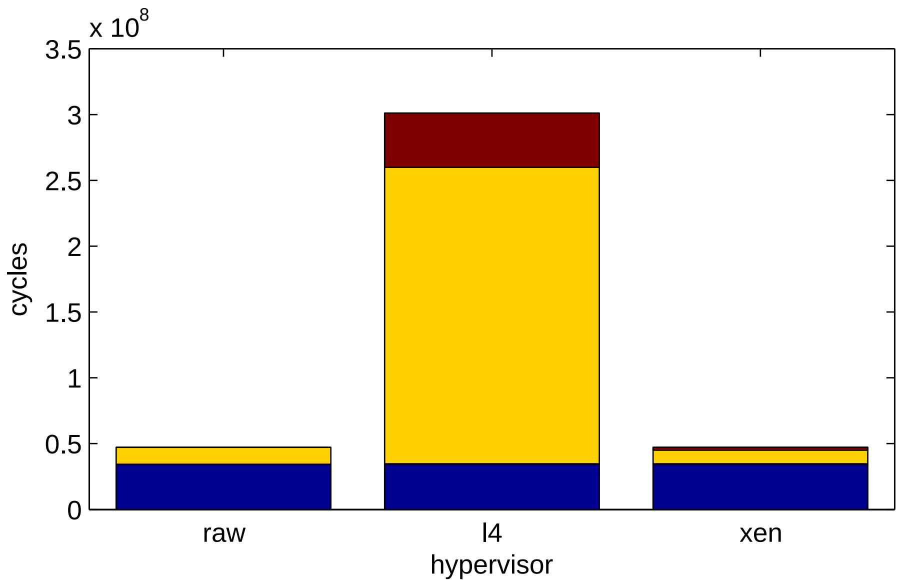

Figure 5.15: Performance analysis of *find* under different hypervisors

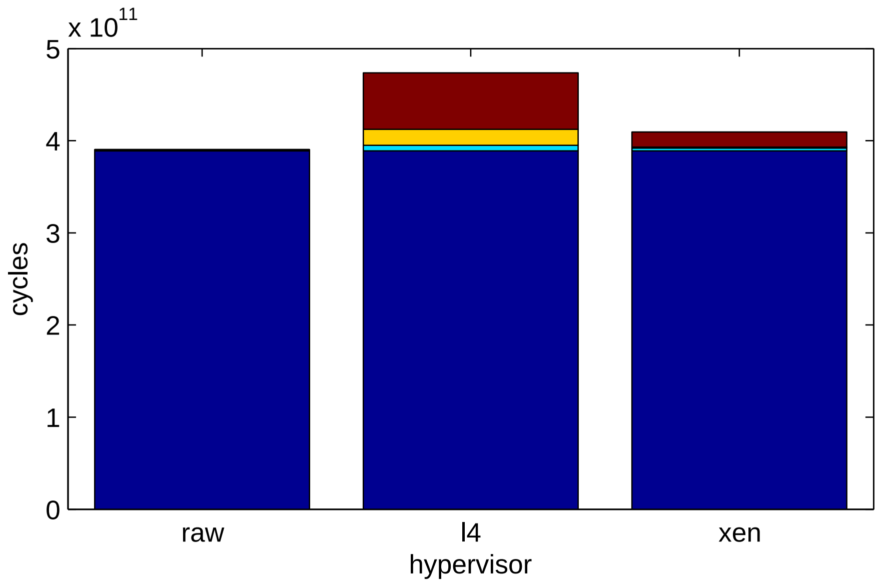

Figure 5.16: Performance analysis of *kernelbuild* under different hypervisors

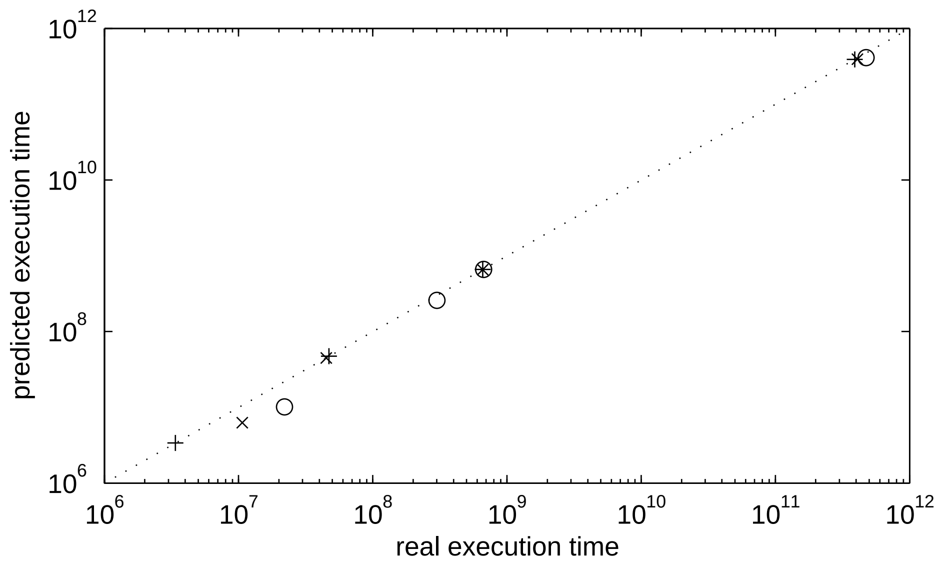

Figure 5.17: Prediction accuracy with VMbench

The *gzip* workload consists of compressing an 1.5 MB archive containing a mix of highly compressible and not very compressible files. This activity is computing-intensive. If virtual machine interference is precluded, it is hardly susceptible to virtualization. Both hypervisors perform almost as well as raw hardware (see Figure 5.13).

In contrast, the *forkwait* workload [AA06] is very operating-system intensive and highly susceptible to virtualization. For VMbench's purpose, it consecutively creates 16 processes, which simply terminated themselves right after being created. The results (see Figure 5.14) indicate that both context switching and sensitive instructions contribute significantly to the virtualization overhead.

The *find* workload stresses system calls, such that kernel entry and exit form the major virtualization overhead. In contrast, the overhead through virtualization-sensitive instructions is rather low.

The *kernelbuild* workload builds the Linux kernel for the IA32 architecture in its minimal configuration. More than 1,038,400 system calls constitute the major virtualization overhead.

5.6 Virtual Machine Interference

Figure 5.17 summarizes the accuracy of VMbench's performance predictions. The log-log scale enables comparing the different types of workload in a single plot. Performance on raw hardware is marked with +, performance on L4

Table 5.1: Prediction accuracy with VMbench

Workload	**Pre-Virtualization on L4**	**Pre-Virtualization on Xen**
gzip	98.6%	99.7%
forkwait	45.9%	58.6%
find	85.6%	99.6%
kernelbuild	86.9%	96.1%

with *o*, and performance on Xen with *x*. Table 5.1 contains the ratio of predicted execution time to real execution time.

The ratios confirm that VMbench predicts best-case performance: In neither case, virtualized performance is lower than estimated. However, they also indicate that virtual machine interference has quite large impact in some cases. To some extent, the inadequacy of the Pentium IV for accurate nano-benchmarks may be the reason for prediction errors.

A vast amount of possibilities exists to evaluate how virtual machines interfere. Future work should extend this analysis by investigating the performance of multiple virtual machines that compete for resources. For example, one virtual machine could run the instrumented idle loop, while another builds a kernel or transfers data over the network.

Chapter 6

Conclusion

Given the huge popularity of virtual machines and the increasing diversity of virtualization techniques, performance analysis of virtual machine environments is constantly gaining importance. This contribution pursues a decompositional analysis of runtime in virtualized environments, resulting in a proposal for a virtual machine benchmarking methodology and in the implementation of VMbench, a prototype virtual machine benchmarking suite.

6.1 Contributions of This Work

The suggested benchmarking methodology addresses the "tension between realism and reproducibility" [Mog99] by proceeding from artificial micro-benchmarks to realistic measurements at macroscopic level.

The VMbench suite deals with several issues that arise when benchmarking virtualized environments. Particularly the pre-virtualization paradigm [LUC$^+$06] enables benchmarking a broad variety of virtualization techniques. Compiler-aided instrumentation allows to measure the latency of different primitive operations. VMbench coordinates benchmarking among the components of the virtual machine environment. It exports results for evaluation with statistical software, which in turn applies the linear model.

The initial evaluation indicates that VMbench predicts the time dilation of operating-system intensive workload well enough to predict the execution time of virtualization techniques under consideration on an ordinal scale.

6.2 Suggestions for Future Work

Considering both the popularity and the diverse interface of the IA32 architecture, I find it unlikely that, within the next years, a single hypervisor will dominate

IA32-based virtualization. Thus, performance evaluation of virtual machines will remain a topic of high interest. The increase of publications on this topic within the last year reinforces this claim.

However, this thesis has discussed performance evaluation and prediction of virtual machines for a narrow domain. It is up to future work to expand the methodology to diverse areas of application.

Improve Prediction Accuracy In order to improve prediction accuracy, specific knowledge about how a hypervisor enforces virtualization is necessary. Upcoming technologies may embody additional performance bottlenecks that the initial VMbench model does not anticipate. Moreover, as proposed by Seltzer et. al. [SKSZ99], simulation may be useful to refine the linear model. Given that the probability distribution of the observed data depends on how virtualization is realized, switching to a non-linear model is not very promising. Upcoming processor architectures [Adv06b] have more accurate facilities for precise time measurements.

Model Work Offloaded to Device Driver Virtual Machines As discussed in subsection 3.4.3, the abstraction of resource containers can model work offloaded to device driver virtual machines. Furthermore, the combination of VMbench and performance aggregation strategies might yield valuable insights about concurrency of virtual machines.

Describe Virtual Machine Configuration in a Portable Way In order to benchmark diverse hypervisors and virtual machine setups, portability is key. It would be convenient to be able to configure different virtual machine environments using common tools. Describing the setup of a virtual machine environment in XML format would also benefit other projects, such as virtual machine migration and data center administration. Similarly, XML-based data exchange would simplify data collection.

Port VMbench to Other Processor Architectures and Operating Systems As soon as pre-virtualization supports other processor architectures such as AMD64 and PowerPC and other operating systems such as BSD, Windows, and MacOS X, the VMbench suite can be ported to these. Given that modern processor architectures have similar performance monitoring capabilities, the ports should be straightforward. Porting VMbench to other processor architectures will improve prediction accuracy, because Pentium 4 is probably the worst-case platform for accurate instruction-level timing. It would be exciting to see how much latest processor features such as IOMMUs [Adv06a] accelerate virtual machine performance.

6.3 Concluding Remarks

Industry, researchers, and customers urgently need standards to compare the performance of virtual machines [CGS06]. Time will show whether a widely accepted, general, realistic, and accurate methodology for virtual machine performance analysis can be established.

List of Figures

List of Tables

Bibliography

[AA06] Keith Adams and Ole Agesen. 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*, pages 2–13, New York, NY, USA, 2006. ACM Press.

[ABD$^+$97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[Adv06a] Advanced Micro Devices, Inc. AMD I/O-virtualization technology (IOMMU) specification, February 2006.

[Adv06b] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual*. Austin, TX, USA, 2006.

[Adv06c] Advanced Micro Devices, Inc. *AMD64 Virtualization Codenamed Pacifica Technology, Secure Virtual Machine Architecture Reference Manual*. Austin, TX, USA, 2006.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 45–58, San Diego, CA, USA, 1999. USENIX.

[BS95] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 1–11, New York, NY, USA, 1995. ACM Press.

[BS97] Aaron B. Brown and Margo I. Seltzer. Operating system benchmarking in the wake of lmbench: a case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 214–224, New York, NY, USA, 1997. ACM Press.

[CDD$^+$] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. pages 135–144, San Diego, CA, USA. USENIX.

[CFH$^+$05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, San Diego, CA, USA, 2005. USENIX.

[CG05] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead of I/O processing in the Xen virtual machine monitor. pages 387–390, San Diego, CA, USA, April 2005. USENIX.

[CGS06] Jeffrey P. Casazza, Michael Greenfield, and Kan Shi. Redefining server performance characterization for virtualization benchmarking. *Intel Technology Journal*, 10(03), 2006.

[CI06] Microsoft Corp. and XenSource Inc. Microsoft and XenSource to develop interoperability for Windows Server Longhorn virtualization, June 2006.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, page 0133, 2001.

[Coo83] G. H. Cooper. An argument for soft layering of protocols. Technical report, Cambridge, MA, USA, 1983.

[DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): a network subsystem architecture for server systems. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, pages 261–275, New York, NY, USA, 1996. ACM Press.

[Dij68] Edsger W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[DKC^{+}02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, pages 211–224, New York, NY, USA, 2002. ACM Press.

[ELM^{+}03] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, 2003.

[EWCS96] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. *ACM SIGOPS Operating Systems Review*, 30(SI):185–199, 1996.

[FDF03] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society.

[FHN^{+}04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. New York, NY, USA, October 2004. ACM Press.

[Fra06] Keir Fraser. Progressive para-virtualization. September 2006.

[GCGV06] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. Technical report, Palo Alto, CA, USA, May 2006.

[GGC] D. Gupta, R. Gardner, and L. Cherkasova. XenMon: QoS monitoring and performance profiling tool. Technical Report HPL-2005-187, 2005, HP Laboratories, Palo Alto, CA, USA.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.

[Gol73] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[GPC^{+}03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.

[HSDH04] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[Int06] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corp., Santa Clara, CA, USA, 2006.

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LIJ^{+}98] Jochen Liedtke, Nayeem Islam, Trent Jaeger, Vsevolod Panteleenko, and Yoonho Park. Irreproducible benchmarks might be sometimes helpful. In *Proceedings of the 8th ACM SIGOPS European Workshop*, New York, NY, USA, September 7–10 1998. ACM Press.

[LUC^{+}05] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[LUC^{+}06] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, San Diego, CA, USA, December 2004. USENIX.

[Mac79] R. A. MacKinnon. The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines. *IBM Systems Journal*, 18(1):18–46, 1979.

[mer06] Merriam-webster's online dictionary. 2006.

[MHS$^+$] Vikram Makhija, Bruce Herndon, Paula Smith, Lisa Roderick, Eric Zamost, and Jennifer Anderson. VMmark: A scalable benchmark for virtualized systems. Technical Report VMware-TR-2006-002, Palo Alto, CA, USA.

[Mog99] Jeffrey C. Mogul. Brittle metrics in operating systems research. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

[Mog06] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. Technical Report HPL-2006-2, Palo Alto, CA, USA, January 2006.

[MS96] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. pages 279–295, San Diego, CA, USA, January 1996. USENIX.

[MSSP02] D. Mahrenholz, O. Spinczyk, and W. Schrder-Preikschat. Program instrumentation for debugging and monitoring with Aspect C. pages 249–256, 2002.

[MST$^+$05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, pages 13–23, New York, NY, USA, 2005. ACM Press.

[MUKX06] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *ACM SIGOPS Operating Systems Review*, 40(2):8–11, 2006.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, San Diego, CA, USA, 2005. USENIX.

[Ous90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *Proceedings of the USENIX 1990 Summer Technical Conference*, pages 247–256, San Diego, CA, USA, June 1990. USENIX.

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

[PS75] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, 1975.

[RI00] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, San Diego, CA, USA, 2000. USENIX.

[SBSM89] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12):1659–1679, 1989.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[SKSZ99] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*, page 102, Washington, DC, USA, 1999. IEEE Computer Society.

[SM] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 155–166, San Diego, CA, USA. USENIX.

[SN05] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms For Systems And Processes*. Morgan Kaufmann Publishers, San Fransisco, CA, USA, May 2005.

[SS96] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.

[Sta02] Carl Staelin. lmbench3: measuring scalability. Technical Report HPL-2002-313, Palo Alto, CA, USA, November8 2002.

[The07] The MathWorks, Inc. *Statistics Toolbox For Use with MATLAB*. Natick, MA, USA, 2007.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[ULSD04] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 6–7 2004.

[UNR$^+$05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer*, 38(5):48–56, 2005.

[VMwa] VMware, Inc. A performance comparison of hypervisors.

[VMwb] VMware, Inc. *VMI Specification, Paravirtualization API Version 2.0*. VMware, Inc., Palo Alto, CA, USA.

[Was04] Larry Alan Wasserman. *All of statistics: a concise course in statistical inference*. Springer, New York, Berlin, Heidelberg, May 2004.

[WCSG05] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Rethinking the design of virtual machine monitors. *IEEE Computer*, 38(5):57–62, 2005.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[Xen] XenSource, Inc. A performance comparison of commercial hypervisors.

[You73] Carl J. Young. Extended architecture and hypervisor performance. In *Proceedings of the workshop on virtual computer systems*, pages 177–183, New York, NY, USA, 1973. ACM Press.

[ZS00] Xiaolan Zhang and Margo Seltzer. HBench:Java: an application-specific benchmarking framework for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 62–70, New York, NY, USA, 2000. ACM Press.