Lehrstuhl für Rechnertechnik und Rechnerorganisation
der Technischen Universität München

Live Replication of Paravirtual Machines

Daniel Stodden

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Arndt Bode
	2.	Univ.-Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 20.03.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.07.2008 angenommen.

Abstract

System virtualization has seen largely renewed interest in recent years, primarily due to the emerge of effective virtualization techniques for commodity platforms in both hardware and software.

Beyond improved resource utilization and administration cost, driving past and present adoption in the data center, virtual machines offer a fair degree of system state encapsulation, which promotes practical advances in workload migration, system debugging, profiling and, last but not least, an ongoing "commoditization" in high-availability system design.

This thesis investigates deterministic replay and semi-active replication for system paravirtualization, a software discipline trading guest kernel binary compatibility for reduced dependency on costly trap-and-emulate techniques. Deterministic replay constitutes consistent recovery of system state by reexecuting an original computation from an event log.

The Xen hypervisor, as one major contender of the paravirtualization paradigm, features an abstract I/O model mapped to shared memory and a small set of inter-VM communication primitives. Demonstrably, this contributes much to a balance between a compact machine interface, thereby low overhead, and full application compatibility. However, Xen's overall architecture also promotes a shift from monolithic VMMs to elements borrowing from classic microkernel design.

A primary contribution is evidence that trace capturing under a piecewise deterministic execution model can be added to an abstract but inherently asynchronous, split I/O model for virtual machines without compromising a typical run-time overhead of about 5% reported with monolithic systems. Furthermore, integration into a commodity monitor does not thwart original system design, which suggests that the approach taken remains generally defendable under maintenance cost considerations as well.

Acknowledgments

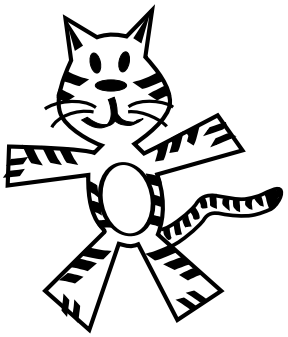
This thesis was written under the careful supervision of Prof. Dr. Arndt Bode and Dr. Carsten Trinitis. Turning, with an original background in operating systems software research, to availability at the application level, then finally back to system software could not have been done without the support and freedom I received.

I am honored to receive expert's second opinions. Thanks go to Prof. Dr. Uwe Baumgarten for kindly devoting his time and expertise as part of my grading committee.

The Department of Informatics at TU München provided an excellent working environment, without which I doubt that any of my ideas would have come to fruition. This includes Hubert Eichner, Toby Klug, Michael Ott, Max Walter and Josef Weidendorfer at LRR. Thank you all, for occasional coffee, listening and frequent helpful suggestions.

I owe the deepest gratitude to my wife, Xenia, who encouraged and supported me so much over the years.

Last but not least, special thanks go to Jonas, our son, for reminding me ever so often that any OK book needs to have a proper tiger in it.



Contents

List of Figures

1 Introduction

Around 2004 to 2005, I was working on project *Balance*, an academic/industrial research cooperation on high-availability (HA) middleware designs for commodity system platforms. HA *middleware* systems are software frameworks for high-level application development in languages such as C/C++ and Java. The term *commodity* in such systems refers to clustered computer architectures built from comparatively inexpensive, industry-standard processors and I/O components, most notably a shift from switched circuits to packet-switched internetworking. The motivation behind software availability frameworks is that while building highly available applications at any level of a layered system architecture can hardly ever be considered trivial, application programmers willing to let their programs conform to a standardized, component-oriented system design can at least be relieved from dealing with low level details of distributed failure detection and control over redundant software component instances.

In hindsight, considering technical properties and perceivable market adoption, there were multiple lessons to be learned. One is that while such frameworks may reduce the overall cost of developing or porting applications, significant effort remains left in doing so. Another simple truth is that while unwanted system outages represent a widespread phenomenon in everyday computing practice, the development process of few projects today incorporates costly countermeasures at the software level, unless availability considerations by users and customers explicitly require to do so.

This is one reason why *transparent* mechanisms, i.e. those which do not require adaption on the side of system architects, are so attractive. Transparent mechanisms basically derive from two concepts: (a) Execution of computer systems comprises a sequence of states, and (b) fault-tolerance is generally achieved through redundancy. Conceptually, fault-tolerance can be reduced, to some degree, to the problem of replicating system state on independent machines.

Transparent checkpointing exploits this concept by replicating system state without introspection of what distinguishes the state of a *service*, such as the contents of a database, from checkpoints of overall processing state, which comprises an entire program context to roll back and resume from [29]. There are

two common problems which prevent above idea from widespread application on *arbitrary* (software) systems. One is bandwidth consumption: system state can be large, and peak system memory bandwidth typically exceeds network or storage bandwidth by orders of magnitude [22]. Active replication is one technique known to overcome the first issue to some degree. Instead of passive state transfers, system *execution* is replicated, regenerating redundant state from. In distributed, message-passing systems, information guiding execution may be regarded as limited to only sequences of input messages processed. *Deterministic replay* carries the idea of active replication over to a more general system model which, in theory, could be applied to any of the software execution environments common today.

Another issue is consistent recovery of system workloads within an execution environment matching the original one. The basic workload unit managed by users today are operating system processes, or groups thereof. While processes thereby appear as a viable unit of state replication, this is unfortunately not the case. The reason is that while raw software state can always be relocated, in practice it is never self-contained. The rich and complex variety of resource interfaces provided by common operating systems makes this task difficult. More importantly, it will require complex changes to any operating system hosting these environments [29, 27].

System virtualization techniques are almost as old as time-sharing computer systems, but saw largely renewed interest in recent years. Like operating systems, virtual machines (VMs) ultimately host application software. Different from customary OS kernels, workload units carried by system VMs are whole operating systems and processes, multiplexed on a single physical machine instance. Sole domain of expensive mainframe architectures in the past, their renaissance is due to the emerge of effective hardware and software virtualization techniques applicable to popular processors and system architectures. As with the HA architectures mentioned above, another round of technological commoditization emerges.

Until today, the driving forces behind the ongoing adoption of system virtualization remain system consolidation, improved resource utilization and reduced maintenance cost. To ease system maintenance, some server-class system virtualization products soon featured VM *migration*, with remarkable performance and apparent ease of implementation. In contrast, almost 20 years of prior research in process migration did not yield much impact on any commodity server operating system. The feature demonstrates an important difference to the software environment facilitated by customary operating systems: transparent, fast workload relocatability for host systems sharing a common network segment.

Recovery for highly available system architectures shares similar properties.

Beyond the design of highly-available systems, replayable VM execution has a considerable number of additional potential applications. Starting from an initial system state, replay can recover any intermediate state a given system traversed in a preceding run. Known applications of such facilities include:

- System debugging, such as running applications to a point where an original computation failed. Inspection of not only failure state, but the entire computation which produced it, provides valuable means for failure analysis [20].
- Improved system profiling. Collection of run-time information on systems is a valuable tool in system design and research. However, specific results are often not repeatable if the execution is inherently non-deterministic. Furthermore, capturing detailed system information has usually a notable performance impact and may itself lead to distortion of original phenomenons encountered. Capturing only non-determinism guiding an execution alone may conserve a particular system run with comparatively low distortion. Deterministic replay then makes the task of detailed analysis repeatable [97].
- Enhanced system security. Similar to profiling and debugging, more recent research revealed the benefits of deterministic replay for system security measures [28]. Future system intrusion detection and analysis tool sets might make use of deterministic replay. Similar to debugging, returning to an uncompromised system state, then replaying execution to a point where compromisation took place can provide valuable insight in how attacks were arranged and how they succeeded.

Various virtualization architectures presently exist. Anticipating that the service relocatability achievable with system virtual machines opens a wider spectrum of future applications, this thesis focuses on deterministic replay for one specific virtualization class, called *paravirtualization*. The motivations for this specific virtual machine families is summarized in the following section.

1.1 Motivation of the Thesis

System virtual machines allow for multiple guest operating systems to share a common hardware platform. They are implemented by virtual machine monitors (VMMs) mapping the execution of a guest system to underlying (real) machine hardware. The mapping is maintained through control over a critical subset of

the instruction set executable by guest systems. Similarly, deterministic replay requires a monitoring facility achieving similar control over a particular ISA subset. Demonstrably, these subsets are not necessarily equal, but in practice overlap to a considerable degree. In order to reduce the development effort spent on a suitable monitor, available VMMs may be extended.

Seeking practical applications, system size and respective overhead needs consideration. What users or system developers on a single machine ultimately expect is *application* support and compatibility. Typical applications comprise only one or a limited number of processes. Compared to processes, whole systems as the future standard unit of repeated execution are considerably more than originally asked for. Moreover, the raw machine interface is comparatively complex. A native operating system is more heavyweight, and takes more cycles to initialize than just the number of processes it carries [13].

Essentially, deterministic replay facilities log information about occurring *non-determinism*, which are system state changes originating externally from the software system executed. This log may then be replayed on alternate instances. To achieve consistent logging and replayability, the structural complexity of the system's environment dictates the amount of change imposed on any existing implementation of that environment, such as a VMM. Given that an existing monitor is to be modified, not only technical feasibility, but *maintainability*, i.e. purely economical considerations, are of practical concern.

In short, the ideal guest system running inside a virtual machine would be small. It would be supported therein if its operating environment were simply structured, as would be determination of environment interactions guiding guest execution. Such properties need not sacrifice feature completeness and an application interface compatible to those of standard operating systems. Changes to a guest kernel, to aid the task at hand, require cooperation on the side of system vendors, but the change appears justifiable.

Although for different reasons, such properties closely match some of the motivations behind system *paravirtualization*. Paravirtualization is a virtualization technique which, different from classic virtualization, breaks with raw machine interface compatibility at the guest system interface. Beyond performance considerations, one reason to do so used to be lack of proper virtualizability of some popular processor architectures, such as the ones based on the prevalent IA-32 and derived instruction sets [69]. In order to virtualize such systems efficiently, paravirtual guest systems will not rely on an original system instruction subset, but request VMM assistance for privileged operations instead. This yields comparatively low overhead due to less dependency on low-level instruction em-

ulation. Similarly, raw I/O device interfaces can be replaced with abstractions at a significantly higher level.

Variants of system paravirtualization are presently implemented by a small number of open and commercially available products. One is the Xen hypervisor originally developed at the University of Cambridge [13]. This thesis evaluates deterministic replay for the paravirtual machine environment as implemented by Xen, recognizing differences to alternative implementations of basic machine paravirtualization where appropriate.

1.2 Thesis and Contribution

In summary, the contributions of this work are

- Architecture and implementation of an extension to a paravirtual machine monitor to efficiently capture non-determinism encountered by paravirtual machine guest systems.

  The system replays uniprocessor guest operating systems controlled by a monitoring facility integrated in a fashion scalable on modern multiprocessor and multicore processor architectures.

  It demonstrates that some specific properties of the Xen VMM, most notably mapping of machine resources to data structures maintained in shared memory, can greatly reduce unwanted interference of such extensions to an existing VMM implementation.

- Experimental evaluation of the performance impact on workloads executed by virtual machines, derived from above implementation.

  An overall slowdown as low as 0.65% for CPU-intensive programs such as the SPEC CINT2006 benchmark suite has been measured.

- Experimental evaluation of bandwidth of determinant log capturing for typical I/O and CPU-intensive workloads.

- Directions on the practical use of performance monitoring facilities available on present x86 family processors for the purpose of replaying asynchronous events on virtual machines (or processes).

Implementation techniques and results presented here were largely inspired by the architecture of the Xen hypervisor, due to the paravirtualization technique it featured. With a background in high-availability system design, the project was called Xen/VLS (*Virtual Lock-Step* or *Virtual Logical Synchronization*). The idea shall be summarized as follows:

> System virtual machines are the most practical solution to achieve deterministic replay for present commodity system applications. Therein, paravirtual machines perform best.

Retrospectively, meaning of the term *performance* is at least twofold. One obvious performance implication to evaluate is relative performance downgrade to target guest systems. Relative performance denotes time spent by the system while capturing event logs, performing a specific task, in comparison to the same task in an unmodified environment. Additionally, bandwidth requirements play a significant role.

Performing "best" requires comparable numbers. Deterministic replay with system VMs was first researched in 1996, back then in order to achieve highly-available guest instance pairs in distributed systems, i.e. semi-active replication. HA-applications, however, come at additional cost, due to its dependency on replica consistency, which adds additional latencies. Full semi-active replication of presently customary network interconnects has not been implemented as part of this thesis. Instead, most of the results presented here have been focusing on proper execution trace generation. The difference will be more thoroughly explained in chapter 2. However, some applications for deterministic replay rather depend on efficient tracing than subsequent replay, or a simultaneous combination thereof [97]. Furthermore, overall performance downgrade in semi-active replication is largely beyond log generation at the system core.

As outlined as part of the motivation, part of how a particular VMM extension "performs" in practice is not determined by slowdown, but rather simplicity. Compared to the present overall popularity of system virtualization, presently known applications of deterministic replay exist in niche markets. If a respective extension is too intrusive to the VMM implementation, or hampers development of future revisions, it is not maintainable under economic considerations. Within the prototype developed here, integration into the VMM ultimately needs not clash with original system design. Elements implementing the original virtualization layer remained largely unchanged, which suggests that the general approach taken remains defendable under above maintenance cost considerations as well. This is not only despite the issues outlined above. In order to achieve proper event logging, changes to the VMM are inevitable. But shared memory as the underlying communications primitive contributed as much to a device-independent I/O model for trace capturing and replay as it presently does for the Xen's core virtualization layer.

Regarding system performance, it turns out that the VMM architecture examined throughout this document has both advantages and disadvantages. Xen's

overall virtualization environment differs from traditional virtualization architectures. Is is based on two simple primitives: shared memory and an abstract event notification mechanism. On the one hand, this contributed much to a comparatively simple extension mechanism to achieve event logging. On the other hand, virtualization of I/O devices in Xen is not performed by the VMM, but one or a small number of privileged guest systems. The overall virtualization environment promotes asynchronous communication with isolated subsystems running in separate addresses spaces. Arguably, the architecture shares some similarities with common microkernel design. Xen's decomposed I/O architecture makes thereby extensions for proper trace capturing non-obvious. Suitable event logs need to reflect state changes at the granularity of single guest instructions in order to achieve full determinism during replay, where I/O virtualization commits device status asynchronously and across separate address spaces.

Core of the extended VMM architecture presented here mainly consists of a device abstraction inspired by customary physical I/O architectures. In real machines, asynchronous memory accesses originating from peripheral system components are typically performed by DMA (*Direct Memory Access*) engines. The same abstraction, as a software-implemented pseudo-device embedded into the machine virtualization layer, can be effectively and efficiently employed to make formerly arbitrary memory access interleaving consistently replayable. The resulting memory access paradigm has been called SMA (*Synchronous Memory Access*). At the source code level, some critical parts of it could be integrated almost transparently into intercommunicating guest systems.

Traditional system VMMs are rather monolithic in design. Recently published results derived for full virtualization on a monolithic virtualization layer reported a performance downgrade of approximately 5% experienced when tracing compute-intensive tasks. While some of the techniques employed are not necessarily fully comparable yet, the results achieved with Xen have been encouraging. On Xen/VLS, run time overhead as low as 0.65% for above CPU-intensive workloads has been measured.

1.3 Organization of the Thesis

Chapter 2 will provide a overview of systems and methods upon which the thesis builds. This includes present trends in computer architecture. Additionally, theory of consistent state replication in fault-tolerant system design will be introduced, commonly referred to as the *state-machine approach*, or variants thereof. Finally, the merits of contemporary virtual machine technology as the

basis for applied state machine replication will be investigated.

The following two chapters turn to system virtual machines in more detail. Chapter 3 introduces basic techniques to efficiently virtualize today's commodity systems. Present paravirtualization techniques, as well as it's individual advantages and disadvantages, is a general technique closer examined. As noted above the specific virtualization architecture pursued throughout this document is not only determined by the paravirtual machine paradigm, but the Xen hypervisor as one specific representative. Xen will be discussed in chapter 4.

Chapter 5 will then describe methods to achieve determinism in monitored paravirtual machine execution, closing with a description of the resulting log format and an evaluation of event frequencies and bandwidth, as well as overall performance impact measured.

A small, but nonetheless important technical detail is consistent replay of asynchronous events. At the bare machine interface, these are typically device interrupts. Efficient replay of those is best performed when aided by dedicated facilities in processor hardware. Unfortunately, few processor architectures in computing history implemented such a feature. The ubiquitous x86 platform on which Xen, the VLS prototype and thereby this thesis build is no exception to this problem. Many contemporary x86 family processors implement a performance monitoring facility, which can be used for this purpose. This will be described in chapter 6.

The thesis will conclude with chapter 7, summarizing on overall results and experience gained. Finally, areas for future research will be suggested.

2 Background

The introductory material presented throughout this chapter is structured as follows. This thesis deals with techniques for the effective replication of software system instances as they are executed on physical machines or, simultaneously, a number thereof. Section 2.1 will touch on past and ongoing trends in the architectures of contemporary computer systems. One are improved multiprocessing capabilities in contemporary processor design, from which scalable architectures for the monitoring facility should benefit. Another is the potential of commodity components and cluster architectures in high-availability system design.

Built upon such systems, the underlying mode of replica execution requires careful inspection of control and data flow during input processing of each such instance. This functionality is commonly implemented as a monitoring facility between the target system and its surrounding execution environment. Section 2.2 introduces monitoring and event logging to execute replicas consitently in distributed architectures.

Finally, section 2.3 will turn to machine virtualization. Similar to the above facility for monitoring replica execution, virtual machines comprise a monitor to achieve resource sharing between a number of guest systems. The remainder of this document mainly considers integration of replica monitoring into multiprocessing-capable virtual machine monitors.

2.1 Real Machines

2.1.1 Machine Architecture

A general purpose computer system comprises a number of resources, divisible into three main types of components: (1) A processor, or a number thereof, executing instructions located in main memory (2) main machine memory, randomly accessible in a single global address space shared by all processors. (3) A number of devices for peripheral data input and output (I/O) [86].

Depending on the number of processors they contain, computers can be classified into uniprocessor (UP) or multiprocessor (MP) systems. Multiprocessor

systems became an option with the advent of multitasking operating systems. Processors in such systems are tightly coupled, managing a number of program instances simultaneously. Program instances at the machine level are processes and threads, which perform sequences of instructions, sharing resources on a common hardware platform.

Symmetric multiprocessing (SMP) machines are built from multiple physical processors with identical instruction sets. Symmetry among processors implies that all processors serve identical purposes, but execute separate threads of machine instructions in parallel. In preemptive multitasking systems, assignment of individual processors to any given thread of execution is largely transparent at the application level.

Microcomputer performance experienced sustained and rapid growth over the past three decades. Moore's law predicted exponential growth in chip complexity over time, and has been confirmed since its invention 1965 [62]. Likening complexity to processing power, performance gains used to evolve accordingly. Continuous gains in computing power, however, come at the cost of tremendous investments in research, engineering and component manufacturing.

With the advent of multiprocessing systems, memory access became a primary concern. Notwithstanding its programming model as a single shared entity, if memory is implemented as such (*Uniform Memory Access*, UMA), bandwidth consumption at the memory interface becomes a major bottleneck, which in turn accounts for limitations in the overall scalability SMP systems can achieve [9]. NUMA (*Non-Uniform Memory Access*, [50]) distributes physical memory among a number of processors, while maintaining the shared memory paradigm at the processor ISA.

Past advances in processor manufacturing were dedicated to individual performance gains in the single thread, satisfying perpetual demand for more data bandwidth and faster speed in sequential processing. The solutions were higher clock speeds, larger caches and implicit concurrency in the sequential execution model. Instruction-level parallelism (ILP) expands performance of sequential programs, but cannot advance beyond a point where ultimately correctness would be at risk. Higher clock speeds come at the cost of increased power consumption and heat dissipation. For years to come, only caching increases will prevail. Single-threaded performance will not sustain its past rate of growth.

2.1.2 Microprocessor Trends

Starting from 2001, the evolution of microprocessors saw a fundamental turn in orientation regarding the question whether, and how, performance improvement in future processor generations are to be maintained. While the question is not whether Moore's law continues to apply, future increases in chip complexity would not map to single-threaded application performance and additional consumer value.

The major shift of focus is an overall shift ILP to thread-level parallelism (TLP). Chip-level multiprocessor (CMP) architectures combine two or more processor cores of one and the same architecture on a single integrated circuit. While the functional interface equals that of symmetric systems, physical colocation of inter-processor communication and the integration of common resources such as the memory and caching hierarchy provide additional performance improvements. Different from ILP, however, thread-level parallelism in single processors does not benefit from a larger number of cores without assistance on the side of application programmers. Multithreaded application development and tuning can be a time consuming and costly task, but within the foreseeable future a necessity to utilize increases in processor bandwidth appropriately.

Shared memory multiprocessor systems represent a commodization from high-performance computer to everyday workstation computer usage. A second major change in is virtualization of instruction set architectures, memory and I/O interfaces in machine hardware. While the motivations are various, one of the major issues the software industry is presently facing is that classic programming models are as sequential as the original machine architecture they derived from. Beyond the high-performance and high-end server market segment, many systems remain underutilized. A turn to virtualization in the industry is in part due to the shift to chip multiprocessing. One of the major issues the software industry is presently facing is that classic programming models are as sequential as the original machine architecture they derived from. Beyond the high-performance and high-end server market segment, many systems remain underutilized. Consolidation of multiple operating system instances and, presumably single-threaded, application programs may reduce overall system and maintenance cost [31].

2.1.3 High-Availability and Fault-Tolerance

Availability is the capability of a system to provide a given service. The concept of a service is arbitrary. Services may be interactive ones, i.e. provided to users,

or provided among individual system components, each being a subsystem of an overall more complex system.

An ideal system would be available at any time during its scheduled times of operation. Time of availability is referred to as *uptime*, while scheduled times of unavailability are usually referred to as *planned downtime*. Unfortunately, systems are generally not ideal, but prone to failures. With general-purpose computer architectures, faults in hardware or software may occur, as do externally induced failures (such as power outages). Either case may lead to *unplanned downtime*. In some environments a given level of service during failure-free operation may be reduced to some degree during recovery, such as performance, but service continuity must be maintained, with reasonably high probability.

Once failures are considered, repair must be as well. Both uptime and unplanned downtime become transient states. Two variables are often found across the relevant literature: *mean time to failure* (MTTF) and *mean time to repair/recovery* (MTTR), both expectancy values of uptime and unplanned downtime in a probabilistic model [84]. The availability A of any given system is thereby defined as follows:

$$A = \frac{\text{MTTF}}{\text{MTBF}}$$

Availability can therefore be quantified in terms of uptime per scheduled time of operation. The value often specified as percentage; in many fields where unconstrained availability is of major concern, such as life-critical services (e.g. telecommunication, medical equipment) availability of 99.999% (dubbed "five-nines") is an often-quoted requirement. With continuous services, where even the concept of planned downtime is not accepted, an alternative expression is *downtime per year*. For five-nines availability, expected downtime would evaluate to 315.36 seconds per year, about five minutes.

High-availability (HA) is an expression for a considerably large availability, generally at a level which cannot be met by interactive repair, such as five-nines. Under the assumption that failure of individual components in a system cannot be ruled out with sufficient confidence, such systems need to adapt upon failures. *Fault-tolerance* implies that partial failures are expected, and provided for. Adaption to failures requires *redundancy* of system components, which will be revisited in section 2.2.

2.1.4 Cluster Architectures

The material presented throughout this document is targeting hardware and system design customary in present cluster architectures. A distributed system is a collection of computer *nodes* interconnected by a network. A *cluster* is distributed system in a local or system area network, cooperating for the purpose of serving common goals. A node is an otherwise autonomous computing device, comprising a number of processors. Generally, cluster computing is classified into three major branches:

Compute clusters comprise a potentially large number of interconnected processing nodes in order to increase overall system performance for computationally intensive tasks.

Load-balancing clusters distribute client requests among a number of servers. Unlike compute clusters, load-balancing clusters comprise at least two different tiers: server nodes providing actual service to clients, and routing nodes (frontends) distributing incoming client requests among servers.

High-Availability clusters comprise multiple machines to improve reliability for a service or group of services. That is, availability clustering nodes operate foremost *redundantly*, while compute clusters process jobs cooperatively.

For the purpose of this chapter, the two fundamental concepts are availability (*redundant*) and compute (*performance*) clustering. The major difference between compute clustering and load-balancing is due to the type of service provided. Compute cluster nodes cooperate on a single task issued by one or only a small number of users. Load-balancing clusters provide network services typically designed to a client-server model, for a large number of clients. This in turn accounts for different communication models. Performance gains achievable in compute clusters largely depend on the performance of network links carrying necessary node intercommunications. In contrast, client-server traffic routed in and out of the cluster accounts for most of the consumed network bandwidth in load-balancing.

Availability clustering and load-balancing are correlated in that they are sometimes combined. Since frontend nodes represent a single point of failure for a large number of clients, redundant deployment is a fundamental requirement in order to retain desired levels of availability. Conversely, multiple load-balanced servers may provide the corresponding level of redundancy without further augmentation. Many successful client-server protocols, such as HTTP, are stateless, rendering servers immediately redundant. Once a faulty machine is excluded from request forwarding by frontends, clients may regain service at any other

machine without readjustment.

2.2 State Machines

One major motivation when replication is applied to virtual machines is to run applications unmodified, i.e. *transparency*. In contrast, many fault-tolerant system designs require applications and interfaces meeting individual constraints of the overall system model, which then in turn makes recovery from failures possible. Running applications unmodified, in contrast, requires the opposite approach: an overall system model meeting the properties of commodity application environments. This requirement lends itself to a replication technique called *semi*-active replication [14], a variant of *active* replication in distributed systems, with properties which make it particularly attractive when applied to general-purpose and commodity systems.

The introductory material regarding active replication is organized as follows: All variants of active replication originally derive from state machines as the underlying processing model. This concept is introduced in section 2.2.1. Second, replicated state machines and fault recovery must meet two important requirements. One is *agreement*, on the sequence of state transitions, subject of section 2.2.2. The other is global *consistency* of recoverable state, discussed in section 2.2.3. A reference architectural model integrating agreement and replicated execution is then briefly introduced in section 2.2.4.

Section 2.2.5 turns to inherent non-determinism in general-purpose runtime environments. The fact that few real-world applications meet the state machine assumption inspired *piecewise determinism* (PWD), a reformulated execution model. It forms the basis of *deterministic replay*, i.e. event (as opposed to *message*) logging to save system state during execution for later recovery.

Different from the state machine approach, some sources of non-determinism predominant in general purpose runtime environments cannot be efficiently agreed upon. Deterministic replay imposes lesser demands on replica agreement to achieve better efficiency. This constitutes *semi-active* replication, introduced in section 2.2.6.

Deterministic replay and semi-active replication mostly differ by the concept of fault-tolerant log dissemination in a distributed system. Section 2.2.7 will discuss causal logging as a fault-tolerant log protocol, which combines maintenance of global state consistency with comparatively low latency induced by replica synchronization.

2.2.1 Deterministic Machines

The basic model underlying any variant of active replication are deterministic, finite state machines, or simply *state machines*. A state machine $M = (S, s_0, \Sigma, T, \Gamma, \omega)$ comprises:

- A finite set of states S, where the machine starts from an initial state $s_0 \in S$.
- Finite sets Σ and Γ representing input and output of the machine.
- A state transition function $T : (S, \Sigma) \rightarrow S$.
- An output function $\omega : S \rightarrow \Gamma$

A *computation* (or *execution*) of M with length n is a sequence of states $s_0 s_1 ... s_n$ in S traversed for a series of inputs $\sigma_0 \sigma_1 ... \sigma_n$ in Σ.

The machine is *deterministic* because any state transition performed by the machine is uniquely determined solely by T, that is: present state and input. Similarly, output at any point during the computation is entirely driven by present machine state. Hence, if *input* is determined, then the resulting state sequence is determined as well, as is observable output.

In the design of fault-tolerant systems, this model is applied to processes, which constitutes the so-called *state machine approach* for replication in distributed systems [73]. Different from the static model above, it is usually formulated in more convenient terms of *deterministic programs* which operate on *state variables*. Programs execute *commands* upon client *request* and generate output accordingly. Again, both state transformations and outputs are determined solely by commands and present state.

It is well recognized that this model may applied to systems implementing requests and outputs in any conceivable fashion. For subsystem decomposition in programming environments, procedure calls may be considered. In distributed systems, the most practical paradigm is the sending and receiving of messages. For virtual machines, I/O instructions will be considered.

2.2.2 Coordination

For distributed, deterministic state machines to produce the same state, all must encounter same set of requests (input). The reason why state machines are a convenient model is that requests are processed *sequentially*. In contrast, concurrent dissemination and/or physical reception of messages in distributed

systems is often arbitrary, due to the properties of the underlying communication network. *Coordination* governs agreement and ordering on the request sequence among all replicas [73]. Coordinated processing of request implies two properties.

Atomicity Any request is either performed by all (correct) replicas, or none. In other words, all correct replicas *agree* on the same set of requests.

Ordering All (correct) replicas process requests in the same (total) order.

Coordination may be performed by clients and servers or, if clients are independent of each other, by the replicated state machines themselves. Ordering of requests *may* be constrained, depending on requirements of the protocol established[1]. Stateful protocols typically assume an implied order which adheres to the intuitive concept of (potential) *causality* as observable in space and time, constituted by Lamport's more general *happens-before* ("$\rightarrow$") relationship among events in distributed processes [56]. The happens-before relation is defined as follows:

1. Processes execute sequentially. If event e_a happens before e_b at the same process, then $e_a \rightarrow e_b$.
2. Message transmission is causal. If e_a is the sending of a message and e_b its reception, then $e_a \rightarrow e_b$.
3. Causality is transitive. If $e_a \rightarrow e_b$ and $e_b \rightarrow e_c$, then $e_a \rightarrow e_c$.

Causality imposes a partial order on the sequence of events. Replica coordination, in contrast, demands a total order on requests. Intuitively, an arbitrary superset of (potential[2]) request causality:

1. Specifically: Two requests issued by the same client shall be processed in the order in which they were sent.
2. Generally, if clients intercommunicate: For any pair of requests r_1 and r_2, if $r_1 \rightarrow r_2$, then r_1 shall be processed before r_2. This order can only be determined by clients.

Mutual agreement has seen intensive research, as it is probably the most elementary problem in distributed system design. The most challenging assumptions are those considering byzantine conditions, i.e. under the assumption of

[1] Independent clients and stateless protocols are typically robust against server-side reordering of independent requests. A prominent example is HTTP in the World Wide Web.

[2] Whether *true* causality actually applies is up to data flow guiding state transformation. Causality in the *happens-before* relation adheres to *potential* causality, e.g. as derivable from external observation.

(arbitrary) process failures [73]. In order to disseminate messages to an arbitrary number of replicas under the conditions outline above, *reliable multicast* protocols play an important role. Frameworks for reliable multicast provide for agreed, ordered message to process groups with a sufficiently general transport layer interface. View-synchronous *group communication* systems augment reliable multicast with support for dynamically changing replica configurations and network partitioning [23].

2.2.3 Global State Consistency

A *global state* G of a distributed system is the aggregation of the individual states of all participating nodes. The challenge with global system state is that there is no global entity which is capable of capturing such state instantaneously. Any single node can only capture its own state, as well as its input and output events [21].

One may thereby characterize an execution of a distributed system as a series of transitions in G. A *consistent* global state is one that may occur during a correct, failure-free execution of the system. This is the case if causality is maintained, which again can be expressed in terms of Lamport's *happens-before* relation: For any two events e_0 and e_1, if $e_1 \rightarrow e_2$ and e_2 happened in G, then e_1 must also have happened in G.

Consistency in distributed systems is an important factor in a number of different fields, including of global predicates such as termination or deadlock, commonly performed on "snapshots" of global state [21], or applications in distributed debugging and garbage collection [23].

Maintenance of local causality in a single sequential process (clause 1 in the definition of the *happens-before* relation) is trivially achieved: Any synchronously taken snapshot of a single node is consistent. However, the consistency of inter-process dependencies (clause 2) is harder to maintain. Intuitively, consistency across machine boundaries demands that for a collection of states in a distributed system is to remain consistent, the following must apply: If one process's state is caused by reception of a message, then the respective sender's state must reflect its transmission [29].

In fault-tolerant system design, one method where the rules of consistency require considerable consideration is state *recovery*, i.e. restoration of system state after a fault. If only part or outdated system state is recoverable, consistency is at risk. This is commonly the case in passive replication, which spurred of

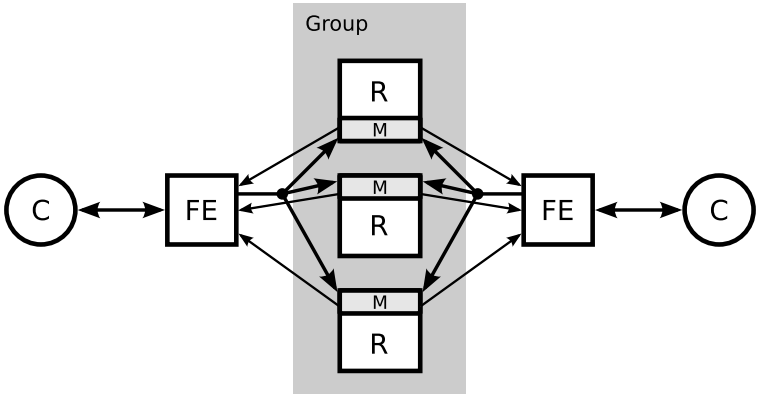


Figure 2.1: Active replication: replicas (R) with frontends (FE) and clients (C) [23].

past research in rollback recovery for complex systems, i.e. systems comprising multiple distributed, intercommunicating components. In that case, a failure in one component may cause the need to roll back respective clients [29].

With active replication, global state consistency needs special consideration when communicating with an "outside world", i.e. externally visible system behavior. This includes any remote client served, as well as peripheral devices a system is connected to. Different from an orchestrated rollback outlined above, outside clients cannot be subject to recovery procedures, hence any interaction with these systems is irreversibly manifested in global state. This in turn constitutes demand for *reliable* multicast and request atomicity: if e is the sending of a message to the outside world, then any event e' ($e' \rightarrow e$) must be recoverable by all replicas [29].

2.2.4 Active Replication

Actively replicating systems organize replicas into groups of redundant system components, forming a single *logical machine*. The overall architecture is shown in figure 2.1. Each replica acts as a state machine. System software on nodes hosting individual replica operates as a *replica manager* [23], or simply *monitor* [73]. Dedicated nodes act as a *frontends* which mediate between individual replicas and clients. Frontends fulfill several functions in one entity:

- Client access: Clients (both client components in complex fault-tolerant

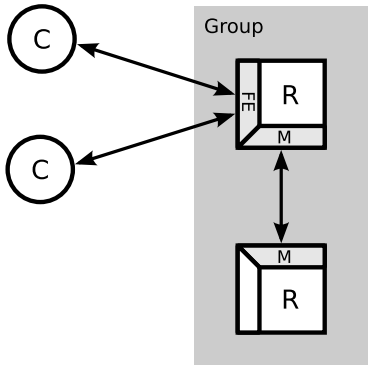


Figure 2.2: Dual redundancy in a leader/follower scheme.

systems or clients in the outside world) may remain unaware of the group topology masked by the FE.

- Dissemination of input messages: Frontends act as *transmitters* [73], forwarding input messages received from any client to all replicas in coordination, i.e. the set of requests and a total order of delivery is agreed upon. In message passing systems, reliable multicast implementations perform this task. Note that if clients are communicating with each other, ordering must reflect causality.
- Collection of output messages: Only one response is delivered to clients, duplicates which are due to replication will be discarded. Depending on the failure model, the FE may act as a voter by comparing responses received from individual replicas.

If only crash-failures are to be considered, up to t failures can be survived by deploying at least $t + 1$ replicas. Since any response is correct under this assumption, the first response collected may be forwarded to the client. Up to t byzantine failures can be tolerated by $(2t + 1)$-fold replication. In this case, the voter may accept a majority after collecting $t+1$ identical responses [23]. Given one or more dedicated nodes for input coordination and voting, three replicas are sufficient to mask up to either $t = 2$ crash faults or $t = 1$ arbitrary node failures.

A single frontend is generally considered insufficient, since it would represent a single point of failure between service provisioning by replicas and service delivery to clients. However, there is no requirement to integrate all functions described above in one single processing entity. Many systems, e.g. NSAA [16],

comprise dedicated, redundant frontend logic.

Others, such as RAPIDS [94] or the hypervisor system by Bressoud and Schneider [19], do not employ dedicated nodes to that purpose, which deserves additional consideration. Indeed, simpler topologies are possible, assuming that only crash faults matter. The role of communication with clients may be assigned to one or more replica nodes, e.g. via election. In that case, voting can be omitted, since no faulty messages can occur. Hence, output operations on the follower may be discarded by the local monitor.

Simplifying the topology further, the number of replicas may be reduced to a pair, as depicted in figure 2.2. In this case, the multicast scheme for event dissemination may be replaced to a simple ordered, reliable unicast protocol (i.e. a FIFO channel). On IP networks, a respective group communication layer can be replaced by TCP, as long as adequate failure detection is provided for.

Independent of the state replication applied, the resulting topology then resembles the general *primary/backup* fault-tolerance scheme, where a respective primary is exclusively assigned with the frontend role. In case of a failure of the primary, provisioning of the service must be taken over by the backup system. A combination takeover of both network ("IP Takeover") and block storage I/O interfaces was published in [17], building a highly-available file server.

2.2.5 Facing Non-Determinism

The state machine approach works best if applied to guide program design. Any component of a fault-tolerant system subject to replication must qualify as a state machine. This is of no concern when building systems from ground up accordingly: anything that can be structured in terms of procedures and procedure calls can also be structured using state machines [73].

However, it is often considered insufficient if either programs or, ultimately, the processes executing them are inherently *non-deterministic* in the state sequence they traverse. The former case of non-deterministic programs is important when seeking to replicate arbitrary applications. The latter is case of non-deterministic processes applies to a large fraction of customary execution environments.

I/O Instructions The original state machine approach demands that requests from remote processes are the only source of information affecting process state [73]. Indeed, the implied request/response scheme of operation is commonplace in client/server architectures, like database systems or web servers. However,

few real-world applications obey strict determinism in processing requests. Typical examples are queries on system time, time-sensitive operations, or true randomness, as e.g. employed in cryptographic functions, none of which behave deterministically across redundant invocations.

Considering high-level languages, many synchronous I/O primitives, e.g. those of event multiplexing (consider the UNIX `select()` call [42]) yield non-deterministic results[3].

Schneider recognized this fact and suggested careful redesign of all replicas in order to externalize all such data sources [73]. Intending to run applications unmodified, the simple assumption of processes behaving like state machines rarely applies in practice.

Concurrency Some non-determinisms derive from the basic requirement of *sequential* execution of either requests as well as individual request processing. Requests are not necessarily processed sequentially. In complex systems, cooperative threads or processes may execute requests in arbitrary order. Processing of individual requests may essentially be multithreaded and/or operating on shared memory.

Concurrent processes on shared memory may produce arbitrary interleavings of state accesses, and therefore perform different transformations on overall state under different timing conditions. Non-determinism due to thread arbitration may be induced by the physical machine. Processors in hardware multiprocessing systems are essentially behaving like distributed systems. It may as well be as part of an essentially non-deterministic software execution environment, e.g. thread scheduling on preemptive uniprocessors, which is essentially driven by time and timers.

Asynchrony Asynchronous event delivery to any software system is typically performed by an immediate control transfer to a service routine which is implemented by that system. Initiation of the control transfer at runtime is performed by the processing environment, not the system itself. The control transfer interrupts the running thread, and resumes execution after termination of the service routine.

In order to have any impact at all, service routines unilaterally need to affect global state. Consistent with the *happens-before* relation, the state transition

[3] Note that this applies at the language level. Admittedly, when considering the machine level, one may argue that non-determinism in I/O multiplexing is due to asynchrony. This is discussed below.

induced by a service routine may depend on the precise point of event delivery within the state sequence traversed by the interrupted thread. Hence, deterministic behavior among replicas can only be enforced by agreement on the precise point of delivery.

Asynchrony in I/O processing is featured by virtually all general-purpose execution environments, since it is fundamental to efficient communication with external entities[4]. Non-determinism due to asynchronous events is propagated across virtually any software layer in the affected system. At the ISA level, asynchrony is induced via *external interrupts*. POSIX-compatible environments empty *signals* as one possible mapping of asynchronous events to processes. The `select()` system call, being a synchronous source of non-determinism at the process level, is determined by asynchronous I/O events at the system level. If service routines are not involved, constructs equivalent to `select()` may be found in most high-level language environments, such as the Java Runtime Environment[5] [51].

In order to overcome these problems, a different model for the execution of such systems needs to be applied. Generally, the concept of machine *state* will prevail. But different from mere state machines, *execution* driving state transitions needs to reflect the effective complexity of the systems described.

Given finite amounts of memory, any practical machine has a finite number of states S. An execution *event* $e \in E$ is a binary relation between states in S: $e \subset S \times S$. For any event $(s_i, s_j) \in e$, write $s_i \xrightarrow{e} s_j$ (e *produces* s_j from s_i). An event may comprise any one of a number of different actions performable by the underlying machine at a given state. One example is execution of a machine instruction residing in memory. Another possible state change would be a control transfer induced externally. In shared-memory multiprocessors, the notion of an event includes changes to state variables performed externally as well. A suitable model may partition the set E of all observable events accordingly: $E = I \cup N \cup D$, where I comprises the set if instructions implemented by the processor, N externally-induced control transfers, and D the set of all externally-induced changes to in-memory machine state.

Deterministic Events An event $e \in E$ is *deterministic*, if and only if for any states s_i, s_j and s_k in S the following applies:

[4] One alternative is *polling*, i.e. frequent synchronous inquiries on the status of such entities. Polling is not an efficient alternative for events occurring at high frequencies or when latency is critical [82].

[5] Earlier versions of the JRE employed the second alternative: Threading, which trades non-determinism due to event handlers with non-determinism due to concurrent thread scheduling.

$$s_i \xrightarrow{e} s_j \wedge s_i \xrightarrow{e} s_k \Rightarrow s_j = s_k$$

That is, e is a (partial) function in S, as opposed to a relation. Informally, deterministic instructions are those which are guaranteed to produce the same result in machine state whenever issued from the same original state.

Events due to instruction execution will be termed *synchronous*, i.e. occurrence of such an event is predetermined from present machine state, but may be preempted by asynchronous events. The complementary set of asynchronous events $N \cup D$ must be monitored for the reasons outlined above.

In fault-tolerant, distributed system design, extension of the state machine approach to include the notion of non-deterministic events lead to postulation of the so-called *piecewise deterministic execution* (PWD) model, attributed to Strom and Yemini [29, 77]. Typically described only informally, it assumes that a suitable monitor must be enabled to intercept and control any execution event whose effect on machine state is not determined from original state. In message passing systems conforming to the state machine model, the PWD assumption is trivially met by controlling message delivery to replicas. For the extended machine model outlined above, this includes synchronous, but non-deterministic, as well as asynchronous events.

2.2.6 Semi-active Replication

The attractiveness of the state machine approach is due to the fact that only events determining the transformation of replica state need to be disseminated. This constitutes *active*, as opposed to *passive* replication (e.g. checkpointing [29, 23]), where critical application state needs to be captured and updated as a whole. Hence, actively replicated state is not captured, but the computational state sequence regenerated, by mutual agreement on any information guiding that sequence.

This is practical for message-passing systems of intercommunicating processes which satisfy the state machine assumption. The need for coordination then only pertains to messages received from clients. Agreement between two nodes according to the state machine approach is shown in figure 2.3, where total ordering on messages in performed by one or several transmitters.

Contrasting message-passing systems, piecewise deterministic execution requires coordination not only of application-level messaging. Instead, *any* non-deterministically occurring event must be agreed upon, i.e. including those

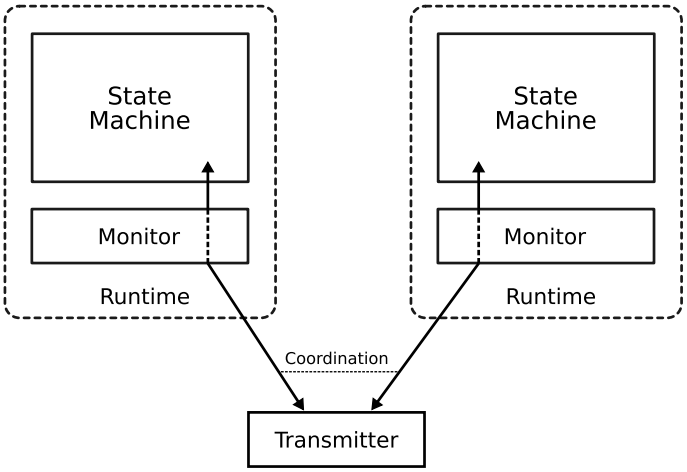


Figure 2.3: Agreement in message passing systems.

relating to the runtime environment executing a given application. The problem is that agreement on events originating from individual replicas, as opposed to a central frontend device, is beyond reliable multicast, but requires a solution to the consensus problem.

Protocols to achieve agreement on individual events are not practical if the tolerable delay in reaching an agreement approaches or even goes below achievable latency on the communication path. This is potentially the case for a large number of the event sources outlined in section 2.2.5. A prominent example are timers and thereby timeouts and the delivery of timer-driven asynchronous events.

In order to reach an agreement, their exact point in an ongoing computation must be agreed upon in order to deliver the event deterministically on all nodes. On a single node, the event would be delivered timely, for maskable events at the immediately next point during a computation where the event is not masked. Reaching agreement among distributed replicas, in contrast, takes a synchronous intercommunication sequence. Since no sufficiently synchronized clocks are available, all replicas must exchange their *prospective* points of event delivery. States reached by active replication cannot be rolled back, hence the most advanced point in the computation needs to be agreed upon. Slower nodes then need to catch up. The result is that while state machine synchrony demands that slower nodes dictate the overall speed of computational progress, delivery latency through agreement always represents the worst-case scenario.

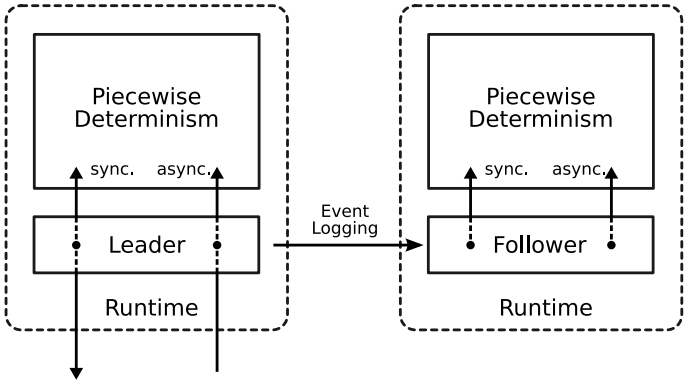


Figure 2.4: Semi-active replication replaces multilateral agreement with unilateral decisions.

Generally, for events occurring at a comparatively high frequency, e.g. interrupts delivered to a virtual machine, the cost of a consensus protocol may easily approach the frequency of events experienced during non-replicated execution, slowing down systems by orders of magnitude. Discussions of the number of messages required to reach distributed consensus and the implied performance impact are found in e.g. [12] and [94]. This excludes systems employing low-latency interconnects in dedicated hardware, such as NSAA, which aims at negative impacts in the range of 5-10% [16].

Semi-active replication [66], also called the *leader/follower* model [14], is a variant of active replication avoiding the relative cost of distributed agreement. For this purpose, the set of nodes is partitioned into one *leader* and a number of *followers*. The idea is to replace consensus, formerly achieved by participation of all nodes, with unilateral decisions about all non-determinism. In short, it is an application of deterministic replay. In semi-active replication, logging of determinants is solely performed by the leader instance. These log entries are disseminated to all follower nodes, which *replay* the original state sequence. The overall structure of this approach is shown in figure 2.4.

The difference to deterministic replay under the PWD model is that fault tolerance requires events to be recoverable in case of failures, where replay for debugging or profiling purposes has lesser requirements. Protocols maintaining consistency across replica failures are discussed in the following section.

Semi-active replication is usually proposed to render virtual machine instances fault-tolerant against crash failures, i.e. it is assumed that faulty host systems

halt execution upon occurrence of such a fault. Admittedly, byzantine conditions due to arbitrary node failures cannot be easily covered by the leader/follower model. This is due to the fact that mutual agreement on any state change induced by the processing environment is dictated by the leader processor alone. In case of a defect, the leader node may therefore propagate erroneous events to all follower nodes. Powell noted that this restriction can be relaxed to some degree, if followers check all log entries against a given range of valid ones [66]. Such checks were implemented as part of the Delta-4 architecture. Upon encountering a log entry deemed invalid, all follower nodes fall back to a common default one. Still, an erroneous *valid* event would cause state divergence to go unnoticed. Delta-4 employed external voting logic in order to correctly mask such faulty events.

2.2.7 Causal Logging

Deterministic replay on one or a number of follower instances enables state consistency among failure-free replicas, but does not guarantee that all events are recoverable if the leader fails, if processing and log transmission are performed asynchronously. Due to the fact that a follower instance is not in synchrony with the leader, failover always implies some (hopefully small) degree of rollback in computational state. This is the major difference from classical lockstepping techniques implemented in hardware (e.g. [15]). In order to recover consistent global state beyond failures, *synchronization* between leader and followers needs to take place.

Generally, synchronization is concerned with the concept of *stability* of events, respectively messages: In rollback recovery, an event is stable if it is recoverable, i.e. as soon as it is saved on stable storage [29]. With log dissemination through messaging, stability implies multilateral agreement on a respective message [73]. For a log transmission over a reliable FIFO channel to a single follower, a simple acknowledgment on the last event received will suffice.

Different approaches when to perform synchronization exist. One is to synchronize on every event, before it affects the replica computation. This constitutes *pessimistic*, or *synchronous* logging [29]. *Causal logging*, in contrast, exploits the fact that replica state only needs to be recoverable once it becomes observable. As shown in section 2.2.3, this is only the case as soon as a respective system component is communicating with its environment. Hence, to maintain global state consistency, synchronization needs to take place just before performing an output operation.

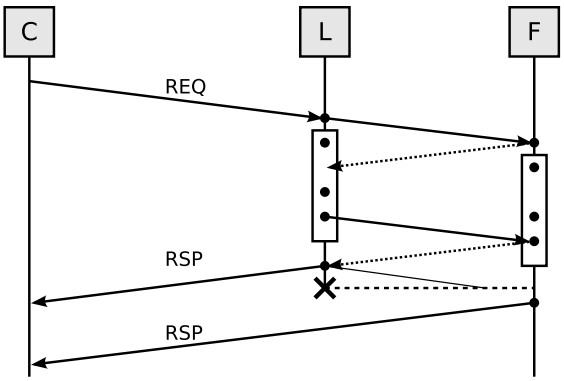


Figure 2.5: Synchronization and duplicate output with semi-active replication.

Figure 2.5 depicts causal logging[6] for only a single client and follower. Bars denote input processing by the replica instance, dots individual events to be logged, arrows indicate communication in space and time. Dotted arrows indicate a notification that an event has stabilized on the follower side. Client C sends a request `REQ` to a service. The monitor of leader node L emits a corresponding log message and delivers `REQ` to the local replica. It then continues processing `REQ`, which for the case depicted leads to execution of 3 additional non-deterministic events. Figure 2.5 shows only log transmission for the first and last event before `RSP`. `RSP` is only committed after the last non-deterministic event is known stable in F.

One benefit of causal logging is that asynchronous log dissemination can be performed parallel to replica execution. Furthermore, asynchronous transmission enables buffering and bulk transfers of events. Still, the remaining need for synchronization imposes some amount of latency imposed on all output operations committable by the leader. As depicted in figure 2.5, every event which happened before the sending of `RSP` must be waited for to become stable. The effect is called *output commit problem* [29].

Different from synchronous logging, asynchronous logging implies that output operations generated by the leader need intermediate buffering. Synchronous logging pauses replica execution and resumes only after event stabilization. Hence, output operations can be committed immediately. With asynchronous logging, output operations occur before stabilization, therefore need to be held

[6]The notation represents a mixture of UML sequence diagrams and Lamport's [56] space-time notation, inspired by Wolf [94].

back.

2.2.8 Duplicate Output

Figure 2.5 also shows the problem of duplicate output operations, which may occur when the leader instance fails. Preventing duplicates is an instance of the unsolvable two generals problem [8, 19]. Sending of `RSP` to C and a notification to L are two separate operations. The notification must be sent after `RSP`, because L may crash *before* the message is actually sent, potentially misleading F. On the other hand, L may crash *after* sending `RSP`, but before the notification is sent. If L fails and no notification is received, then whether L failed before or after sending `RSP` remains indistinguishable to F [7].

Message duplicates are typically a non-issue in IP-based intra- and internetworks. TCP/IP networking recognizes that segments not only may be lost (then re-requested) but duplicates may occur (e.g. due to gratuitous re-requests). Generally, Internet protocols cope well with duplicate messages, as do I/O device interfaces accessed on such networks, such as network attached storage.

2.2.9 Related Work

The presentation above discussed output commit and duplicate output operations in context of message-passing systems. However, the same concerns apply to operations on peripheral I/O hosted by replicas. Bressoud and Schneider found that a similar degree of tolerance applies to the SCSI protocol [19] when leader and follower host controllers connect to the same bus. In summary, they demonstrated that (1) native SCSI I/O devices tolerate reissues of the same command and (2) the SCSI protocol comprises status codes which a monitor can emulate in order to let drivers reissue outstanding commands. Property (2) can be used upon takeover by followers in order to let drivers redirect command completion notifications to the new host interface, by reissuing the original command, as property (1) suggests. In practice, network attached I/O would represent a more straightforward solution.

Much of the original work pursuing the use of deterministic replay in fault-tolerant systems focused on *log-based rollback recovery* [29]. Rollback recovery performs (transparent) uncoordinated checkpointing (i.e. *passive* replication) of application state to stable storage, at regular intervals. *Rollback* is implied by recovering from failures by restoring saved, but potentially outdated state. The

[7] Note that the same problem applies to frontend nodes once they are subject to failures.

problem is maintenance of consistent global state across distributed system components upon recovery from a failure, leading to a so-called *domino effect* where multiple components need to roll back to remain globally consistent. *Log-based* rollback recovery combines checkpointing with deterministic replay in order to recreate pre-failure state up to the point where the original computation failed. These works lead to postulation of the piecewise deterministic execution model [29, 77].

Semi-active replication was pioneered by the *Extra-Performance Architecture* (XPA), part of the Delta-4 project [14, 66]. Delta-4 aimed at replication of arbitrary processes on a dedicated host-system architecture interconnected by a reliable multipoint communication system. Network interfaces are implemented by specialized interface processors and performed agreement and voting. Delta-4 combined both passive and active replication in a selectable fashion. It differed from the work described here in that the system required both dedicated hardware as well as application support in the form of a software development framework.

The HP NonStop Advanced Architecture (NSAA) [16] is the successor to the original NonStop system [15] developed by Tandem Computers. Tandem NonStop was a cluster of self-checked (i.e. fail-stop) processor modules, each of which comprised two processors in clock-synchronous lockstep and corresponding voting logic. The designers of NSAA name a number of present and future trends why lockstepping cannot be carried on for commodity processors [16]. One is CPU frequency scaling for power saving, another are minor variances induced from very high clock speed and multiple clock signals on a single die, which do not affect normal operation but propagate non-determinism up to the ISA level. It demonstrates that deterministic replay can close that gap.

NSAA is a clustered architecture partitioning Intel Itanium SMP servers ("slices") into logical processors. Machine memory is partitioned, each logical processor runs a separate stack of the OS kernel and applications. A logical processor thereby combines processor cores from different slices. Similar to XPA, consensus, voting and network interconnects are in dedicated logic. On the software side, NSAA represents an upgrade to the traditional NonStop kernel and support routines. It is not an adaption of a commodity system and libraries, nor does it mimic any existing operating environments. Similar to the work presented here, performance monitoring facilities are used for interrupt replay. This will be revisited in section 6.1: The most distinguishing property of NSAA is that it implements a consensus protocol on machine state in order to synchro-

nize event delivery.

In the same way it does to virtual machines, semi-active replication lends itself to language-specific runtime environments. One such effort was Wolf's RAPIDS project for distributed Ada 95 [94]. Ada programs are subdivided into partitions, which may run in separate address spaces or, in the distributed case, even on different systems. RAPIDS performed transparent replication of partitions using semi-active replication. Like with all execution environments, the difference is the level of abstraction performed. Where execution of hardware and virtual machines is driven by elements of the machine interface (e.g. instructions, interrupts or ports) replication of high-level language (HLL) program execution rather considers respective control flow facilities at the language level: thread scheduling, standard I/O facilities and I/O multiplexing.

2.3 Virtual Machines

Over the years, the term *virtual machine* has been picked up for a number of fundamentally different purposes. The original purpose used to be time-sharing between multiple operating system instances on the same physical machine. Today, another widespread application of virtual machine architecture is in the design of compilers and accompanying software run-time environments, e.g. as the well-known Java Virtual Machine [75]. Nonetheless, the pervasiveness is not due to dilution of an original meaning, but justified by a proper definition of the two terms contained: *Virtualization* and *Machines*. The latter will be covered in the following section, the former in section 2.3.2.

2.3.1 Machines

Picking up the state machine paradigm outlined in section 2.2, a *machine* comprises memory, the capability to perform input and output operations to and from peripheral resources, and the capability to process in-memory state according to a software system it hosts. This rather broad concept of a machine can be applied not only to the bare hardware/software interface, but instead recurs at different levels of system software stacks. Figure 2.6 depicts a typically layered system architecture and two interfaces of major concern. The physical boundary between hardware and software is defined by the *instruction set architecture* (ISA). The ISA is partitioned into two instruction subsets: a *privileged* one and a *non-privileged* one. The non-privileged subset represents

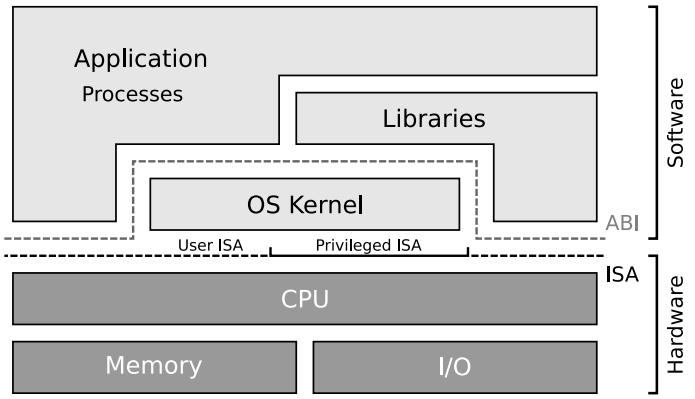


Figure 2.6: Machine decomposition and system interfaces

the *user* ISA available to all application processes. The privileged one incorporates access to privileged processor state as well as access to machine memory and I/O subsystems, accessible to software running at sufficiently high privilege levels exclusively.

Customary systems reserve access to privileged machine state to an operating system kernel. The interface exposed to application programs and libraries constitutes the *application binary interface* (ABI), which executes only non-privileged instructions on bare hardware. Privileged operations are performed indirectly, via a surrogate *system call interface*. Requests to the operating system are thereby subject to interpretation and validation by the callee. This constitutes the understanding of the system ABI as an *extended* machine interface [35], nonetheless thereby an instruction set architecture of its own.

2.3.2 Virtualization

A formal definition of virtual machine construction was developed by Popek and Goldberg [65]. It states *Virtualization* establishes by a one-one homomorphism (a *monomorphism*) from a (virtual) *guest* system to a (real, i.e. physical) *host* system.

Homomorphisms are maps on algebraic systems of sets and operations, preserving the original structure to which they are applied, as depicted in figure 2.7. Applied to computer systems, a algebraic structure suitably describing guest systems was introduced in section 2.2.5: Call the finite set of all guest states S. Execution of the guest systems is performed through some sequence

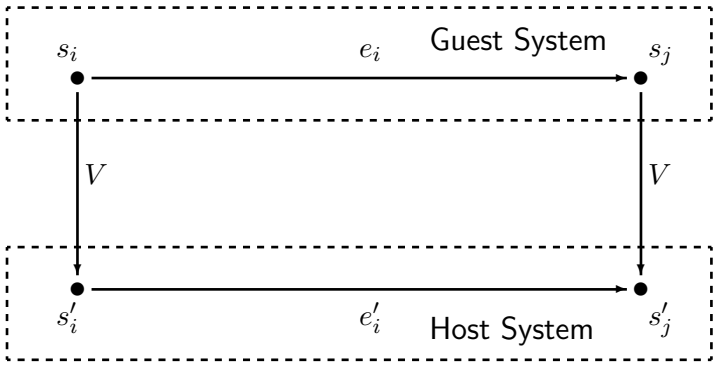


Figure 2.7: The machine virtualization homomorphism [65].

of operations $e = e_0 e_1 \dots e_{n-1}$ in E on guest system state. The *virtual machine map* $V : S \to S'$ maps original guest system state S to real system state S' of a respective host.

Note that practical virtualization of a machine interface does not necessitate the construction of V. While V must exist in the mathematical sense, it is in practice established by mapping guest system *execution*: For any e, the host system will execute a sequence $e' = e'_0 e'_1 \dots e'_{m-1}$ in E' on the hosting processor. The homomorphic nature of the virtual machine map entails *equivalence* of the guest's representation: For any two guest states and execution events s_i and $s_j = e(s_i)$, we have

$$V(e(s_i)) = e'(V(s_i)).$$

Virtual machines are established by *interface* equivalence, offering a fair degree of flexibility between interface and implementation. A *virtual machine* (VM) is a pure logical construct, which the runtime environment establishes in order to execute a guest system. The concrete entity in virtualization is implemented as a software layer on the hosting physical system: A *virtual machine monitor* (VMM) is a system software entity guiding the execution of one or a number of virtual machines. Its implementation is also referred to as the *control program*.

The large number of different applications of machine virtualization is partly due to a multitude of candidate interfaces, the important ones of which have been introduced in section 2.3.1: ABI virtualization constitutes *process* VMs, while ISA virtualization constitutes *system* VMs. A typical process VMM may be realized as a simple application program. By the same definition of vir-

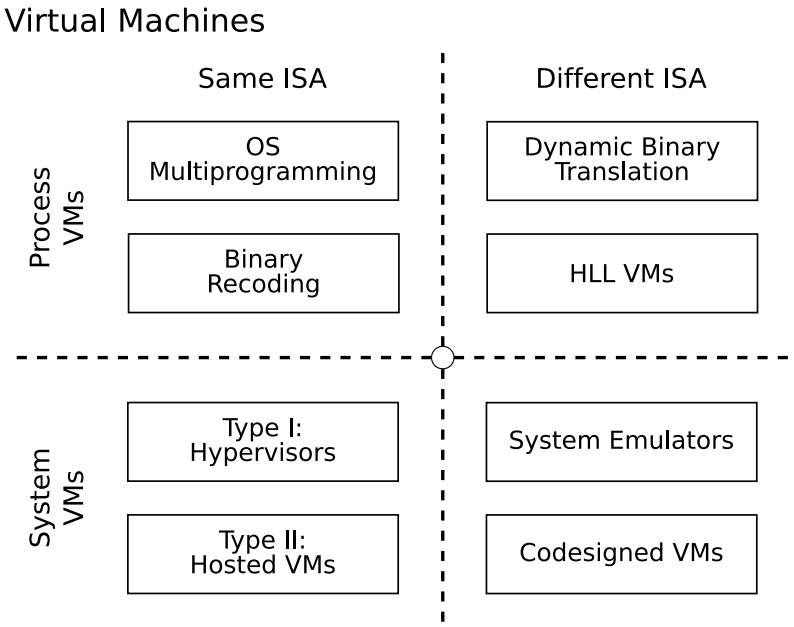


Figure 2.8: A virtualization taxonomy.

tualization given above, processes themselves qualify as virtual machines: The surrogate interface established by the operating system establishes an abstract virtual machine map. System virtual machines, in contrast, host entire operating systems, including a respective set of applications as processes on top of them. A second dimension applicable to both process and system VMs differentiates between the correspondence of potentially differing instruction set architectures (ISAs) above and below a given virtualization layer. A guest system may be designed for an instruction set architecture different from that of a host system executing it (*different-ISA* VMs [75]).

As a matter of principle, most contemporary virtual machine technology can be classified within these two dimensions: Figure 2.8 depicts a taxonomy of various virtualization techniques, originally derived from [75]. For the purpose of this thesis, same-ISA system virtual machines are of primary interest, discussed in the following section.

2.3.3 System Virtualization

System virtual machines host entire operating system instances, or a multitude thereof, independent of type and version. Different from process ABIs, they

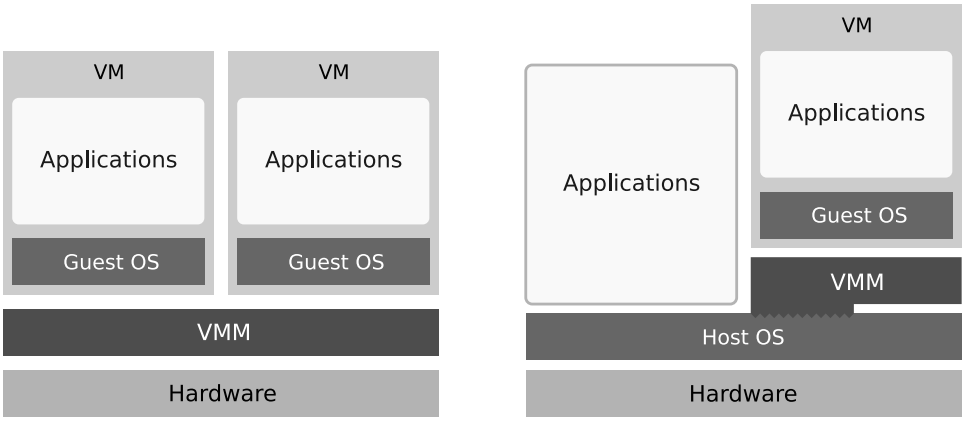


Figure 2.9: Type I (Hypervisors) and Type II (Hosted) VMMs

typically expose an interface to guest systems identical to an original machine interface they were written for [2]. Similar, "full" I/O virtualization emulates device interfaces in support of native I/O drivers in guest operating systems. Earliest same-ISA system VMs were developed by IBM in the late 1960's [2], back then in order to overcome some of the shortcomings of multiprogrammed operating systems at that time. Before the emerge of affordable microcomputers lowered the cost barrier to separate machines within one organization, virtualization offered time-sharing for portability of applications written for different operating systems and versions, sharing a single physical host [35].

The taxonomy depicted in figure 2.8 contrasts same-ISA system VMs with whole-system emulation, as would be the case if the instruction set architecture of the guest system does not comply with the hosting machine. Same-ISA VMs features two classes, denoted as *Type I* and *Type II* VMMs [34]. Figure 2.9 depicts the difference between the two. Type I VMMs are called "classic" virtual machine monitors [75], or *hypervisors*. Conceptually, they take the place of an original operating system, as a software layer between any guest operating system and the machine hardware. Type II VMMs are not layered on bare hardware. Instead, the underlying machine is controlled by a regular host operating system. Conceptually, the VMM is running on top of the host system. As the system model depicted in figure 2.9 shows, practical interfacing between the host OS and VMM is is not limited to the process ABI exclusively. While a large fraction of system virtualization can be performed in process context, part of the VMM's responsibility, such as privileged operations involved in task switching, are typically not covered by the system call interface. Instead, hosted designs

are typically *dual-mode* VMMs, which comprise one or more OS processes and a supplemental kernel space component.

2.3.4 ISA Virtualizability

System software is typically designed to run at highest processor privilege levels, in order to access and govern original hardware resources. In order to virtualize a system, control over machine resources must be ultimately reserved to the VMM.

Popek and Goldberg's virtualization model model was geared towards Type I VMMs, but equally applies to hosted VMs. Its major contribution were formally defined requirements for effective *virtualizability* of "3rd generation" computer architectures, basically the customary processor ISAs of that time. Classic virtualization does not promote refinement of existing instruction sets. Instead, it builds upon the same protection mechanisms separating operating system and individual application state from each other [33]. "3rd generation architectures" refers to a feature set not much different from today's processor ISAs. It mainly requires memory virtualization and a protection mechanism separating privileged from non-privileged operations [25].

Section 2.3.1 outlined the basic structure of processor ISAs, separated into privileged and non-privileged classes of instructions. A basic ISA conforming to this model distinguishes application (*user*) from OS kernel (*supervisor*) mode. Virtualization implies *deprivileging* of a guest operating system kernel. A virtual machine monitor will run in supervisor mode, relegating hosted guest kernels to user mode. Intuitively, one may assume that only privileged instructions are subject to interception by a VMM.

At the the level of the bare instruction set, *virtualizability* contrasts *sensitive* with *innocuous* instructions. A sensitive instruction is one (potentially) either altering privileged processor state (such as resource access or processor mode in effect), or whose effect on (virtual) machine state depends on the present processor mode. Their first virtualization theorem expresses requirements ISA virtualizability, as follows:

ISA Virtualizability For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Privileged instructions issued by deprivileged guests generate *traps*, i.e. guest operating systems trigger control transfers to supervisor mode due to violation of machine protection policies. Since trap handling remains under control

of the VMM, this mechanism constitutes the very foundation for machine virtualization, dubbed *trap-and-emulate*: Upon faulting to the monitor, sensitive instructions are emulated by the VMM according the the virtual machine map V. This procedure thereby maintains full transparency for the system software whose execution has been interrupted. The map maintains original semantics of the machine interface, translating all operations according to V.

Popek and Goldberg expressed three properties of system VMs, of which only two are shared with different-ISA VMs. One is functional *equivalence*, as discussed in the previous section. Machine behavior in real-time, however, must be excluded from consideration. A second property is maintenance of *resource control*, i.e. full isolation of all guest systems from direct, privileged access of machine resources. There are exceptions to that scheme: some I/O resources may be directly accessed by guest systems, unless they are to be shared by different guests. Direct device access will be picked up in section 2.3.5.

The *efficiency* property distinguished same-ISA virtualization from software emulation techniques. All innocuous instructions are executable under resource control and equivalence assumptions without intervention by the VMM. The relatively high degree of efficiency is maintained because sensitive instructions account for only a small fraction of the overall stream of instructions executed on the virtual machine. Effectively, the efficiency property can be restated as: A "statistically dominant" instruction subset is to be executed directly.

Few, if any original instruction set architectures were designed with virtualizability in mind. Different from an intuitive understanding of the correlation between sensitive and privileged instructions, violation of the virtualizability rule stated in the theorem is, in fact, common. Non-privileged control sensitive operations are highly unlikely, since privilege elevation uncontrollable by supervisor-mode software would be in violation even of the system protection principles affecting operating systems on the original hardware. However, issues due to behavior sensitivity are far from uncommon. All are due to deprivileging of the operating system, thereby only affecting the OS kernel. Goldberg introduced the concept of *hybrid VMs* [33] to overcome this problem without sacrificing the efficiency property. In hybrid VMs, all instructions executed in virtual supervisor mode are interpreted in software, which still qualifies as efficient by above definition, but largely sacrifices the emulation ratio achieved with virtualizable systems.

Chapter 3 will discuss some of the predominant issues when virtualizing commodity systems, such as the popular x86 architecture, and introduce present alternative virtualization techniques with greater efficiency than hybrid VMs.

2.3.5 Machine Relocatability

Active replication of system VMs shares one assumption with a related feature: virtual machine *migration*. Namely, the assumption of *relocatability* of workloads within a distributed environment of otherwise independent machines. Migration is presently implemented by a number of server-class virtualization products, such as VMware ESX Server [63] and Xen [22]. Migration moves a running virtual machine instance to a different physical node. The same concept in turn applies at least to semi-active replication: while large parts of system operating environment visible to a replayed guest system may be a product of emulating I/O interaction from an event log, *failover*, due to loss of the leader node, must leave the system continuable on a fully operational machine interface consistent with original leader state.

This section investigates why system virtual machines provide a much more convenient machine interface for this purpose than processes do. It will therefore contrast the complexity inherent in process relocation with that in system VMs.

Process migration in distributed systems has been an active field of research throughout the 1980's and 1990's. Motivations include dynamic load distribution or administrative purposes (such as service continuity, as discussed in the previous section) [61]. For the purpose of this thesis, the idea of migration within a local or system area network and a single administrative domain shall be sufficient.

Despite large investments in research and development, twenty years of ongoing research in process migration have not generated much asset in mainstream computing. Publishing results of their work on process migration in the Sprite operating system, Douglis and Ousterhout first summarized some the of criteria ultimately deciding on the applicability of migratable processes in end-user computing environments [27]. They state that all design issues ultimately result in a trade-off between four conflicting factors [27]. Generally, the same trade-off applies to workload migration with any process virtual machine, e.g. including HLL VMs [61].

Transparency at the machine interface accounts for changes to system software imposed on developers. Full transparency enables migration of arbitrary, unmodified applications. Similarly, functional equivalence before and after migration helps to meet user expectations. Compared to motivation in fault-tolerant system design, a similar transparency property is attributed to the state machine approach in section 2.2 (p. 14).

Residual Dependencies arise if the source node maintains resources on behalf

of the running process which cannot be migrated together with it. Transparency would demand that the source continues to service the migrated process over the network. operating system will typically translate operations to message exchange with the source in order to perform this task. This is typically called *forwarding*.

Performance is a non-functional property at the machine level, but central, partly because it applies at different aspects of migration. One is run-time application performance, another is migration speed. As an example, the migration procedure may terminate more quickly at the expense of a larger number of residual dependencies (such as memory remaining on the source node) left. Residual dependencies in turn will degrade run-time performance after migration.

Complexity refers to the implementation of process migration as part of the underlying execution environment. Process migration is an auxiliary feature typically added to existing systems. In case of operating systems, kernels must be extended to extract, serialize and (remotely) restore process state accordingly. Complexity does *not* refer to the overall challenge of appropriately modifying systems accordingly. Of much greater concern is the *amount* of change required to the system, since state migration tends to affect "virtually every major piece of an operating system kernel" [27].

Returning to the notion of processes as VMs, all residual dependencies are due to parts of the source machine not within the guest system memory image, but bound to the source execution environment. This may include specific hardware components. A straightforward example for residual dependencies would be specific hardware devices only the source node can provide. Even assuming otherwise homogeneous pools of machines, interactive processes cannot take the human interface devices with them. Limiting migration to non-interactive applications, the same would apply to any open file hosted by the source node to which the migrated process maintains a handle, unless the underlying file system is networked.

I/O Resources Migratability of arbitrary machine resources depends on resource types, which can be various. Demonstrably, the number of resource types again add to implementation complexity. I/O resources types can be generalized within the machine model introduced in section 2.3.1. The resulting classification will then be used to contrast system virtual machines with operating system process migration.

The model which will be discussed here is depicted in figure 2.10. It distin-

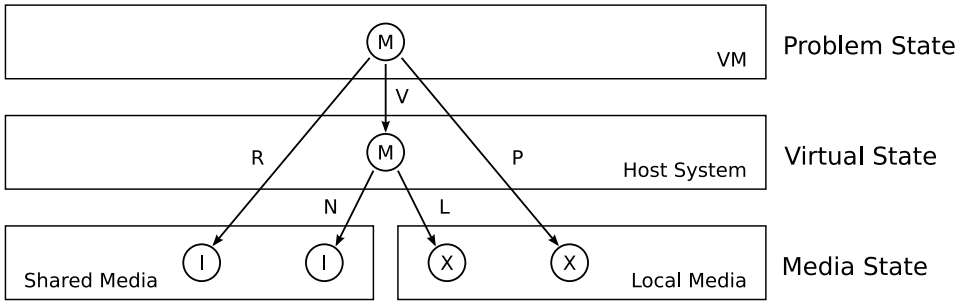


Figure 2.10: Migratability of machine resources.

guishes three types of resource states. First is VM in-memory state comprised by the virtual machine image, such as a file descriptor maintained by a standard library. Call this state *problem state* (following Goldberg [33]). All state beyond problem state is external to the virtual machine. One is *virtual state*, which represents also in-memory state, but maintained by the host system (VMM or operating system). Finally, part of the resource state may reside in hardware. An example would be frame buffer content on a graphics display adapter. Similarly, resources may be located on different hosts, accessible both from the source and destination node, such as a file on a networked file system. Both variants constitute *shared* and *local media* state. Figure 2.10 depicts an already considerable number of practical resource constellations derivable from these three layers[8]. Examples correspond to process virtualization:

- The resource may be solely in media state, directly accessed by the guest system. Call those resources *physical* (P), e.g. device state due by an I/O port driver implemented in user space.
- Call resources hosted in in-memory host system state *virtual* (V). Such a resource may be *purely* virtual or backed by physical media:
  - If the resource is backed by local media, call it *local* (L), such as a file in locally attached disk storage.
  - If the resource is backed by a shared medium, call it *networked* (N), such as a file shared by a remote file server.
- Finally, shared media may be accessed directly, typically via a message passing interface, on a transport layer insensitive to relocation within the

[8]Further refinement may consider additional layers, such as distinguishing between media state (i.e. content) and control (i.e. device interface) state at the physical layer.

surrounding node topology. Call those resources *remote* (R). An example would be remote file access performed with a client protocol implemented by a process or library.

In-memory resources are generally migratable, including those in problem and virtual state. Figure 2.10 marks those with an (M) accordingly. Shared media are advantageous. Assuming that the transport layer connecting it to a respective host system allows for a location-independent access (I), these resources do not need to be migrated. In contrast, migratability of resources comprising media state depends on an additional factors, namely *exchangability* (X). Exchangability is typically not decidable per resource class, but must be attributed per instance: A physical display device may be migrated together with a guest system, possibly moving frame buffer contents. For migration of interactive applications, however, users may equally expect the same resource to reside at the source host [27].

With regard to the original (von Neumann) machine model, processor and machine memory assigned to guest systems are virtual resources backed by local hardware. Homogeneous nodes provided, a virtual processor is trivially migrated, since the amount of CPU state is small and typically well-defined. The memory image of a guest systems may be large and therefore time-consuming to migrate in its entirety. Problem state, however, does not need inspection to be migrated, provided it is restored to its original form, and is therefore trivially dispensed with from the standpoint of implementation complexity. I/O resources, in contrast, may comprise any combination of physical, purely virtual, local, networked or remote ones. Again, *virtual* I/O resources are the most common ones, therefore discussed in more detail in the following.

Virtual I/O State Figure 2.11 depicts a number of resource classes provided by customary UNIX ABIs. These include regular files as well as special device classes, such as terminal I/O. Regular files may be local or networked ones. All these items contribute to the extended machine interface defined by the process ABI. For migration to remain transparent, each type of resource has to be individually extracted from the source host and restored on the destination host. Since such a task is uncommon, the necessary extensions to the host system must be implemented as part of the overall process migration facility [61].

There are reasons why a large fraction of overall process state resides in the virtual space. Different from system virtual machines, a primary purpose of modern operating systems is not only resource sharing but resource *abstrac-*

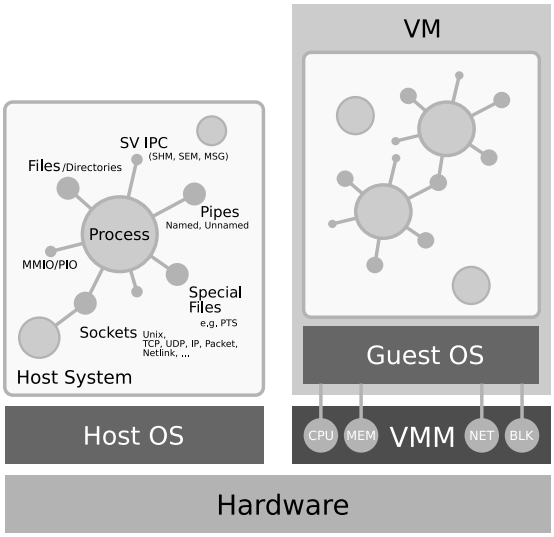


Figure 2.11: Resource complexity in OS vs. VM processing environments.

tion [82]. Formally, abstraction is a homomorphism, as is virtualization. However, abstraction is different from virtualization in that (a) the amount of interface state S_{if} on the guest side is reduced in proportion to the real interface ($|S_{\text{if}}| < |S'_{\text{if}}|$) and/or (b) the number of interface operations E_{if} is significantly smaller ($|E_{\text{if}}| < |E'_{\text{if}}|$). That is, abstract interfaces are (intentionally) of lesser complexity. State saved in problem space, however, is not eliminated from the migratability equation. It rather contributes to virtual state maintained by the operating system, thereby inversely affecting migratability: problem state can be copied verbatim. It is thereby less complex to migrate than virtual resource state.

Virtualization pursues interface properties different from those guiding ABI design, such as resource control. A good example demonstrating the differences in the resulting virtual vs. problem state ratio is networking. Relocation of TCP connections endpoints for application-level failover requires complex extensions to the host operating systems TCP/IP stack. Such a system has been described by Koch et al. [55]. TCP (OSI layer 4) is a stateful, stream-oriented transport protocol, rehosting a server endpoint thereby requires careful synchronization of sequence number and other related state between leader and follower instances, and interim knowledge of the protocol on the side of developers. Migrating the entire system, however, would find the TCP/IP stack transparently migratable

in problem state. Instead, the guest network interface (OSI layer 2) becomes the virtual resource to be relocated. Layer 2 entities, however, provide a stateless, datagram-oriented connection endpoint, thereby of considerably lesser complexity.

Similar properties apply to storage interfaces. Persistent storage for processes is typically accessed as as files. To eliminate the issues of moving media state (i.e. file content and metadata stored on physical disk), file systems need not be local, but may be networked, including the possibility of *diskless* client systems effectively eliminating any dependency on local storage. Still, migrating virtual client state associated with open files requires similar deep understanding of the networked file system protocol and implementation.

Past advances in operating systems research recognized the fact that the core resource abstraction performed by virtually any modern operating system comes at a cost. Additional downsides are trade-offs in I/O throughput, e.g. due to one-sided optimization applied when mapping abstract interfaces to hardware. Another issue is functional (and vendor) lock-in due to lack of choice: The chosen abstraction typically dictates the operating system personality to any application deployed on a given machine, thereby to users. These motivations spurred past research on *exokernel* designs [30], which separated resource multiplexing and protection from abstraction. An typical exokernel TCP/IP stack would be implemented as a library. While no impact on commodity system design can be attributed to exokernel research in hindsight, this demonstrates the functional similarity between operating systems and system virtual machines.

Process Interdependencies The discussion so far focused on the machine interface presented to the guest system. There are additional issues arising with process migration, which can be summarized with the conclusion that the process environment is a non-transparently shared resource. That is, ABI of multiprogrammed operating systems promote some relations of processes which depend on process pairs (or groups) being colocated on the same physical machine. Demonstrably, the resulting relations are typically hard to maintain if only either process is to me migrated across different system instances.

Figure 2.11 depicts part of the IPC mechanisms available. Similar to device I/O, IPC primitives are typically virtual, such as sockets or pipes, which are managed and accessed in virtual I/O space, or physical, such as shared memory. Different from device I/O, if the respective remote process as the peer entity is to remain on the source node, it forms a residual dependency. Depending on the communication primitives employed the interface may not be transparently

or efficiently networkable, shared memory being one example. Such cooperating processes may either need to be migrated groupwise, which may counteract migration for load balancing purposes, or remain bound to the source node.

Avoiding implementation complexity resulting from residual dependencies in IPC relations, microkernel designs such as Mach [1] often considered to be better suited for transparent migration [61]. Since Mach communication interfaces are passing messages, retrofitting of network transports to such interfaces is more straightforward than to system calls.

Present virtual machine intercommunication is typically network-oriented. While guest systems may communicate with each other, the established communication layer forms an virtual OSI layer 2 network topology carrying ethernet frames between them (see e.g. [75, 79, 95]). Compared to local IPC, the network layer employed may generate relative overhead if colocation actually applies, but gains a fair amount of location-independence within a shared LAN or SAN segment.

Control Plane An additional issue is process management across physical system boundaries. It relates to maintenance of the process control block (PCB), which holds metadata such as process IDs (PIDs) and parent/child relationships. Even the simple concept of process IDs easily demonstrates where adaption of process management does not perform well within original process management: process IDs are unique only per host system instance, nonetheless visible to processes, a non-determinism and prone to collision after migration. Sprite introduced persistent *home* nodes, which maintained the PCB.

Solaris MC [53] employed a global process management layer distributed across all nodes, layered above the regular local process management facilities. Process management operations are then either mapped to the local OS instance or redirected via remote invocation. Globalized process management contributes much to full transparency of process relocations to system users. Beyond mere migration, it represents a major building block of single-system-image (SSI) cluster operating systems, such as the MOSIX [11], Solaris MC [53] and the more recent OpenSSI [88] projects.

Still, while global process management and globally unique name spaces can be effectively implemented on top of an original, single-node OS code base, it represents a major change to the original system [53, 89]. Presently, no commercially available operating system of notable popularity includes such features in mainline kernel revisions distributed to end-users.

The issue is that multiprogrammed operating system ABIs comprise a con-

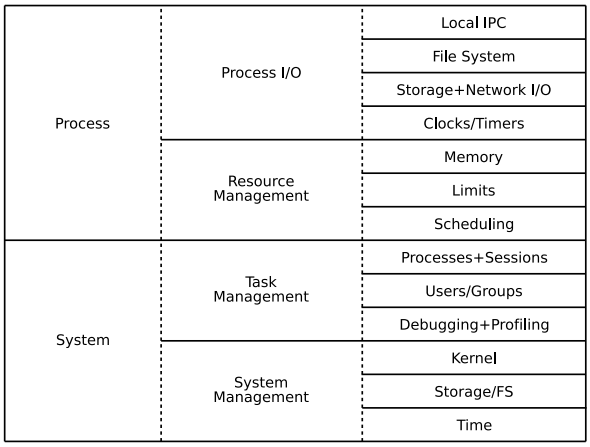


Figure 2.12: A classification of system calls in the Linux 2.6 kernel.

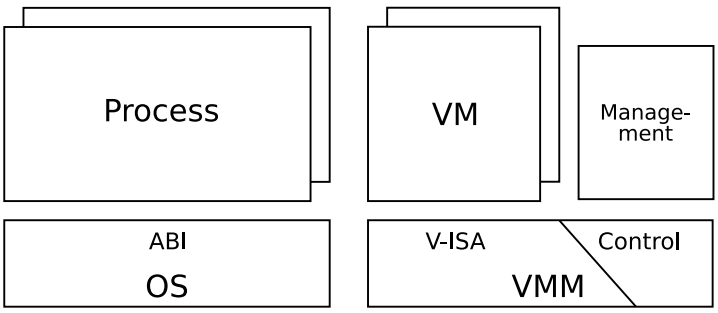


Figure 2.13: Control interfaces in virtualization and process management.

trol interface exposing a shared process environment beyond migratable I/O resources, promoting tight process coupling. Figure 2.12 shows classes of customary system calls derived from the present Linux ABI. While functional requirements of processes may be limited to I/O and resource management, a multitude of interfaces dedicated to system-level and task interaction promote impending deeper dependencies on the process environment. Figure 2.13 contrasts process ABIs with control interfaces in VMMs. System virtual machines are differently designed, due to the fact that resource sharing is to remain transparent for guest systems. Control and monitoring facilities managing guest systems are typically separate from the machine interface, and limited to administrative purposes. In hosted VMs, it integrated into the host, as is the VMM itself. Type I VMMs, such the Xen hypervisor subject to this thesis, feature an interface extensions for this purpose, but it is available only to one or a small number of guest systems with special privileges (see section 4.2).

Summary In summary, a notion of superior migratability of any virtual guest system interface over others may therefore be based on the following criteria:

1. The ideal guest environment is limited to the bare machine interface. IPC should be networked or at least networkable.
2. Resources are ideally remote. If virtual, then networked, thereby more quickly migratable then those backed on local media.
3. Virtual I/O resource state is small, compared to the amount of I/O state kept in problem space.
4. The number of virtual resource *classes* is small, thereby reducing implementation complexity.

Item 1 avoids residual dependencies and/or change to an original host systems which arise if guest systems (processes) are tightly integrated with other guests (processes). Present system VM architectures more adequate since inter-VM dependencies need not be exploited. The host operating environment assumed by guest systems is limited to the machine interface.

Some variants for networked I/O resources (class (N) in figure 2.10), as demanded by item 2, are available for both operating systems and system VMs. Diskless OS installations in clustered systems are far from uncommon, especially for SSI clustering (e.g. [88]). A stronger solution in terms of migratability, however, is full remoting (class (R), bypassing most, if not any virtual resource space in host system software. I/O access in problem state is the domain of system virtualization, e.g. the networking of storage interfaces below the file

system layer. The protocols employed are typically based on present iSCSI [72], Fibre Channel (FC), or the upcoming Fibre Channel over Ethernet (FCoE) [49] standards.

Item 3 at least partly affects migration performance. It amounts to the data volume which needs to be extracted from the virtual execution environment on a case-by-case basis. Since virtual I/O state needs to be migrated and restored per instance, the run-time impact is typically larger than for problem state copied verbatim. Item 4, in contrast, accounts for the complexity of the I/O interface. Server and network applications on system virtual machines are typically sufficiently hosted with a comparatively small number of I/O resources. The two must fundamental classes are network and storage I/O, each typically based on a common device type for any resource instance created. Additionally, disk storage interfaces may be remote, further reducing the amount of resources in virtual space. Optimizing for migratability, the peripheral I/O interface may even be reduced to a single network link carrying both client traffic and any block storage interfaces.

2.3.6 Piecewise Deterministic VM Execution

The major contribution of Popek and Goldberg's virtualization theorem was a proper formalization of the virtualization paradigm for modern computer architectures. The proof of the theorem [65] builds upon a machine model comprising machine memory, basic memory virtualization (trivially based on segmentation) and a trivial resource protection model comprising supervisor and user execution modes. Memory virtualization enables transparent relocation of the virtual machine in machine memory. Privilege levels enforce traps during execution of sensitive instructions by guest. Traps, in turn, enable emulation of a sensitive instruction subset.

The issue of enabling deterministic replay is largely about enforcing the PWD model introduced in 2.2.5. Obviously, the technique employed for classic machine virtualization lends itself for this purpose. The question which arises from this observation is: Can execution model conforming to the piecewise deterministic execution assumption be formalized, conveniently, as a augmentation of instruction set virtualization? Building on the definition of *deterministic events* provided in section 2.2.5, one may start with the concept of non-deterministic instructions. A theorem building upon the original virtualization paradigm might be stated as follows:

Piecewise Deterministic Virtual Machine For any conventional third genera-

> tion computer, a virtual machine monitor may execute guest systems under the piecewise deterministic execution model, if the set of non-deterministic instructions for that computer is a subset of the set of privileged instructions.

In foresight of practical applications, this theorem already carries a general flaw: It does not incorporate the entire set of non-deterministic *events* outlined in section 2.2.5. Specifically, the above theorem does not anticipate events originating externally and asynchronously; which is unfortunate since asynchrony will remain a major subject of this thesis. The same flaw, however, applies to the generally accepted proof of the virtualization theorem. Popek and Goldberg explicitly excluded both interrupts and I/O instructions from their machine model. Their execution model builds upon a purely deterministic instruction set, which contributed much to a sound virtualization approach.

The remainder of this section will be limited to discussion of a proof sketch, thereby avoiding a detailed repetition of the the original machine model and the virtualizability proof it could build upon. The sketch will remain incomplete, as one can show that the machine execution model employed for the original virtualization theorem is not strong enough to incorporate non-determinism without significant (and complex) extensions. A formal proof would therefore go beyond the scope of this thesis. Still, there's some value to be gained: An outline demonstrates how virtual machine monitors lend themselves to practical event logging and replay.

Incorporation of piecewise determinism into system virtualization may be performed in three steps:

1. Incorporation of non-deterministic instructions in the ISA. This has been provided for in section 2.2.5, non-deterministic instructions being a a subset of the complement of deterministic events.
2. Construction of an augmented control program, incorporating control over non-deterministic instructions. This includes logging and respective replay of state changes induced by non-determinism.
3. Demonstrate that the inclusion of piecewise determinism is not in violation of the original properties of a suitable VMM. These properties, as outlined in section 2.2.5, are *equivalence*, *resource control* and *efficiency*.

Figure 2.14 shows an instruction set diagram corresponding to the requirements stated above. Efficiency and resource control are trivially demonstrated. Under virtualization requirements, non-privileged instructions are considered statistically dominant during guest execution. Hence, efficiency is maintained

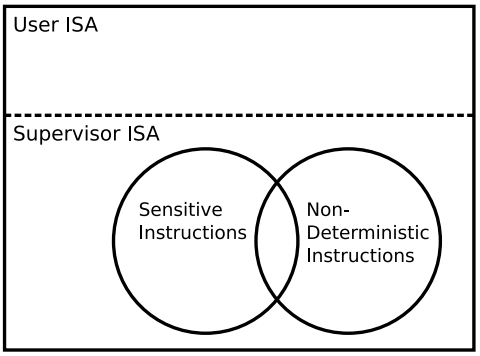


Figure 2.14: Formal requirements for non-deterministic instructions emulation in virtualizable ISAs.

by the same standards guiding whether the original virtualization theorem holds. Furthermore, one can observe no *sensitive* instruction is to be exempted from emulation by the augmented control program. For this reason, overall resource control remains unconstrained as well.

A question more difficult to answer is which effect logging and replaying of non-deterministic instructions has on construction of the augmented control program, thereby on a provability of the *equivalence* property. Figure 2.14 depicts a generalized model, where non-deterministic instructions *may* coincide with sensitive ones. It should not go unnoticed that the intersection in practice can be expected to be considerably large. Non-determinism is typically due to I/O resource performed by guest systems. When multiplexing programmed resources among multiple guests, I/O instructions clearly coincide with the control sensitive subset.

Limiting instructions to functional state transitions, one can resort to a trivial existence proof. Since guest system state is countable and transitions deterministic, emulated guest state transitions can then, in theory, be determined from tables. This constitutes emulation through *interpretation* [65], which avoids additional assumptions about the nature of privileged the instruction set. Non-deterministic instructions, however, cannot be emulated through mere interpretation. Practical instruction emulation is rather based on execution of the original instruction, or a variant thereof, subjected to validation and a variable degree of translation by the VMM. In fact, I/O virtualization requires a slightly stronger assumption about the underlying instruction set architecture: The assumption that any I/O interface programmed by guest systems can be mapped to the original I/O resource, or some appropriate surrogate thereof. Some of the

possible variants of I/O resource accesses by guest systems have been discussed in section 2.3.5. Practical I/O virtualization being one of the most complex issues in system virtualization, due to the variety of different interfaces and techniques, no attempt is made here to provide a sufficiently generalized formalization.

It should be noted that inequality of sensitive and non-deterministic instruction subsets as suggested by figure 2.14 appear to be not uncommon in practice. One example is the x86-architecture `rdtsc` (*Read Time Stamp Counter*) instruction, which will discussed in section 5.1.2. The time stamp counter register [6] is not an I/O resource external to the processor, therefore not sensitive unless actual timing dependencies of guest system on individual counter readings apply.

Traps and emulation of individual guest instructions affect virtualization performance.

Performance downgrade due to state consistency in semi-active replication is largely beyond log generation at the system core. Focusing on tracing, as this thesis does, certainly remains ignorant of quantums of practical interests. Nonetheless

The raw machine interface expected unmodified guest systems is comparatively complex. Classic system virtualization establishes an operating environment functionally equivalent to original machine hardware. The hardware/software interface is subject to constant evolution and refinement in order to incorporate new features and microarchitectural change. In order to meet demands of a large software installation and user base, especially commodity system architectures are required to maintain backward compatibility.

2.3.7 Related Work

So far, several projects have been exploring deterministic replay for different types of system virtual machines. An early prototype of a hypervisor-based system was developed by Bressoud and Schneider [19]. While being a hypervisor, the purpose of virtualization differed to some degree from the commodity virtualization subject to this thesis, e.g. no resource sharing between between multiple guest systems was involved. Functionally, the system therefore differed not much from the monitoring facility generally included by replica managers, which will be discussed in section 2.2.4.

A second major difference was the replication protocol implemented. Synchronization was based on fixed *epochs* (measured in instructions executed) of fixed

lengths, where the protocol implied that interrupts were only delivered at individual epoch boundaries. Such a protocol is comparatively simple to implement, but has drawbacks. One is that interrupt latency is increased. Notification from sources external to the processor are not serviced immediately but delayed by an average of half the epoch length. The second concern that maintenance of global state consistency requires a log protocol partially synchronous with guest execution. This will be discussed in section 2.2.7.

ReVirt [28] was based on UMLinux (FAUmachine) [15], a port of the Linux Kernel to the Linux ABI, similar to paravirtualization. Like UMLinux itself, ReVirt is mainly an extension to the host operating system, adding event logging and replay from a previously generated log. Similar to the architecture discussed in chapter 5, the kernel will log events to a ring buffer structure shared with a user-level process. ReVirt aimed at logging system state for later examination during intrusion analysis. UMLinux is a versatile and accessible virtualization solution, but suffers from some architectural constraints, which will revisited in section 3.3.6.

Deterministic replay has very recently been added to the VMware Workstation VMM for Intel architectures. *ReTrace* [97] is an extension to a hosted virtualization product specifically targeting system profiling and debugging purposes, based on deterministic replay. The idea is based on the fact that an event log comprising only the information necessary to guide an original computation is comparatively small when compared to many types of execution traces used in the scientific and engineering communities for analytic purposes (such as memory or device I/O traces). Since replaying from the log implies that any machine state sequence traversed can be revisited exactly as originally experienced, more detailed and interactively refinable traces can be gained during replay ("trace expansion") without compromising equivalence of each subsequent run. Obviously, this assumption does not include microarchitectural state such as cache or TLB misses. Nonetheless, it holds for any type of event above and including architected state. One concern left is a remaining degree of *distortion*, i.e. differences to execution on bare hardware, which cannot be prevented due to the VMM intercepting access to both sensitive and non-deterministic machine state. The authors report a mean slowdown in trace capturing of 5.09% for CPU-bound workloads.

3 System Virtual Machines

This chapter explores techniques for commodity platform virtualization in practice today. For the purpose of this thesis, the most important one will be *paravirtualization*. However, the specific properties of paravirtual machines justify comparison with traditional system VMs and their specific strengths and drawbacks. Furthermore, paravirtual system VMs did preceed presently ongoing development of extensive virtualization support in general-purpose processors. Hence, the present landscape of system virtualization techniques shall be summarized.

Section 3.1 will introduce *traditional* (or *classic*) system virtualization. This includes the fundamental *trap-and-emulate* implementation style underlying any related VMM implementation, as well as a number of typical issues encountered on commodity processors such as the x86 architecture.

Major manufacturers of x86 and compatible processors have recently begun to augment hardware with direct virtualization support. Section 3.2 will discuss these changes. One might argue that part of the remarkable degree of innovation in the virtualization area during the last years is due to issues encountered on the x86 processor. Lacking original support for traditional processor virtualizability, advanced techniques for efficient virtualization of the original x86 ISA have been developed. On the software side, this gave rise to two major different branches of x86 *software* virtualization technologies: *binary recoding* techniques and *paravirtualization*.

Binary recoding shall only be touched very briefly. Readers interested in the material are referred to [75, 3, 79]. Generally, it employs run-time decoding and adequate translation of guest object code prior to execution, in a dynamic fashion. Code actually executed by a VM is thereby reduced to only a safe instruction subset of the one employed by the original system[1].

Paravirtualization is based around the concept of modifying system software

[1] Throughout the relevant literature, the technique more often called *binary translation* [3]. But this terminology contradicts the terminology of *translation* typically used with process VMs (see section 2.3.2). The approach will be referred to as *recoding* throughout this document.

adequately to effectively support virtualization. Section 3.3 will discuss general properties of paravirtual machine interfaces. A more detailed discussion of the Xen hypervisor then follows in chapter 4.

Section 3.4 will summarize families of virtualization discussed here. Section 3.5 on the foreseeable future of hardare-implemented and paravirtualization.

3.1 Traditional VMs

Traditional system VMs expose an interface to guest systems identical to a machine interface they were written for [2]. This implies emulation of all sensitive instructions, at least to a degree compatible with the feature set used by target guest operating systems. Beyond the raw instruction set, it furthermore implies emulation of native I/O interfaces. To retain control over processor resources, two general techniques are most important for the following discussion:

Deprivileging executes guest system software at a lower privilege level than the original, native modes. As already outlined in section 2.3.4, virtualization of a sensitive instruction set is typically performed by making privileged instructions executed by guest kernels trap to a VMM as the sole entity

Shadowing is a general technique which derives shadow structures from guest-side primary structures residing in a respective machine's physical host address space. The latter is not limited to memory addressing. Dedicated I/O address space or the name space spanned by a set of control registers may be shadowed as well. The predominant example are shadow page tables, but is only necessary on ISAs featuring architected page tables, as does the x86. Its discussion will be delayed until section 3.1.2.

In summary, there are two major issues resulting from deprivileging in combination with page protected virtual memory.

Privilege Compression refers to issues which arise when originally different protection domains effectively share the same privilege level, due to the underlying machine being controlled by a VMM.

Deprivileging of guest operating systems is mandatory, in order to prevent guest systems from controlling hardware resources. However, modern microprocessors commonly distinguish only a very limited set of different privilege levels. As a common example, most operating systems, e.g. POSIX [42], are sufficiently supported with *user* versus *supervisor* modes. Given only two privilege levels, the guest operating system will run in user mode.

However, executing the operating system at the same privilege level as applications will leave the address space dedicated to the guest operating system unprotected. Collisions of privilege levels is referred to as privilege compression.

Address-space compression refers to issues arising from allocating part of any guest virtual address space to the VMM.

Few modern microprocessor architectures implement address space switching in hardware. Instead, the task at hand is performed by system software. This implies that in order to implement control transfers to a VMM, at least part of it (individual trap handlers) need to be mapped into the virtual address spaces of any guest system. A fundamental problem in simulating guest virtual memory is to provide the necessary protection of such memory regions, which should remain private to the VMM.

Protection of virtual memory dedicated to the VMM is commonly performed by guest deprivileging. Since access to virtual memory areas can be controlled based on the respective privilege levels of program code being executed, a VMM may effectively protect its part of any address space. To prevent a VMM from being compromised by guest systems, whether intentional or due to software defects, the respective part of virtual memory needs to be write-protected. Additionally, if guest systems are to be prevented from possible detection (or inspection) of a VMM being present, both read and write access require simulation.

Emulation of the respective memory regions implies redirection of guest access to dedicated page frames by the VMM. This again poses a potential performance problem due to excessive faulting.

Due to its prevalence, the 32-bit IA-32 architecture and its 64-bit-capable successors (*AMD64*, Intel *EM64T*) play an important role in both the server and workstation market. Virtualization of x86-based systems has therefore seen large interest in recent years [31]. However, virtualization of the processor can present major obstacles, depending on the nature of the architected interface.

I/O virtualization apart, this justifies an investigation of the x86 memory and instruction set architectures. For this purpose, legacy “real” and “virtual 8086” compatibility modes are not discussed, as their role in present and future application development is negligible for the purpose of this document. Instead, the following outline focuses solely on the 32-bit and 64-bit x86 protected mode interfaces.

3.1.1 Processor

As described in section 2.3.3, the fundamental paradigm underlying classic virtualization as underlying the P&G theorems employs guest system deprivileging in order to maintain full control of the processor. This section outlines the issues encountered when deprivileging is applied to the x86 ISA. For this purpose, first the protection model of the x86 is investigated. This demonstrates how, and when, address-space compression and privilege compression apply. Last but not least, a number of virtualization issues due to the instruction set need consideration.

Ring Deprivileging The x86 architectures employs a comparatively complex 2-bit protection model with 4 separate privilege levels (*rings*). In order to control processor. Ring 0 provides non-faulting control over the CPU, and maintains control over individual privileges of software running in lesser rings. Operating systems, including various UNIX-flavors and Microsoft Windows, typically use rings 0 and 3 exclusively, where the least privileged ring 3 is assigned to applications.

Ring deprivileging of guest systems reserves control over the processor; only the VMM will run in ring 0. However, different from classic two-level protection models, the guest OS may run at 1 instead. The resulting privilege model is called the 0/1/3 model. Similarly, executing the guest operating system with user privileges corresponds to a 0/3/3 model.

The 0/1/3 model is indeed advantageous, because it supports comparatively simple VMM designs. Control over the processor requires ring 0 privileges, and access control over privileged operations performed by lesser rings is maintainable from ring 0. Since page protection on the x86 distinguishes ring 3 from ring 1, it avoids privilege compression in most guest systems. However, memory protection on the x86 only takes advantage of all four rings when using segmentation. Page protection is limited to a single bit (*user* vs. *supervisor*); rings 0, 1 and 2 are treated equally. This implies that while kernel memory remains protected from applications, the VMM still remains exposed to guest system software. Hence, some challenges remain:

Address-space compression Generally, all practical VMMs for the x86 ISA are mapped to some degree into guest virtual memory.

As an alternative, IA-32 legacy includes hardware address-space switching via *task gates*, which could be used to effectively separate the VMM and guest system address space entirely. Generally, the feature is rarely

employed, the major reason being that there are severe performance implications due to excessive TLB flushes. Furthermore, the 64-bit ISA does not provide task gates, as part of a general, simplifying ISA redesign while following demands of modern operating systems more closely. The issues will be briefly revisited when turning to efficient MMU virtualization in section 3.1.2.

Privilege compression The x86 architecture supports memory segmentation in addition to paged address translation. In 32-bit protected mode, segment limits can be used to protect a VMM [83, 13]. While segmentation in 32-bit architectures cannot be turned off, few commodity operating systems today make any use of it. Instead, most of them follow a "flat" addressing model, where all segment selectors comprise the entire addressable address range [5]. One positive effect is that flat memory segmentation can be virtualized with very little effort; segment selectors are rarely altered and additional memory models do not need consideration. Typically, a VMM may map itself into topmost virtual memory and enforce segment limits accordingly [95].

However, another effect from the predominance of flat segmented memory was that most of the support for memory segmentation has not been carried over to 64-bit modes. Support for memory segmentation was reduced to only a small number of utility segments, in favor of an overall flat addressing model [5]. This limits VMM implementations to the 0/3/3 protection model when hosting 64-bit capable guest systems, which would compromise protection of a guest OS from code user mode code. Essentially, a VMM needs to follow control transfers between guest applications and system software, modifying linear-to-physical address mappings and access privileges accordingly, via a page table switch. This issue is referred to as *ring compression* [83].

Sensitive Instruction Set The x86 ISA is not virtualizable in terms the conditions of the Popek and Goldberg theorems discussed in section 2.3.4. More precisely, there are a number of instructions which are *behavior sensitive* but *not privileged*, i.e. their effect on machine state depends on various individual privilege states of the processor, instead of generating a trap. Results vary from detectability of both the virtualization layer as well as the VMM itself, over operations which simply "fail" silently, to performance or even security issues.

One example of a behavior sensitive instruction is `popf` (*pop flags*), which loads the x86 *flags* register with the value saved on the stack. The *flags* register contains both ALU result codes, to regular program control flow, as well as

the interrupt (`IF`) flag. The latter controls delivery of external interrupt to respective service routines. System software may use the instruction frequently when enabling and disabling interrupts in order to perform critical sections of code conflicting with interrupt handlers. In case of deprivileged kernel code, the instruction will not trap, but merely leave the *IF* flag unmodified.

Ring aliasing *Ring aliasing* [83] generally applies to issues which arise due to guest software being run at different privilege levels than it was originally written for. The result is that deprivileging becomes observable directly or indirectly.

A common case of ring aliasing applies when the actual privilege level is directly observable by guest systems. In case of the x86 architecture, segment selectors in the 32-bit ISA reflect the privilege level the processor is operating at when a selector is created. Code operating in a virtual kernel mode can therefore determine the *current privilege level* (CPL) by reading the present code segment selector from the `%cs` register.

Access to segment selectors is not protectable by the VMM. The instructions involved belong to a larger set of behavior-sensitive instructions which are not privileged.

Visible privileged state Proper shadowing of processor registers requires that only simulated access to primaries shall be performed by guest systems. Virtualization becomes observable otherwise, since the contents of shadow registers are likely to differ from the ones expected by guest systems.

Segment and interrupt descriptor table registers on the x86 ISA are accessible only via dedicated instructions. They can be set only by code in ring 0, attempts from lower rings will generate a general protection fault. The respective instructions for reading (i.e. storing) them however can be used without fault, at any privilege level.

Such issues are considered beyond those called aliasing. Aliasing issues are due to deprivileging, while visibility of privilege (or generally, virtualization) may be considered a more general issue with different parts of the ISA.

Hidden machine state Part of the x86 processor state is affected only indirectly by certain instructions, i.e. it can be modified by system software, but not inspected or directly restored beyond that point.

Segment *registers* are both readable and writable, but only reference in-memory segment *descriptors*. However, processors cache segment descriptors

into a hidden portion of the segment registers in order to speed up segmentation performance. Subsequently modified segment descriptors may therefore leave hidden state irrecoverable.

The problem mostly applies to systems which use segmentation frequently. The processor remains virtualizable, since guest-side changes to descriptor tables can be traced. But the task includes run-time determination of hidden register values and restoration upon a world switch between different virtual machines. Both can be only performed indirectly. Compared to architected processor state, virtualization of hidden state remains significantly more complex.

Interrupt Virtualization Sensitivity of the `popf` instruction entails a second issue: efficient simulation of external interrupts. To maintain processor control, a typical secure VMM will never execute guest systems wit interrupts disabled. The guest-side `IF` flag is therefore commonly a shadowed structure. Instead, it controls the delivery of virtual (simulated) interrupts by the VMM.

Operating systems may toggle this flag frequently, typically in order to protect critical sections of code from racing with service routines. This behavior makes continuous interception of interrupt masking costly. On the other hand, interception is necessary for efficient virtualization of interrupt delivery. The VMM needs to conform to a guest-side `IF` setting in order to maintain fidelity [83]. On bare hardware, interrupts are held pending and triggered as soon as the mask is cleared, which is difficult to simulate effectively by a VMM. IA-32 protected mode operations do not allow for interception of any attempted access on the `IF` flags[2]. Even if possible, only a selected set of combined machine state and flag transitions (interrupt unmasking while VM interrupts are pending) should fault to the VMM, for performance reasons.

3.1.2 Memory

Efficient MMU virtualization is complicated due to a TLB being managed in hardware. The difference becomes clear when compared to MMU virtualization of RISC architectures, which typically employ architected, i.e. software-managed TLBs (e.g. UltraSPARC [80]). Architected TLBs leave determination of an address translation, thereby the structure of page tables, to the operating system; TLB misses are trapped by system software. MMU virtualization on RISC architectures is therefore comparatively simple, since only the TLB needs

[2] In fact, the IA-32 ISA comprises "*Protected Mode Virtual Interrupts*" [46]. However, this covers only part of, not the entire set of sensitive instructions. This is in contrast to statements found in [83].

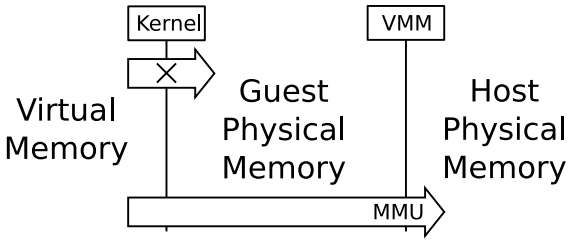


Figure 3.1: Virtual and pseudophysical address translation through shadow paging.

to be virtualized, lending itself to a comparatively straightforward trap-and-simulate mechanism [75].

In order to speed up memory translations upon TLB misses, many architectures implement search logic for in-memory translation tables in hardware. This includes PowerPC [91], IA-64 [47] and, to some lesser degree, UltraSPARC [80]. On such systems, full memory virtualization necessitates memory tracing and simulated write access to in-memory data structures to maintain original machine interface semantics. The x86 architecture being no exception, the entire page tables structure is architected.

The common technique to virtualize architected page tables are *shadow page tables* [3, 75, 87, 95], visible solely to the VMM, as the machine counterpart of the virtual, *primary* tables maintained by the guest system. By protecting guest write access to primary page tables, updates to the virtual tables by guest systems are subjected to traps and simulation by the VMM. The general approach of simulating access to primary memory structures via guest access protection is known as (memory) *tracing* [3].

The resulting translation is depicted in figure 3.1. Generally, two different views of what constitutes a "machine" memory address need distinction. Two terms will be used throughout this document, consistent with the terminology across a considerable part of the relevant literature (e.g. [3, 95]):

- The *machine* addresses space refers to real, hardware memory as addressed on the system bus. It represents addresses e.g. written to a shadow page table and visible to the real MMU.
- The *pseudophysical* address space constitutes page frame addresses as seen by the guest system. These are the addresses written by guests to their primary page tables, translated to the shadow structure by the VMM accordingly.

The implied overhead remains tolerable since the majority of address translations is still performed at the TLB [87]. Nonetheless, page table shadowing remains relatively expensive due to increased memory consumption dedicated to paging and the relative cost of tracing page tables accesses, as opposed to trapped and simulated TLB updates. Page table updates tend to occur in batches, while often only a subset of entries is immediately, if ever, consulted by the MMU. Furthermore, page table entry bits written by the processor (*accessed* and *dirty*) need to be propagated by the VMM so that guest-side paging can continue to function correctly [13].

3.1.3 I/O

Section 3.1.1 discussed the issues of efficient virtualization of external interrupts, which contributes to I/O at the ISA level. Beyond the processor architecture, I/O virtualization on x86-based machines does certainly not suffer from the ISA. However, the I/O architecture of this class of machines have some important properties which deserve consideration as well. The most important issue is the huge variety of different hardware configurations, which affects some architectural considerations when designing a VMM. Another is the relative complexity of the I/O interface.

Peripheral I/O on x86-based systems is typically open-ended, which contributed much to the original success of the IA-32 ISA (then being one fundamental building block the original "PC architecture"). As a consequence, today's market for commodity system components hosts a whole ecosystem of independent equipment manufacturers. Today, there exists a multitude of different devices and correspondingly different interfaces. At the same time, device driver design largely depends on individual operation system interfaces[3]. The result is a high entry barrier for system software not integrated with established operating systems, since only the latter can support a relevant share of individual machine configurations.

This in turn affects options in VMM design. One major advantage of hosted virtual machines is that requisite software support for a given platform is largely provided by a present host operating system. Virtual resources provided to guest systems are then mapped to the host system by a respective VMM. Classic (Type

[3]During the mid-1990s, different corporate initiatives aimed at unifying I/O interfaces. One was UDI (*Uniform Driver Interface*), promoting an OS-level driver interface which was source-level compatible both across operating systems as well as different processor, I/O bus and memory architectures. Another was I_2O (*Intelligent I/O*), a *split-driver* model comprising class- and OS-specific drivers for a uniform, standardized set of class-specific interfaces at the I/O layer. None gained significant market share.

I) VMMs, in contrast, would run on bare hardware. Section 4.2 will introduce the concept of delegated I/O virtualization to an operating system within a Type I VMM architecture.

3.2 Hardware x86 Virtualization

Presently, x86 virtualization is mainly pursued by two competing manufacturers on the market: Intel and AMD. Despite smaller differences, both product lines share much in common, and is supported in coexistence by most virtualization products available on the market.

Hardware virtualization is a presently ongoing effort. It is not only affecting the instruction set architecture, but future extensions to memory and I/O subsystems as well, which constitutes an incremental process. This section only briefly describes two components of presently available architecture extensions. The most fundamental change is general virtualization of the bare processor ISA. It adds two new modes of operation to the x86 architecture, which are commonly called *Root* mode, dedicated to a VMM, respectively *Non-Root* mode, which runs guest systems deprivileged [83, 5].

The second extension described here are *Nested Page Tables* (NPT). While NPT remains only optional to x86 hardware virtualization, it is deemed important to deterministic replay in future paravirtualization, and will be revisited in following chapters.

Privilege compression is generally due to the fact that while processors typically distinguish only two privilege modes, *user* versus *supervisor*, a fully virtualized processor needs to host three different entities: the VMM, a deprivileged guest system kernel, and user applications. All of these entities require individual protection. Applications need to remain isolated from each other, which remains to be ensured by address space separation. However, a traditionally deprivileged guest kernel will share the same privilege level as applications. As outlined above, this is at least the case with the 0/3/3 privilege model 64-bit system have to resort to. Therein, the VMM must equally remain protected from the guest. Ideally, it VMM could be located in an entirely separate address space in order to counteract address space compression.

Due to the major importance of backward compatibility in the x86 market, mere changes to the protected mode privilege levels are insufficient for full machine virtualization. The x86 ISA consists features considerable number of different operating modes changing the semantics of the instruction set. Examples are 16-bit real mode as well 32- and 64-bit protected modes, both of the lat-

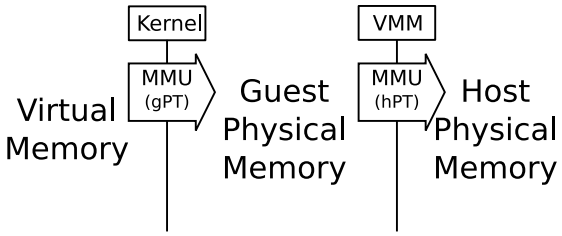


Figure 3.2: Address translation with nested paging.

ter accompanied with a number of additional compatibility modes to integrate legacy systems on newer architectures.

Consequently, a "hyperprivileged" mode capable of virtualizing the entire processor ISA needs to support any of the above. Guest system execution therefore employs a privilege set which remains orthogonal to ring-based protection: In one dimension, a virtualizable x86 processor distinguishes root mode from non-root mode, where root mode corresponds to the hyperprivileged mode of the VMM. Whether operating in root or non-root mode, each separately compatible execution modes, e.g. x86 protected mode with customary 4-level privileges.

While root mode thereby appears not much different from a non-virtualized processor, non-root mode implies deprivileging. Transitions between root and non-root mode are typically entries to and from the VMM. One notable advantage of such a design is that it supports hosted VMMs particularly well, since the host system may wish to utilize the full set of available privilege levels in root-mode.

Nested paging (NPT) has two major purposes [5]: (1) it removes the need for the hypervisor to shadow guest page tables by intercepting individual updates, and (2) it solves the overall problem of address space compression. NPT augments the existing hardware page translation mechanism by a second level of translation.

Figure 3.2 shows the resulting address map. Where shadow page tables would map virtual guest addresses to machine addresses, machine memory virtualization with nested paging leaves one *guest* page table (gPT) directly writable to the guest system. These tables thereby map virtual addresses to the pseudo-physical guest addresses. Address translation by the MMU then translates guest physical addresses to real machine addresses by a second translation sequence, by consulting a *host* page table (hPT) private to the VMM.

Nested paging is only available when ISA virtualization is enabled as described above. While guests running with nested paging enabled transparently receive two levels of page table translation, all systems running in root mode, including the VMM, are exempted from this mapping. Instead, the VMM my populate a global virtual address range entirely separate from all non-root modes. The issue of address space compression arising with a VMM resident in guest virtual memory is thereby eliminated.

3.3 Paravirtual Machine Design

3.3.1 Concept

Classic system virtual machine design enforces a large degree of binary compatibility, not only for applications, but also the operating system. Full virtualization includes support for any core kernel components comprising architecture-specific code. Additionally, a relevant set of device drivers, e.g. for storage or network host controllers, needs to be supported. As learned from previous sections, the degree of compatibility comes at a cost.

Paravirtualization is a variant of system virtualization which differs from classical VM design in that full binary compatibility with the target platform is abandoned, in order to improve performance, scalability, and simplicity of the virtualization layer [93, 13]. For this purpose, an idealized software-interface, similar but not identical to the physical machine is defined and implemented by the VMM. Architecture-specific portions of guest systems are rewritten (*ported*) to the surrogate software interface.

Contrasting the trap-and-simulate techniques employed by traditional VMMs, guest systems on paravirtual machines are thereby *virtualization-aware* to some desired degree. System protection unconstrained, paravirtual machine guests will typically not issue privileged or sensitive instructions. Instead, the surrogate machine interface evolves around the idea of a non-privileged subset of the machine ISA, *extended* by service routines, which are typically referred to as *hypercalls*. Service routines are called by guest operating systems to request assistance in tasks requiring supervisor privileges. Throughout this document, such an extended ISA will be referred to as the PVMI (*Paravirtual Machine Interface*)[4].

Considering the VM taxonomy introduced in section 2.3.2, the idea of ex-

[4]Inspired by VMI, the title of a paravirtual machine interface originally proposal by the makers of the VMware virtualization product [10].

tended machine interfaces for protected execution of privileged services is strikingly similar to the objectives of common operating system ABIs. It should be noted however, that the approach is not particularly new. First proposals for cooperative approaches in virtual machine interface designs date back to the early days of system VM applications [35]. The major difference between operating system ABIs and PVMIs is the individual degree of interface abstraction applied.

Different from established standard ABIs, e.g. POSIX [42], paravirtual machine interfaces are still subject to research and ongoing refinement by individual system vendors. At the time of this writing, there are at least three different, vendor-specific PVMI specifications alone for x86 platform virtualization:

Xen (Univ. of Cambridge) as the first representative of paravirtualization for commodity operating systems was published in 2003 [13]. Different from the alternatives below, a core fraction of the hypercall interface is largely independent of the processor ISA. It presently supports 32- and 64-bit x86 but IA-64 and PowerPC architectures, as well as Linux guests and a number of additional Unix variants.

VMI (VMware, Inc.) was specified in 2005 by VMware Inc. VMI is implemented by company's own VMware ESX Server (Type I) VMM. It was proposed as a standard PVMI for the Linux kernel, but received only limited adoption as one of the available alternatives.

Hyper-V (Microsoft Corp.) was published in 2008 [60]. It is presently supported by a respective suite of virtualization products developed by Microsoft, comprising a hypervisor for x86 platforms, in support of paravirtualized deployments of the MS Windows Server 2008 operating system.

The prevalent operating system ABIs in use today, e.g. POSIX [42], are designed for portability and independence of any particular machine architecture. Core resource management at the process ABI, most notably memory and I/O, is thereby largely independent of a particular machine family. Paravirtualization is different in that it performs at a much *lower* abstraction level. This is well justified: contrasting portable operating system ABIs, a PVMI is not designed to facilitate portability of guest systems to different system architectures. For this purpose, most commodity systems in use today facilitate hardware abstraction levels of their own. Instead, even highly machine-specific calls to the VMM provide access to relevant platform features where necessary.

The major benefits of paravirtualization compared to full virtualization are due to the non-transparency of virtualization at the guest interface. They can be classified as follows:

Machine Idealization leaves details of individual subsystems at the physical level to the VMM [13]. This facilitates some desirable degree of *abstraction* at the VM interface.

A benefit derived from abstraction is more optimal behavior of the guest. Where full virtualization needs to sacrifice performance in order to maintain full transparency of the virtual machine map, paravirtual interfaces can be tuned for relatively low overhead.

Virtualization-awareness exposes physical details of the virtual subsystems to the guest system. Once established, virtualization-awareness enables *cooperation* on the side of the guest system. Where full virtualization needs to hide performance-critical details of the virtual machine map, paravirtual guest systems may commit to an altered execution environment.

The following sections follow the application of these two principles among the core machine resources, being processor, memory and I/O virtualization. A sufficiently general discussion can remain independent of individual implementations. The Xen hypervisor as one particular representative will be discussed in chapter 4.

3.3.2 Processor

With respect to the processor, the level of machine idealization and efficiency through virtualization-awareness can be outlined as follows:

Sensitive Instructions The issue of potential non-virtualizability of commodity processors can be overcome by appropriate modifications to the guest system. Behavior sensitivity typically affects guest code operating in virtual supervisor mode. Non-privileged application code, in contrast, need not change.

Privilege Compression Elimination of privileged instructions is due to cooperation on the side of guest systems. System protection may remain unconstrained, i.e. guest kernels need not be trusted. Privileged operations on behalf of guest system may remain subject to validation by a respective VMM. Hence, paravirtual machines may remain affected by privilege compression, depending on the processor architecture.

3.3.3 Memory

Contrasting traditional system VMs, paravirtual guest systems do not necessitate full transparency of physical memory address space virtualization. To

virtualization-aware a guest memory management, physical addressing may be revealed. In practice, guest systems still require mappings between linear and physical addresses to function properly. But cooperation entails significant advantages:

- In case of hardware-managed TLBs, virtualization-aware guest-systems may gain read-only access to the real page tables. Contrasting the extreme of shadowed page tables, this reduces the overall amount of memory dedicated to virtualize paging. In order to control memory reservations, updates performed by the guest system are performed via service routines. This removes any need for tracing and simulation.
- Beyond the virtualization layer, there are guest-side optimizations in memory management which depend on awareness of the physical memory topology. There are various opportunities for intelligent page placement. One is page coloring in order to reduce impending collisions in memory caches indexed by physical addresses [52, 13], which remains ineffective if cache indexing of a physical memory map does not correspond to that of the machine. Another example is efficient NUMA support in SMP-capable guest-systems. To be effective, NUMA-capable systems need to take memory topologies and a machine memory distribution among different processors into account [24]. Hence, transparent migration of virtual processors by a VMM is likely to counteract guest system optimization strategies.
- Physical memory management controls the actual memory allocation of individual guest systems. Memory *reclamation* is best managed not by the VMM, but from the inside of the guest systems [87]. The common technique employed is extending the guest OS with so-called *balloon* drivers [87, 95]. To reclaim memory, pressure on guest system memory management is increased by allocating memory from it. The advantage is that guest-side page replacement can make better decisions about which pages to reclaim than the VMM, which typically lacks intrinsic knowledge about guest-side memory management strategies applied [87].

3.3.4 I/O

Section 3.1.3 stated how the large variety of I/O hardware constrains some of the options in VMM architecture when applied to commodity platforms. However, the same diversity affects the design native operating systems. One consequence is that virtually all portable and commodity operating systems today feature software abstraction layers for all relevant device classes. While subsequent changes to architected processor interface support is usually not

feasible on end-user installations, most commodity systems feature open-ended driver architectures for peripheral I/O.

This helps to avoid I/O device emulation. For this purpose, paravirtual guest systems feature dedicated drivers for individual guest systems, which rely on idealized device interfaces at the ISA level. Benefits of I/O interface idealization can be summarized as follows:

Elimination of native I/O Emulation of hardware interfaces commonly involves emulation of port-mapped or memory-mapped I/O. In paravirtualized guests, control transfers to a given I/O virtualization layer may employ guest-side calls to service routines instead, essentially removing the need for traps and device emulation by the VMM.

Extensible Control and Status Control and status interfaces for virtual devices may be mapped to shared memory. One advantage of shared memory is that the need for copying control information between guests and and the VMM is reduced. But more importantly, structure and content of such control information will typically be device- or device-class specific. Mapping control interfaces to memory is extensible without changes to the core VMM interface. In fact, only requisite means for the *control* path (i.e. notifications between virtual I/O devices and the guest system accessing it) and *data* path (i.e. shared memory) need to be provided [38].

High-level Command Interfaces Operations on idealized devices may be performed at a granularity approximating that of the device driver level. Hence, the number of individual control transfers to the virtualization layer, when compared to emulation of native I/O, can be reduced to a reasonable minimum [75]. As a side effect, idealized interfaces make device driver design to a respective OS interface relatively straightforward, especially when considering the complexity of modern hardware device drivers [32].

Zero-copy I/O Bulk data transfers at the virtual interface may employ DMA-style memory accesses for transmission. For this purpose, it is advisable to transfer I/O data separate from control information (*out-of-band* buffering), which can greatly reduce the need for expensive intermediate copying of data at the guest system boundary [13]. Instead of transmitting buffer contents, communication disseminates references to machine memory carrying buffers. As a special case, this facilitates optimization techniques such as "short-circuiting" of inter-VM transfers in virtual network topologies [75], which can then move user data without copies.

3.3.5 Applicability

Compared to the other extreme of full virtualization, paravirtualization has both advantages and disadvantages. The major disadvantage is the necessity of modifications to respective guest systems. In the case of proprietary system software, its mere applicability depends on source-level accessibility of operating system code and sufficient backing on the side of respective owners. Furthermore, sufficiently liberal licensing regarding the distribution of derivative works needs consideration. Presently, it therefore lends itself to either vertically integrated systems, i.e. hardware/software bundles maintained by a single vendor, or open source kernels. The latter case is the predominant one in the present commodity market, especially in absence of sufficient vendor backing.

Even taking source code accessibility issues out of consideration, not only initial porting, but long-term maintenance of target systems needs to be taken into consideration. To reduce the overall cost of maintenance on the side of system vendors, guest system interface standardization is an important part of the equation[5].

Last but not least, the ability to run *legacy* operating systems (i.e. systems with low attraction to both port and maintain) is lost, as well as that to run systems which shall run unmodified by user policy.

3.3.6 Related Work

Many modern virtualization solutions pursue paravirtualization to a greater or lesser extent. Especially the concept of paravirtual I/O as described above, i.e. idealized software interfaces at the ISA level, has been applied to a wide range of virtual machine solutions ([75], [32]). For this purpose, I/O emulation is performed only as long as necessary, e.g. in order to aid operating system installation. Original drivers are eventually replaced by those operating on machine interface extensions. The approach requires maintenance of system software per guest operating system, but is justified by large improvements in performance or user experience.

The term *para-virtualization* was originally coined by the *Denali* system [93].

[5]The delicacy and present degree of uncertainty regarding that process is best exemplified with the present state of paravirtualization in the Linux operating system. So far, multiple competing vendors of paravirtual machine monitors found themselves seeking acceptance in the mainline distribution of the Linux kernel, with different proposals. Ultimately, the issue was solved not by commitment, but rather adding another level of indirection to the respective kernel sub-architecture support. The interface is called *paravirt-ops* and part of Linux since version 2.6.21 (April 2007)

However, the focus of Denali was mainly on "lightweight protection domains" for large-scale service multiplexing, meaning very large numbers of virtual machines on a single host system. Simplicity being key to that goal, not only interface idealization was pursued, but many core properties of existing operating systems were taken out of consideration, including guest-side task switching and virtual memory. Presently, the concept of paravirtualization is primarily associated to the Xen virtual machine monitor, discussed in the following section.

Within the Linux kernel, ports to virtual environments are not new. Linux on the IBM S/390 and zSeries mainframe architectures has been developed by IBM since 1999. Some do consider the approach comparable to paravirtualization [13]. However, one significant difference is that the hardware/software interface in IBM mainframe partitions is built on CPU microcode. The philosophy is different to the extended instruction subset of a PVMI implemented on top of the original machine instruction set.

Both *User Mode Linux* (UML) [26] and FAUmachine [15] (formerly UMLinux) are ports of the Linux kernel to the Linux ABI. Since the ABI does not meet the paravirtual design criteria of a machine-centric interface described above, it demonstrates that porting operating systems even to interfaces at significantly higher levels of abstraction is feasible. However, compared to dedicated, machine-level interfaces, a number of issues remain. A predominant one is privilege compression: the UML project comprises dedicated patches to the host system in order to separate guest kernel address spaces ("SKAS") from applications. While the changes are comparatively small, it demonstrates some of the limitations of a process ABI in order to host whole-system VMs.

3.4 Taxonomy of Commodity VMs

The prior discussion of commodity hardware virtualization concludes with a graphical representation of a present virtualization landscape for commodity systems. Figure 3.3 displays a taxonomy of system virtual machines, as a refinement of the original taxonomy of figure 2.8. Note that an extended taxonomy does not need to contradict the original sub-classing applied. All techniques presented cover *same-ISA* and *system VMs*; the applicability of a categorization into either *hosted VMs* or *Hypervisors* remains unconstrained. Instead, present technological advances account for two additional dimensions:

Guest ISA Compatibility One dimension is the machine interface provided to guest systems. For traditional VMs, it is identical to the original machine interface, which enables guest operating systems to run unmodified. One

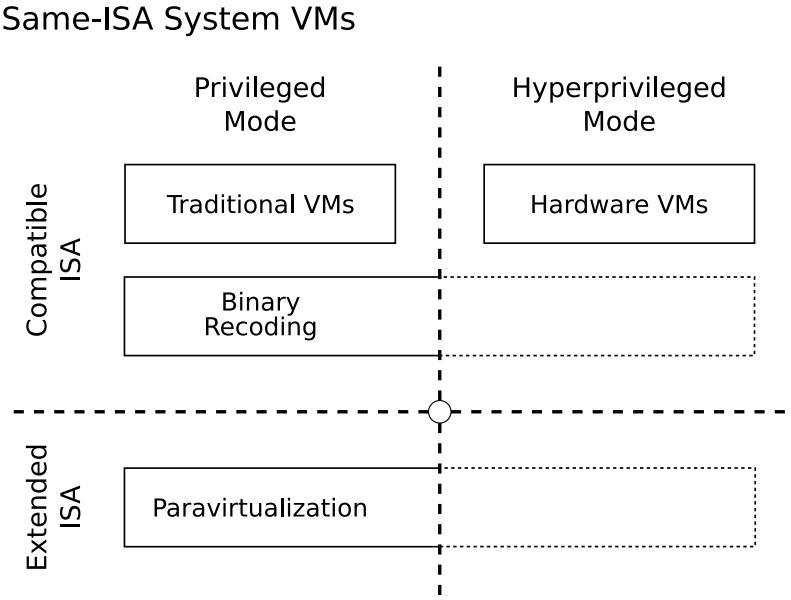


Figure 3.3: A present and future taxonomy of commodity system VMMs.

can summarize this category as *compatible* ISA VMs. This class includes binary recoding techniques as well, since the performed ISA subsetting remains transparent to the guest machine interface.

Paravirtual machines do not produce a superset of the original machine interface; ISA subsetting applies here as well. However, the PVMI differentiates themselves by adding new elements to the remaining non-privileged interface. Figure 3.3 therefore classifies this as *extended* ISA.

Hardware Virtualizability The second dimension describes the degree of virtualizability of the underlying machine interface. Traditional system VMs are built upon a privilege and protection model matching demands for contemporary, yet native operating systems. Virtual machine monitors satisfied by traditional processors are referred to in figure 3.3 as *privileged mode* VMMs. With architectures virtualizable in the traditional sense, this includes "Traditional VMs". In the case of traditionally non-virtualizable machines, this includes binary recoding and paravirtualization as outlined above.

Given explicit processor support for virtualizability, neither paravirtualization nor binary recoding are necessary. Instead, the concept of the traditional VMM can be readily carried over to a dedicated processor interface.

Presently, this is most often referred to as a *Hardware VM* (HVM), where a VMM will operate in some sort of *hyperprivileged* mode[6].

3.5 Future Evolution

Surprisingly, hardware virtualization for commodity systems presently appears to remain non-disruptive to the market for existing software virtualization platforms. Instead, it was noted that, despite largely simplified VMM design, no fundamental performance gain can be attributed to pure hardware VMs [3].

Both Intel VT-x and AMD-V extensions may still improve as their implementations mature. Hence, present performance concerns may vanish over time. Adams and Agesen suggest that hybrid approaches combining software virtualization, foremost binary recoding, with some of the more positive effects of hardware extensions may evolve in future [3]. Figure 3.3 lists presumable future accordingly: dotted lines indicate assumable future adoption of hyperprivileged operating modes for present software virtualization techniques.

Such hybrid VMMs combining hardware and prevailing software virtualization techniques are likely to represent the future in paravirtualization. Section 3.1.1 noted that ring compression, remains a major obstacle in x86 virtualization especially with 64-bit operating modes. Lacking segmentation, any secure VMM needs to adjust guest page tables to protect itself from the guest kernel. An attractive option to shadow page tables are nested page tables implemented by the MMU. However, x86 processors implementing page table nesting are not yet broadly available.

More importantly, use of NPT represents a partial break with the present machine interface presented to paravirtual guests: Due to the virtualization architecture implemented by present x86 processors from either AMD and Intel, NPT is inseparably bound to full ISA virtualization, i.e. the guest system needs to run in VMX Non-Root mode (AMD-V guest mode, respectively) which significantly differs from e.g. the ring-1 deprivileging 32-bit paravirtual guests are subjected to, and assuming to be present.

[6]This term is rarely found in the presently relevant literature. Its use for the purpose of proper classification and was inspired by the documentation for hardware VM support recently introduced to the OpenSPARC architected interface [81]

4 The Xen Virtual Machine Monitor

Whole-system paravirtualization has seen large interest in recent years. This is in part due to its primary representative being a robust implementation available as free software and correspondingly liberal licensing. Originally developed at the University of Cambridge, the Xen hypervisor received remarkable backing by commercial system vendors, including Intel, AMD, IBM, Sun and Hewlett-Packard. Xen is available for a number of different system architectures, including x86, IA64 and PowerPC.

The Xen project originally focused on a machine interface and support for fully paravirtualized guest systems. The mainline guest operating system is Linux 2.6. Original research at Cambridge additionally spawned ports of the NetBSD and Microsoft Windows XP operating systems, where the latter one presently remains unavailable to the public. Other systems available include OpenSolaris, FreeBSD and OpenBSD.

Beyond paravirtualized guest systems, the present version 3.0 of includes support for hardware-virtualized x86 processors as introduced in section 3.2. Due to the large differences in machine interface, hardware virtualization support differs much from paravirtualization, but can be integrated in the same overall virtualization solution. This section focuses entirely on paravirtualization of x86 processors, excluding present advances in hardware virtualizability.

4.1 Architecture

Formally, Xen is a regular hypervisor, i.e. running on bare hardware, without a hosting operating system in between. As pointed out in section 3.2, this is unusual on commodity platforms, e.g. considering the issues of supporting peripheral I/O for a virtually unmanageable number of possible machine configurations. In fact, the dependency on any commodity operating system featuring the necessary platform support prevails. However, Xen layers this system on top of the VMM, a major design element which deserves closer examination.

Figure 4.1 depicts the basic architecture. The VMM only provides for the most elementary subsystems of the real machine:

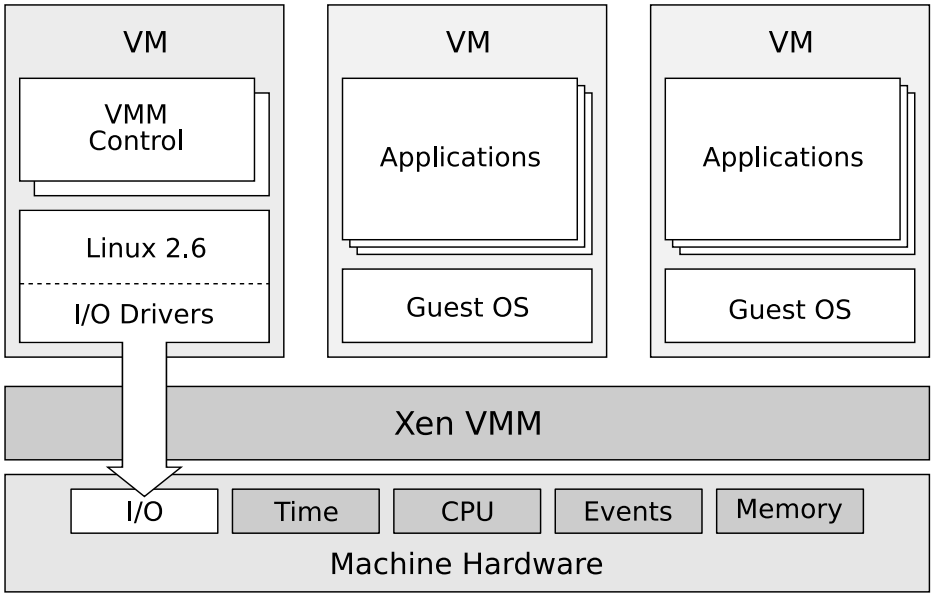


Figure 4.1: Xen virtual machine architecture.

- CPU virtualization, i.e. scheduling of available physical processors among a number of virtual ones (VCPUs).
- Machine memory, including virtualization of the MMU and allocation of available physical memory. Additionally, the VMM implements sharing of pages among different paravirtual machines.
- External events, i.e. routing and transmission of signals which drive asynchronous control transfers on a virtual processors. Xen maps machine interrupts to a software abstraction called *event channels*, which will be discussed in section 4.4.
- Timing sources, including cycle-accurate presentations of real and virtual (i.e. VM execution) time. Additionally, the VMM provides timer-driven events to guest systems: a periodic timer event operating at 100Hz frequency and a programmable interval timer.

To ease porting of operating system software, the processor interface matches the architected ones wherever idealized interfaces are not advantageous. Examples are include interrupt stack frames and the format of architected page tables [95].

With the exception of time, as one of the most fundamental resources (and

one typically implemented either directly within the processor or accompanying chipset), the above list excludes virtualization of I/O devices. In fact, these devices are driven not by the VMM, but dedicated class of domains which are granted sufficient access to I/O ports and memory. This is discussed in the following section 4.2. Virtualization of the same resources in order to achieve resource sharing between multiple VMs will be described in section 4.5.

4.2 Domain Privileging

As shown in figure 4.1, VMM control over hardware resources excludes peripheral I/O, such as network or storage interfaces. Instead, Xen leaves the task of controlling various I/O hardware to a sufficiently privileged class of guest systems. The virtual machines hosted by Xen are referred to as *domains*, distinguished by numbers (*Domain IDs*), i.e. `dom0`, `dom1`, etc. Conceptually, Xen differentiates between three different domain types:

Privileged domains are permitted to access the VMM *control interface*, i.e. perform administrative tasks like creating or destroying additional domains and control their resource allocation.

Driver domains do not control the VMM but are entitled to directly access a (potentially limited) set of hardware I/O devices. Barring control over machine memory and processors, they may therefore drive various peripheral devices on behalf of the system as a whole.

Unprivileged Domains are "regular" domains, typically the ones employed to host users and applications. Device access is fully virtualized. Xen was designed to carry up to approximately 100 of such unprivileged, paravirtual domains [13].

In practice, the distinction between driver and privileged domains often remains only a conceptual one. After initialization, the Xen VMM boots into an initial `dom0`, as the single privileged domain entitled to perform administrative operations. Hence, control plane software is running on `dom0`. In fact, the life cycle of the VMM is bound to the domain, halting it implies a system halt subsequently performed by the VMM. The same domain may be employed as a single driver domain.

The same initial domain operates as an driver domain. The VMM contains just enough platform-specific code to provide user interaction during system boot, then releases the console to domain 0. Domain 0 thereby provides console I/O to an administrator, driving video and input devices, while processor

and memory allocation are controlled by the VMM. Domain 0 typically runs a privileged but paravirtualized Linux kernel, an X11 windowing environment and user applications as the administrator sees fit, including control software to create and administer all additional virtual machines. In figure 4.1, the leftmost virtual machine corresponds to domain 0, combination of control and hardware privileging. Addition domains may host arbitrary operating systems. For the remainder of this document, it shall be sufficient to refer to both driver and control-privileged domains as `dom0`.

System decomposition into VMM and driver domains fulfill various purposes. Consistent with considerations introduced in section 3.3.4, the paravirtual machine interface does not include hypercalls dedicated to peripheral I/O, but only to basic CPU and memory subsystems described in the following section. Hence, I/O virtualization may be performed elsewhere, provided that the necessary communication primitives (shared memory and control transfers by remote notifications) are provided.

- One reason of practical importance is to offload the task of providing device drivers to existing systems, such as the Linux kernel. In that respect, driver domains in Xen combine bare-hardware virtualization achieved with hypervisors with the accessibility of hosted VMMs.
- More fine-grained system decomposition into multiple driver domains can increase dependability through isolation, the same goal which spurred research on microkernels [38]. Different from monolithic operating systems, faults in driver domains occur in isolation and can recovered from [32].
- Another consideration is that system decomposition into separate I/O domains may gain more traction in future for security reasons. This is the one of the ideas behind initiatives such as Intel's upcoming *Trusted Execution Technology* (TXT) [48], formerly known as *LaGrande*.

4.3 Machine Interface

At the core of its guest system interface, the Xen VMM exports a set of hypercalls, which available exclusively to the guest operating system kernel. Call initiation and return builds upon the same processor mechanisms for privilege elevation employed by OS system calls. On the 32-bit x86 architecture, a dedicated software interrupt vector (`int 0x82`) is allocated. 64-bit processor modes share the `syscall` instruction with the guest system for this purpose.

The Xen 3.0 PVMI consists of 38 hypercalls plus a number of architecture-specific ones [95]. The non-privileged set of hypercalls derives from the parts of the machine interface maintained by the VMM.

Processor:

- CPU scheduling, e.g. voluntary CPU yield or domain shutdown.
- VCPU control, such as initialization, activation/deactivation and status inquiry.
- Task switching.
- Special register access, e.g. debug resources.
- Trap table setup.
- Return from interrupts.

Memory:

- Physical memory management, e.g. in support ballooning
- MMU management.
- Inter-domain memory sharing.

Events:

- Event callback setup
- Event channel operations
- Scheduler interaction (e.g. blocking to notification)

Time and Timers:

- Interval timer control
- Scheduler interaction (e.g. polling event channels)

Other calls include operations reserved to privileged domains, i.e. `dom0` or I/O operations for driver domains. Machine-specific calls on x86 include memory segmentation and VM86 mode.

Hypercalls are not only ISA elements instructions triggering transitions to the VMM. Xen emulates privileged instructions wherever the complexity of both instruction parsing and emulation is not significant enough to justify addition of a hypercall. Examples would be x86 `in`/`out` instructions or control register access.

Both hypercalls and instruction emulation mark *synchronous* transition points for control transfers from guest systems to the VMM. These are software-induced traps driven by software execution, i.e. synchronous with regard individual instructions causing their occurrence. A variant are synchronous events are

	Synchronous	Asynchronous
VM → VMM	Traps + Hypercalls	External Interrupts
VMM → VM	Trap Bounces	Event Notifications

Table 4.1: Control transfers between the Xen VMM and paravirtual guests systems

traps and faults triggered due to errors in software execution, *exceptions* in x86 architecture terminology.

Asynchronous control transfers, in contrast, typically originate from sources external to a processor. At the machine ISA, these are external interrupts triggering immediate control transfers to dedicated ISRs in system software. Interrupts either originate from I/O devices or from other CPUs, as inter-processor interrupts (IPIs).

Combinations of synchrony and the transition direction between virtual machines and the VMM are shown in table 4.1. In order to maintain control of the processor, all processor interrupts and exceptions are serviced by the VMM. Real hardware interrupts cannot be trusted to guest systems in a safe fashion, since non-cooperative or corrupt interrupt handling may divest the VMM from control over the processor [32]. In particular, Xen maintains the interrupt controller and performs IRQ acknowledgment before scheduling the ISR in a driver domain. If faults are due to machine resource management they remain up to the VMM to handle; guest system software will continue operation afterward without further notice. Hence, all hardware-induced control transfers return control back to the VMM.

Like with transitions to the VMM, the virtual machine interface comprises both synchronous and asynchronous transfers from the VMM to guest systems. If exception handling is up to the guest system, it is forwarded to the guest. This scheme is performed for many exception types, page faults being a common example. The simulation of guest interrupts in software is called a *trap bounce*. On x86 processors, hardware exception handlers in system software are controlled via an architected *interrupt descriptor table* (IDT) managed in host memory. Xen interfaces to guest systems via a virtual IDT (a *trap table* [95]), whose format is an idealization of the x86 descriptor layout. In contrast, exception numbers and the stack frame layout remains compatible to the real machine in order to ease porting of guest kernels.

Contrasting the real IDT [46], a virtual IDT controls guest exception handlers exclusively. The discussion so for did not consider interrupts at the *virtual* machine. In fact, virtualization of external events follows a different model. *Event notifications* are subject to a much greater degree of interface idealization. This is described in the following section.

4.4 Event Virtualization

External interrupts to a guest system typically originate from virtual sources, as opposed to the real machine. An example are the virtual interval timers and timer interrupt mentioned in section 4.1, both being fully idealized resources which multiplex the real hardware timer circuitry controlled by the VMM. The same applies to inter-processor interrupts connecting not physical, but virtual processors then mapped to the real machine. There are two exceptions to this scheme, both of which are due to the paravirtual domain architecture outlined above: servicing of hardware interrupts by driver domains [32], and inter-domain event notifications.

Xen manages all external events via *event channels*. Like hypercalls, event channels are pure software construct turned into an architected element of the paravirtual machine interface [95]. Those rather familiar with the POSIX interface [42] may consider a comparison with I/O notifications on UNIX-based systems: I/O *handles* are integer variables referencing privileged state. For asynchronous I/O, signals drive thread control transfers upon state changes of the underlying descriptor. Alternatively, blocking operations may suspend a calling thread until an event occurs.

Similar mechanisms are found in microkernel architectures such as Mach [1]. Different from Mach's port-based communications and POSIX IPC however, event channels carry no messages, only operations affecting control flow within the receiving side [38]. Data exchanges is commonly performed in shared memory, which will be briefly discussed in section 4.7.

The guest-side handle to an event channel is an integer *port* number referencing control information private to the VMM. A channel may be created by guest systems and *bound* to one of a number of different event sources, depending on availability and assigned privilege level:

- Virtual interrupt lines, e.g. the periodic clock tick provided to any guest system.
- Physical interrupt lines, i.e. those tapped by `dom0` or additional driver

domains.

- Inter-processor notifications, i.e. event signaling between two VCPUs within a single domain.
- Inter-domain notifications, i.e. event signaling between domains sharing the physical machine.

The latter two bindings are bi-directional, i.e. notifications may be sent and received on either end. Event notifications are delivered via the same trap bounce mechanism employed for exceptions, but occurring asynchronously with regard to guest system execution.

Contrasting exception handlers, guest systems only provide one common entry point for event notifications. Event multiplexing by guest systems is based on in-memory control information shared with the VMM: pending notifications are flagged by the VMM in a common bit field and subsequently cleared by the guest upon event delivery. A second bit field allows guests to mask individual event channels.

4.5 Split device I/O

Paravirtual, unprivileged domain (`domU`) kernels fully implement the concepts of virtualization-awareness and interface idealization discussed in section 3.3.4. For this purpose, paravirtualized guest kernels follow a "split driver" model: in addition to native I/O drivers for various components of the underlying hardware, driver domains implement *backend* drivers, which cooperate with complementary *frontend* drivers across virtual machine boundaries.

Present paravirtualized Linux kernels implement driver pairs for network and block (i.e. disk) I/O. For driver domains, a paravirtual PCI interface is provided. Others include a TPM (*trusted platform module*) and a video frame buffer interface. For the purpose of this document, only block and network I/O will be covered. The general architecture is shown in figure 4.2. It involves the following interfaces:

1. Interfacing of the frontend driver with the unprivileged guest system it serves. For this purpose, frontend drivers expose the same interface to their surrounding guest system as native drivers would for real devices.
2. Frontend/backend cooperation across virtual machine boundaries. Xen driver pairs commonly employ message passing in shared memory for communication.

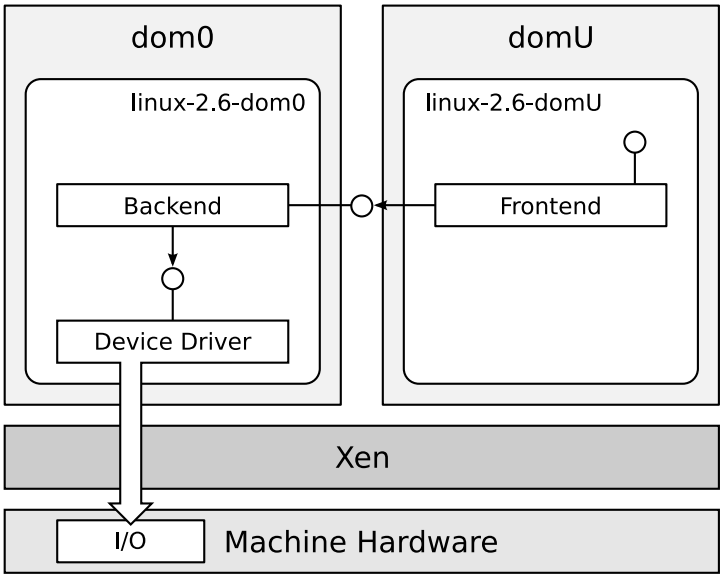


Figure 4.2: Xen split driver model on paravirtual Linux guest OS instances.

3. Interfacing of the backend driver with the guest system of the driver domain. Figure 4.2 depicts the relatively straightforward case where the backend device maps requests to a real device hardware.

There are exceptions to an immediate mapping from the backend device to a real one. Generally, the virtual device map may involve additional degrees of indirection. An example is network virtualization, discussed below.

Frontend/backend communication follows a simple request/response scheme. While more flexible message formats would be possible, frontends unilaterally generate requests, while the backend generates exactly one response for each request after processing it. The memory shared for passing messages is commonly structured as a circular buffer comprising a single memory page. Message transport on the ring then operates in a basic consumer/producer pattern. Request and response messages both comprise a fixed message format, and both request and response queues share a single ring.

Inspired by [13], figure 4.3 outlines a frontend and backend driver operations on a typical I/O ring. The sample shown corresponds to a ring state after seven requests, four of which have been served. Messages carry a unique identifier with each request which is reproduced in the associated response. Response messages

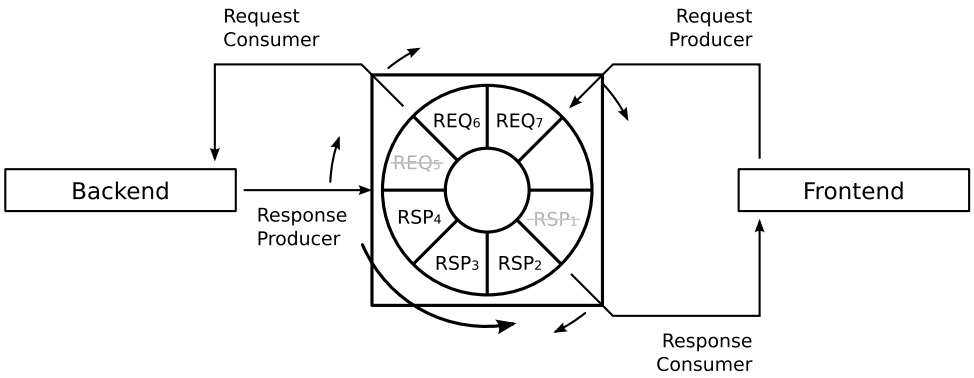


Figure 4.3: Message flow in Xen I/O rings.

replace request messages. Hence, message $\texttt{REQ}_5$ has just been consumed by the backend and becomes the position for a yet outstanding $\texttt{RSP}_5$ message. Message $\texttt{RSP}_1$ has been consumed by the frontend instance and will at some point be overwritten by $\texttt{REQ}_9$.

Note that message ordering is up to the protocol established by driver pairs. Most notably, requests do not necessarily have to be responded to in the order they were issued.

The ring structure as described above is sufficiently flexible to support both network and disk I/O device paradigms [13]. While message queuing on the ring buffer dictates a specific reception order, request identifiers enable out-of-order processing of requests by the backend, hence more elaborate I/O request scheduling on hardware disk controllers remains unconstrained [13, 32].

Message arrival between frontend and backend is signaled via interdomain event channels as introduced in section 4.4. Event notifications in turn may be performed decoupled from request generation. Multiple entries may be batched before performing a control transfer, which allows for trade-offs between I/O throughput and latency on the frontend side [13, 32].

4.6 Example: Network I/O Virtualization

The split network I/O architecture in Xen is shown in figure 4.4. Unsurprisingly, frontends implement network layer 2 devices to the guest operating system, one

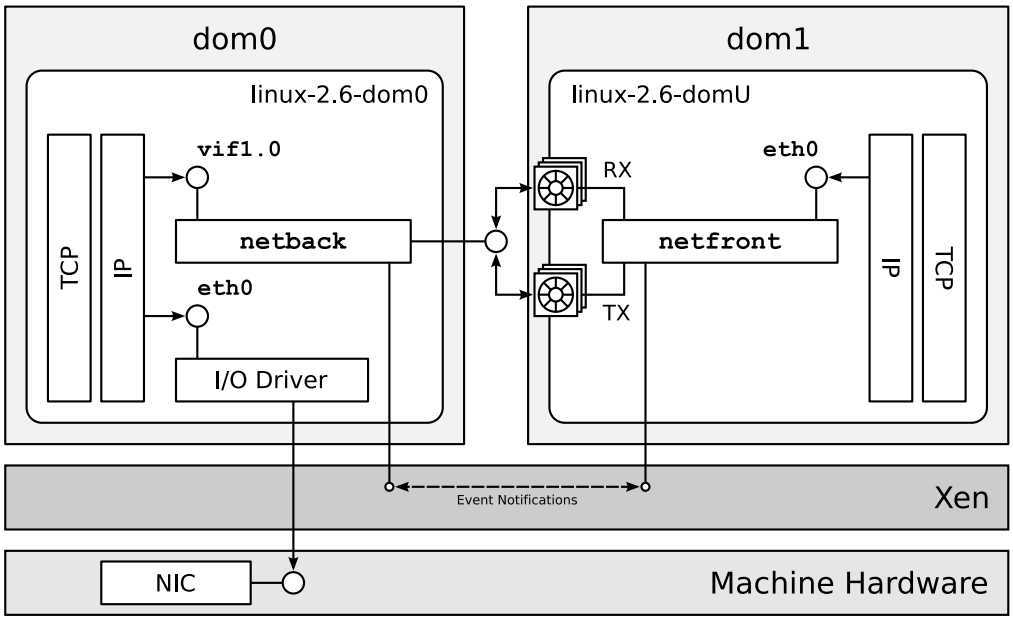


Figure 4.4: Xen network virtualization architecture.

per paravirtual network interface provided to the virtual machine. Network virtualization is described here because it features a more complex backend device interface than block I/O, and a larger degree of indirection regarding the map to an eventual network interface card.

In accordance with criteria outlined in section 3.3.4, ring messages carry no user data. Instead, requests reference dedicated out-of-band data buffers. All memory committed to device I/O has to be allocated by a respective frontend device, as opposed to the backend. The idea is to avoid performance interference of different frontend domains, which may occur if they had to be served from a common buffer set in the driver domain ("QoS Crosstalk") [13]. Notably, this requisite also holds for receive buffers. Instead of "delivery requests" passed by the backend driver, frontends send `RX` requests passing buffer references. Zero-copy transfers are either performed via DMA [32], or trading ownerships of individual buffer pages between backend and frontend (*page flipping*, see section 4.7).

There are two logical views on the network driver split worth considering. First, *architecture* of the interdomain device interface resembles idealization of a customary hardware network interface [13]. It comprises separate packet

queues for each transmission direction: one for packet reception[1] (`RX`) and one for transmission (`TX`). Separate I/O streams easily map to those of a modern full-duplex LAN interconnect. With block I/O, in contrast, mechanical disk architecture and respective I/O interfaces rather suggest a single queue and out-of-order processing.

Second, *correspondence* of frontend/backend pairs behaves like a point-to-point connection at the data link layer, i.e. a "virtual crossover cable". For this purpose, the backend driver registers virtual link layer devices within `dom0`; each an individual counterpart for a respective frontend device instance. Consistent with the overall system network architecture, packets are not transferred to any real device but passed up the network stack. This adds some level of indirection regarding the map to a real device, which in practice may not always apply, e.g. considering purely virtual network topologies comprising multiple virtual machines.

There are several variants regarding the virtual network beyond the unprivileged guest interface. Common ones are layer 2 bridging or respective routing or masquerading techniques performed at layer 3 [95].

4.7 Memory Sharing

In the Xen, memory sharing complements event channels in order to facilitate IPC. It is different from application-level shared memory such as POSIX shared memory [42] or file remapping techniques on various UNIX flavors, in that it is *asymmetric* and *transitory* [32]. Asymmetry means that all guest memory, whether shared or private to a VM, retains a single owner. Transitory means that a permission to remote access on memory can later be revoked by their owner.

Each domain carries a private *grant table* in guest machine memory. Entries include the remote domain receiving the grant, the page frame in question and read/write permission and present usage of the grant (updated by the VMM). The primary operation of a guest system on its tables is to grant or later revoke guest access to remote domains. The hypervisor itself maintains a cache of *active* grant entries, maintained in memory private to the VMM [32], counting the number of references maintained by grantees when referencing respective pages within their own virtual memory maps. A *grant reference* is an index into

[1] Directions denote the perspective of the frontend device, thereby those of the physical machine, i.e. packet *reception* implies a traversal of a packet from the backend to the deprivileged guest system.

a domain's (the granter's) own grant table, which then can be communicated to a respective remote domain (the grantee) to use.

Grants are described to some detail in [95] and [32]. For the purpose of this document, understanding of typical operations on the side of grantees shall suffice. All are performed via hypercalls, and take the grantee a grant reference from the granter, validated by the VMM:

Mapping Page frames may be *mapped*, i.e. incorporated into pseudo-physical and virtual address spaces of the grantee.

Transfer Memory may be traded, i.e. page frames may change (but never lose) ownership.

Granted memory mapped by a grantee is handled via *grant handles*. Generally, neither handles or grant references thereby need to expose machine frame numbers to either side. Paravirtual domains may refer to their page tables to determine frame addresses, but never need to. Similarly, grant references and handles provide for unique page identifiers with guests operating on shadow page tables as well.

4.8 Supplementary Interfaces

The above discussion of paravirtual I/O discussed only the predominant interface classes, presently network and block I/O. Within the overall system architecture of Xen/VLS, discussed in the following chapter, two additional interfaces exposed to guest systems need at least brief consideration:

System Console Domain 0 provides a single virtual console interface per paravirtual guest system. It serves a purposes similar to a serial line interface for remote-management in customary server machines: Boot messaging, system event reporting, and a terminal interface for administrative access.

XenStore XenStore is a service process running on `dom0`, which maintains a hierarchical database of system configuration items. It plays an important role in Xen's overall virtualization architecture. The store incorporates data serving administrative purposes, such as the number of created virtual machines and information covering their individual configuration. XenStore implements a notification mechanism. Client applications can subscribe to state changes of arbitrary subtrees in the store hierarchy. As soon as item stored change their value, notifications are emitted. But more importantly, this storage interface,

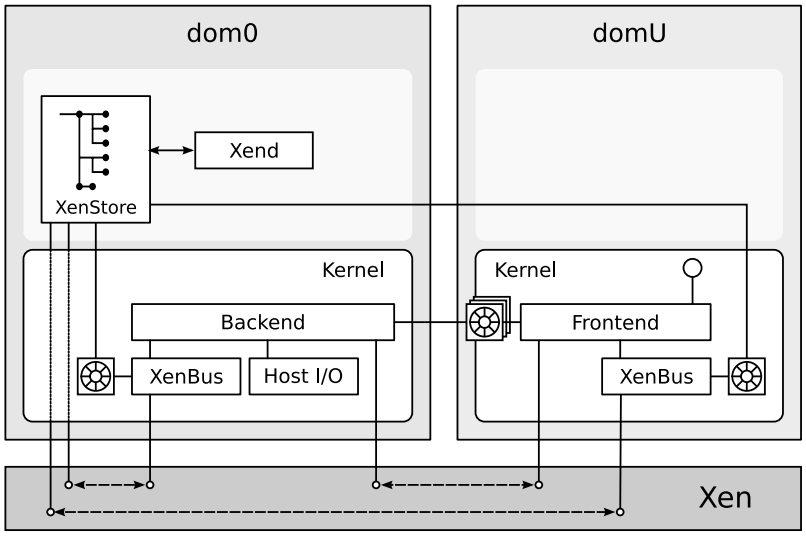


Figure 4.5: The XenStore directory service.

including the event notification mechanism, is carried across paravirtual guest system boundaries. The *XenBus* protocol is a pseudo-device driver interface integrated into all guest systems ported to the paravirtual machine interface.

In combination, these storage and system-level interfaces form a basic IPC mechanism for inter-domain communications. The use of XenBus is to some degree comparable to the purpose ACPI serves on modern x86-based machine architectures. Uses include connection setup between driver pairs, i.e. exchange of respective machine memory addresses, event channel port number and state change indications. Figure 4.5 depicts the reliance of driver pairs on XenBus for interface control operations and connection setup.

Generally, all I/O interfaces are built upon inter-domain event channels and and circular I/O buffers similar to the ones described in the foregoing section. However, compared to practical data throughput and latency requirements and the resulting amount of work spent by system designers on optimizing throughput, interfaces beyond block storage and network I/O play a significantly lesser role for the purpose of this thesis. Chapter 5 will touch the above components only briefly and where appropriate.

5 Xen/VLS Architecture

Interplay of the Xen hypervisor and accompanying driver domains resembles common microkernel design. Deployments in server configurations typically operate concurrently on a shared-memory multiprocessor system, with intercommunicating paravirtual machines separated into different address spaces. Such an architecture is fairly different from traditional approaches to deterministic replay and semi-active replication. So far, deterministic replay lent itself rather to monoloithically structured host system environments (e.g. [94, 97]).. With Type I VMMs, classic I/O virtualization would typically be integrated into the hypervisor.

The issue of arbitrating concurrent system components for the purpose of controlling the interaction of target virtual machines with their processing environment has been a predominant factor in the design of Xen/VLS, influencing large parts of the overall architecture. The system has therefore ultimately been split into two separate components, both of which are part of the core VMM:

SMA (*Synchronous Memory Access*) controls shared memory updates induced to target guest systems by various other system components.

The SMA facility represents a fairly general solution to achieve piecewise deterministic execution under address space separation and thread-level parallelism. It is demonstrably capable of managing both inter-domain communications as well as interfacing guest systems with the hypervisor.

VLS builds upon SMA to control and log the interaction of target guest systems with their processing environment.

Conceptually, VLS mediates between the SMA facility and a central log reception facility built into any sufficiently privileged domain (typically `dom0`). The process includes generation of an appropriately formatted stream of event log entries and control over replica execution in face of congestion.

The remainder of this chapter is organized as follows: Sections 5.1 to 5.3 summarize individual subsystems of the virtual machine interface presently provided to paravirtual guest systems, under the criterion of effect on control and data

flow. The first step is therefore identification of any potential sources of non-determinism affecting guest execution.

Section 5.4 lists additional considerations to be taken into account, such as the amount of change imposed on the overall system architecture. Section 5.5 includes some alternative designs to achieve determinable I/O. One is event migration, a more general concept from which SMA was ultimately derived. Section 5.6 will describe *SMA Channels*, which interfaces the SMA facility with guest systems hosting I/O backends as described in section 4.5.

Section 5.7 will summarize the techniques employed within the VMM in order to synchronize arbitrary memory accesses with guest execution. Different from the the comparatively simple producer/consumer protocol by which split I/O drivers communicate, some of the synchronization primitives employed by the core VMM, such as grant table management and event channel activations, rely on atomic read-modify-write operations to synchronize with the VMM, which requires different

Section 5.8 turn from Xen's SMA layer to the VLS component, starting with the structure of the resulting log format. Furthermore, it describes the transport mechansism built into Xen, emitting the captured stream of event log entries. The present VLS implementation forwards this data stream for further processing to `dom0`.

Moving from arbitrary memory accesses to reliance on above SMA channels requires explicit support from backend drivers. Some necessary changes on the backends must be programmed on a case-by-case basis. Section 5.9 will study concrete effect on an exemplary implementation based on Xen's present block storage virtualization specifically under the aspect of maintainability.

Finally, section 5.10 will evaluate performance of the implementation. The results were derived from customary I/O workloads, benchmark programs and some synthetic tests. They cover overall event log bandwidth measured for different classes of system utization, as well as the impact on execution time.

5.1 Processor

Interaction between the Xen hypervisor and paravirtual guest systems can be summarized as follows:

- Hypercalls issued alter the processing environment of a given VM.
- Memory updates performed by Xen, on a dedicated page shared with the guest (`shared_info`, discussed below). The shared memory interface

includes the abstract time sources described in section 4.3.

- Event channel activations. These comprise `shared_info` memory updates as describe above. Additionally, however, they immediately affect control flow in the target system, by transferring execution to a respective event handler.
- Emulation of privileged instructions.

No sources of non-determinism assumption are attributable to hypercalls. This should come at no surprise, since hypercalls do not incorporate I/O resources, but only validated access to static processing resources directly managed by the VMM. The following section introduces `shared_info` contents and its effect on computational state. Section 5.1.2 will then touch on non-deterministic instructions.

5.1.1 Shared Information

Consistent with the general concept of idealized device interfaces mapped to regular memory (section 3.3.4), time sources are not represented within the hypercall interface. The initial memory allocation of any paravirtual domain includes one dedicated page of memory called `shared_info`, named after the data structure dictating its layout. The `shared_info` structure is divided into one global section, i.e. carrying information affecting the whole domain, and one or more VCPU-specific sections (`vcpu_info`), one per virtual processor. Presently, Xen guest systems are limited to a theoretical maximum of 32 processors. The page incorporates a considerable fraction of the machine interface introduced in section 4.3:

- Per-domain wall clock time, as a virtual real-time clock measuring present time and date. Furthermore, a presentation of system time per virtual processor, in nanoseconds since boot.
- A bit vector of pending event channels, i.e. those on which notifications have been sent but not yet processed. Accompanying fields include an index to above bit vector, as well as global and per-port event masks (idealized counterparts to the `IF` flag and a programmable interrupt controller, see section 3.1.1). The event map involves both global domain and virtual processor state.
- Architecture-specific information, e.g. on x86 systems a virtual representation of the `%cr2` (faulting address) register.

Clearly, control transfers through event channel activation need to be replayed consistently across repeated execution of the system, as do respective indications in the shared info structure. Similarly, system time updates are performed by the VMM on a regular basis. In contrast, the x86-specific fraction of `shared_info` does presently not include any sources of non-determinism.

5.1.2 Non-deterministic Instructions

The last interface element listed above to connect guest systems with the VMM is instruction emulation. With paravirtual machines, non-deterministic processor instructions will play a lesser role with regard to implementation effort when compared to shared memory, since no direct device I/O is involved. Nonetheless, processor timer I/O needs consideration. While real-time applications are not of concern for the purpose of this thesis, modern computer systems provide (and expect) fine-grained time sources. Some POSIX elements [42], such as the `nanosleep()` function, incorporate timings at nanosecond granularity. Since a desirable degree of precision cannot be efficiently provided solely with timer values readable from shared memory, guest systems perform time extrapolation instead. On x86 systems, the `rdtsc` (*Read Time Stamp Counter*) [45] instruction provides a processor cycle-accurate timing source, which can be scaled appropriately to extrapolate system time at nanosecond-precision.

The `rdtsc` instruction is trivially trapped by adjusting a respective control register flag, tracing then performed by re-execution in the VMM. In the event log format developed as part of this thesis, traps due to `rdtsc` are part a class of execution events named *fixed faults*. It will therefore be briefly revisited in following sections.

Within the prototype developed for evaluation (section 5.10) no instructions beyond `rdtsc` are traced. It should be noted that there are more instructions of potential impact. One field are processor capability inquiries, such as performable with the `cpuid` instruction. Such instructions matter as soon as execution replay across heterogeneous system boundaries is considered.

5.2 Machine Memory

With paravirtualization, potential non-determinism due to the machine memory image underlying the guest system requires thorough consideration. As pointed out in section 3.3.3, the physical address space visible to paravirtual guests is typically fragmented.

Clearly, exposure of non-contiguous, arbitrarily fragmented address ranges is in violation of the PWD assumption, as it may easily affect traversal of such address ranges during page table alteration and machine memory management. To exclude such effects on execution replay, a linear machine address map is needed, such as the one provided by shadow page tables.

The Xen VMM includes a number of different memory virtualization strategies. In support of full system virtualization, shadow paging is one of them. Indeed, the available shadow paging mechanism can also be readily applied to paravirtual guest systems.

In practice, however, shadow paging for paravirtual domains found few uses beyond academic and research projects such as Xen/VLS. According to future roadmaps indicated by VMM developers, the implementation is likely to vanish in upcoming releases of ported guest systems such as the ported Linux kernel maintained as part of the Xen source distribution.

As described in section 3.5, one long term alternative hardware assist for Xen's paravirtual machine interface by Xen are likely nested page tables. The approach has additional merits, such as an effective solution for privilege and address space compression. Seeking deterministically replayable paravirtual VMs, it would elegantly remove the issue of a non-deterministic memory map. At the time of this writing, nested paging is a comparatively new feature in x86-based ISAs. The first revisions are only available to a broader public since release of the AMD K10 architecture [7] in late 2007. Variants from Intel will follow.

Due to present unavailability and the comparatively large change required to the hypervisor in order to combine paravirtualization with hardware assisted ISA-virtualization, the performance implications of enforcing linear address spaces have not been evaluated in section 5.10.

5.3 I/O

I/O comprises input and output. To achieve piecewise determinism, output performed by replica guest systems is of no concern. With semi-active replication in a mere crash failure model, a minor exception is event log stabilization during output commit as discussed in section 2.2.7. The task of logging and replaying guest input from virtual I/O devices and the specific issues involved when performing this task on Xen.

As learned from chapter 4, Xen's I/O model is commonly based on circular message buffers and event channels. A fully piecewise deterministic system

typically comprises the following interfaces:

- Block Storage
- Network Interfacing
- Console I/O
- XenBus

Throughout this chapter, only frontend devices under the PWD model will be considered. Given the largely symmetric shape of the communication path taken in circular buffers, techniques equivalent to the ones pursued here could be applied to backends. Such alternative setups are not considered here, since customary server workloads on non-privileged machines would gain no obvious benefit from them.

Differences to monolithic systems are due to the concept of driver domains, contrasting I/O resource management fully and directly embedded into the hypervisor. In summary, effective generation of replayable input traces on paravirtual Xen guest kernels thereby represents a significantly larger challenge than with monolithic virtualization architectures. Xen's I/O model has the following properties:

Asynchronous State Changes State changes in device status affect replica state directly, and asynchronously. Like with all asynchronous events (see section 2.2.5), a monitor needs additional measures to maintain consistency across replicas.

Context Separation Guest input may be originate from different physical processors than requests are issued. In contrast, determination of the precise system state (i.e. up to the granularity of individual instructions executed) where updates occur is only available when interrupting execution.

Address Space Separation Updates originate from non-global address spaces (those of the driver domains). Not only needs control to be moved to the target execution context, but data as well.

With the exception of address space separation — due to the VMM always being present in global virtual memory — the same attributes apply to the `shared_info` structure described in section 5.1.

5.3.1 Programmed I/O vs. DMA

System I/O comprises control and status inquiry for device functions, as it is typically concerned with transport and management of raw raw user on behalf of

upper system layers. Xen's I/O control architecture is built upon shared memory and guest system notifications as the lowest-level primitives. The resulting paravirtual machine interface thereby implicitly borrows from a communication model well-established in modern I/O architectures: DMA transfers, originating from devices to host system memory. On the other hand, the accompanying control interface is much different from those employed in native device I/O, thereby from virtual I/O as would be the case with traditional Type I or II VMMs in full system virtualization. This section will discuss the differences before discussing alternative solutions to the problem.

I/O device architecture typically comprises two components: an I/O controller, interfacing the device to the host system, and the actual device, such as a network interface. Control of a physical processor over I/O devices is performed by accessing a respective device's I/O controller over an intermediate I/O bus. At the bus level, a typical device driver action may be to issue commands by writing to a control register. Conversely, status information is inquired by reading the same or a companion status register.

An I/O controller interface may be memory-mapped, i.e. colocated in the machine physical memory address space), or port-mapped, i.e. accessible via a dedicated I/O address space [64, 75]. The latter is commonly the case with x86 processors, deriving from early personal computer systems.

When issued for control operations as well as user data transfers, either variant constitutes *programmed I/O* (PIO), i.e. data exchange initiated by the host processor. With programmed I/O, state changes to system software interacting with the system occur synchronously, by execution of load/store or `in`/`out` operations. State changes induced by I/O are performed in context of the executing replica.

`In`/`out` operations are privileged and therefore trap to the VMM. Tracing programmed I/O could therefore be performed with trap-and-emulate techniques. Similarly, I/O memory can be emulated transparently via shadow paging [75]. However, different from many existing machine interfaces, guest I/O in Xen is not programmed. The driver domain model implies input operations performed through externally induced changes in machine memory.

Similarly, customary hardware I/O controllers may transfer data via DMA (*Direct Memory Access*) [64], in order to relieve host processors from the task of executing large amounts of input or output data. The difference to physical I/O is that DMA transfers are usually only performed for user data, such as network frames or blocks of disk storage, while Xen's asynchronous I/O ring model rather follows the DMA paradigm.

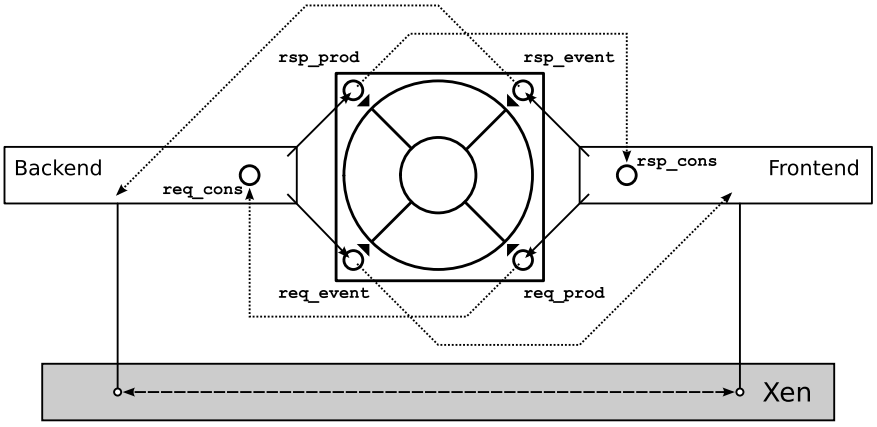


Figure 5.1: Data flow between split device drivers in Xen.

The split device driver architecture employed by Xen comprises control and status interface in shared memory. Different from native device I/O, control and status information is carried on circular buffers in shared memory: control operations are requests issued by frontend driver instances, status is returned via response messages (see figure 4.3). Compared to native systems, guest state changes due to status input rather resemble DMA transfers, since messages are generally issued without synchronization.

5.3.2 Generalized I/O Ring Protocol

Network and block device drivers included with the Xen hypervisor share a common framework for the implementation of I/O ring buffers. Both backend and frontend drivers access the ring structure via a set of C preprocessor macros. Request and response message types are variable.

While usage of this framework is not strictly necessary, it represents the standard facility for the purpose given. Most operation on the shared I/O ring page require careful attention to memory access ordering for individual loads and stores on shared ring indices as well as message contents. Care has to be taken both within the code emitted by the compiler as well as regarding potential reordering performed by an underlying machine architecture. The header files include the necessary memory fencing, thereby taking care for correctness and portability in face of all platforms presently supported.

Figure 5.1 shows the data flow through shared and private machine state im-

plemented by the framework. As explained in section 4.5, request and response queues share a common ring. Message consumer indices ({`req|rsp`}_`cons`) are held private by each side, only respective remote producer indices ({`req|rsp`}_`prod` are written to shared memory, to be read by consumers. Each field has exactly one reader and one writer. Data consistency is trivially maintained without locks, as long as a common memory access ordering is maintained.

Two additional indices deserve closer examination. Upon notification via an associated event channel, drivers will typically consume all pending messages. The general ring model employed by both network and block I/O interfaces allows a message producer to detect that consumption of queued messages on the remote end remains in progress. For this purpose, each consumer side maintains an additional event mark index ({`req|rsp`}_`event`). Producers omit notification of the remote connection end until the producer index passes the the notification mark indicated by the consumer, thereby avoiding gratuitous notifications and calls to the VMM. This feature is called *notification hold-off*.

The individual state where an update is performed by a driver domain has large potential impact on the execution path taken on a frontend side:

- Like all guest kernel operations, ring processing is preemptible by the VMM, leading to arbitrary, thereby non-deterministic interleavings between memory accesses performed by a respective frontend and backend.

- On multiprocessors, backend and frontend perform ring updates concurrently. This contributes much to the maximum device throughput which can be accomplished on such systems, especially given address space separation: "World" switches between domains carrying front- and backends imply larger latencies than they would in monolithic systems, due to excessive TLB flushing.

- There is no one-one correspondence between issued messages and notifications, e.g. a consumer may find more messages pending than were available when the producer sent the notification.

There is a multitude of different candidate techniques to achieve synchronization, as well as additional considerations to be taken into account, such as the amount of change imposed on existing code. Additional considerations are discussed throughout the following sections. In summary, there are multiple alternatives to determine memory accesses within the given I/O interface

5.4 Design Considerations

Considerations regarding the design of deterministic I/O replay based on the Xen architecture can be dealt with in terms of two separate issues:

Replay Consistency Ring access must be executed consistently on both the leader and any follower machine instance. If the split driver architecture shall be maintained as is, accesses to shared memory must be be synchronized with frontend execution in all replicas. In order to control replica execution accordingly, memory accesses need to be coordinated between the backend and either the frontend driver, or (transparent to frontends) with the monitor, or both. Due to the degree of context separation and deprivileging, the desired level of control cannot be performed by guest systems alone.

Impending negative impacts on on frontend/backend performance should be avoided, or at least minimized. Most notably, this includes maintenance of the presently degree of concurrency achieved among individual driver pairs.

Event Log Transport Guest input needs to be logged, thereby aggregated into a single (ordered) sequence of log entries. This implies not only generation of events, but log transport to a sufficiently privileged entity, such as a process running on `dom0`. This is complicated if guest input is initiated from entities residing in multiple, separate address spaces, as is the case with Xen driver domains. The path taken for transport of events may have impact on the amount of change imposed on existing backend drivers, as well as on performance.

Generally, data channels established between individual backend instances and `dom0` would represent a larger change to drivers than e.g. a common interface fully integrated into the VMM.

As with all systems, the protocol architecture for various device classes is subject to incremental evolution and refinement over time. An example of this process are recent performance enhancements achieved by a fundamental redesign of the memory sharing facility employed for bulk data transfers on the network interface [71]. Furthermore, new interface classes beyond presently employed abstractions of network and block storage I/O evolve over time[1]. Both evolution and reformation demand for two important properties of both leader and follower modes:

[1] One such example presently under development is an SCSI-conformant I/O driver protocol for unprivileged guest systems [41].

Maintainability demands that changes to present software should be small, both on the side of the VMM as individual backend or frontend drivers. This requires that neither consistency nor log transport should be unlikely to interfere with future enhancements. Similarly, the effort it takes to augment new I/O interfaces should remain small.

Generalization demands that changes to the VMM in support of deterministic replay should not be device or device-class specific, in support of an arbitrary, potentially growing number of interface types without further extensions. In other words, a workable solution should separate *mechanism* from *policy* [82].

Maintainability consideration rule out fundamental redesigns of Xen's I/O virtualization. Such architectures may indeed receive consideration, as they are found in classic system VMM design. I/O virtualization integrated into the VMM provides for simpler log generation and transport, since all driver code then shares a single, global address space. An integrated architecture would typically include not only backends but colocated device drivers, which is commonly the case with many present Type I (e.g. VMware ESX [85]) and any Type II VMMs in monolithic operating systems (e.g. VMware Workstation [74], KVM [54]). In contrast, section 4.2 identified reasons why the decomposed architecture is beneficial and hardly subject to general reconsideration in the future.

Traps and emulation of programmed I/O instructions issued by replicas are traced and replayed more straightforwardly than shared memory accesses. Programmed I/O would typically apply in full virtualization. However, returning to a port-level programmed model would counteract the motivations behind present paravirtual I/O interfaces discussed in section 3.3.4, such as high-level command and status interfaces and zero-copy I/O by memory sharing. But consistent with paravirtual guest design, dedicated hypercalls may be issued in order to access device state consistently, as long as a sufficiently generalized interface can be formulated.

To some degree, extensions to the frontend driver architecture, to be described in section 5.6.3, will promote the programmed I/O paradigm, but only in a fashion compatible with original architecture.

Under the criterion of interface *generalization*, replica input cannot be logged in terms of request and response packets issued to a respective ring structure, as all elements involved are specific to the class of a respective device. However, replica input at a next lower level of shared memory updates to a small number of candidate pages is sufficiently general to support all existing driver pairs in

both trace and replay mode.

5.5 Design Alternatives

5.5.1 Virtual I/O Memory and Protection

Seeking coverage of all possible alternatives for maintainable replay consistency, this section considers the feasibility to enforce replica-driven synchronous I/O transparently. Page protection could be used to synchronize executing replicas with externally induced memory updates. Without taking intimate knowledge of the frontend driver operation into account, shared memory input to replicas may then be performed as follows:

1. The backend requests protection of the shared memory region from all frontend accesses, i.e. protect all replica pages mapping a respective page frame in the replica's shadow page tables.
2. The backend performs all updates to the ring. If the VCPU executing the frontend faults on the ring during this step, the VMM will suspend execution until completion of step 3.
3. The backends unprotects the ring from replica guest access. The frontend may resume if it was suspended during step 2.

What this technique effectively pursues are traps of I/O memory accesses, as would be performed in full virtualization. It enables consistent replay; the logical point during execution at which a memory update becomes visible to the frontend is upon completion of step 3. If the replica is not blocked, it can be determined within the monitor, by interruption of replica execution in order to remove the page protection in guest context.

Utilizing memory protection would require only small changes to backend drivers, and none to frontends, which accommodates good maintainability. Furthermore, only a subset of ring accesses issued by the frontend would need to be trapped. However, a performance-critical one, considering access interleavings during concurrent execution of front- and backend.

Futhermore, the above algorithm will likely not scale when frontends and backends operate concurrently, as will be the case for applications with high bandwidth demands and corresponding access frequency, such as network I/O rings. This is due to several facts:

- Changes to page tables of the target system require a TLB flush on any processor executing the target system. This operation cannot be entirely

performed by the initiator, hence costly inter-processor synchronization needs to take place.

- Unselective TLB flushes are costly, since all virtual-to-physical address translations in the processor working set are to be discarded.
- Updated page regions are small, while the protected memory range potentiates collisions with unrelated (deterministic) ring accesses on the frontend side.
- Changes to the protection bits set on present pages cannot be implemented efficiently unless a reverse mapping from page frames to page table entries is maintained. The Xen hypervisor presently does not provide reverse mappings.

Other issues remain. As an example, mere memory protection lacks a solution to efficient log generation and transport yet.

Transparent memory access consistency via page protection may be a viable alternative, e.g. if trap-and-emulate represents the normal mode of operation, or access patterns are unlikely to suffer from excess collisions. For fine-grained, frequent and concurrent access patterns, such as those performed on Xen's I/O rings, controlling memory update on the initiator side represents the more promising approach. This concept will be pursued across the remainder of this chapter.

5.5.2 Deferred Processing

Early experimental versions of Xen/VLS were limited to tracing and replaying interaction between guest systems and the hypervisor only. The initial design is introduced here because it explains the relative importance of the later-developed SMA mode in order to keep changes to existing monitoring facilities small.

Compared to inter-domain I/O, most updates to shared information are relatively easy to trace. First, no address space separation applies. The hypervisor is mapped to the top 64MB global virtual memory shared by all address spaces. Second, the number of distinct items kept in shared information which are critical to the PWD assumption is well-defined and comparatively small, as listed in section 5.1. An early experimental tracing facility comprised only control over system time abstractions and event vector updates, as well as `rdtsc` emulation.

Updates to system time in Xen [13] are relatively frequent, performed e.g. upon each new time slice dedicated to a respective VCPU. Wall clock time, in

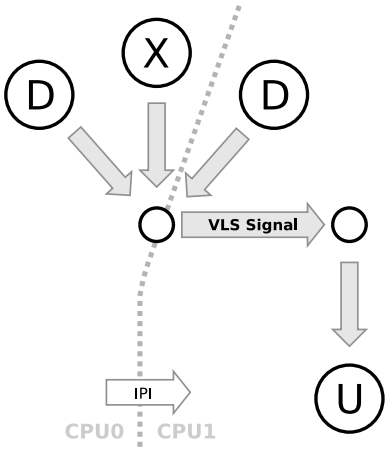


Figure 5.2: Inter-processor event migration. VLS *signals* constitute software event processing. Hardware IPIs migrate events across different CPUs.

contrast, is typically kept in synchrony across all virtual machines. Changes to wall clock time will only occur when `dom0` readjusts its virtual real time clock. However, such updates may originate asynchronously from arbitrary processors.

In order to control and log updates to an interrupted guest system at the granularity of individual guest instructions, the VMM timer subsystem was altered accordingly. For this purpose, an early prototype of the VLS extension augmented the VMM with a simple event processing mechanisms, called *signals*. Within target system execution, a signal is nothing more than a particular work item marked pending within the VMM, such as a necessary update to system time. Processing of pending signals is made a common element on the return path from any control transfer between a guest system and the VMM.

VMM code executing on arbitrary processors may emit such signals to a given VCPU. Figure 5.2 depicts the event processing scheme accordingly. All accesses to guest system state are thereby carried into the processing context of the target system, independent of they originate. Ultimately, it is the state change performed during signal processing what is traced and replayed by the VMM.

Conceptually, signals derived from a class of deferrable functions called *softirqs*, originally developed for the Linux 2.6 kernel [18] and equally present in the Xen VMM[2]. Similar to softirqs, signals are processed just before transferring control from the VMM back to the guest system. Softirq processing includes tasks such

[2]Arguably, a large fraction of architecture-specific code as well as some fundamental kernel facilities present in the Xen VMM were originally derived from the Linux kernel.

as world switches performed by VCPU scheduler. The main difference between the concept of signals and softirqs is that work items carried through signals are bound to a virtual processor rather than a physical one. If a virtual processor is migrated to a different CPU, pending signals migrate as well.

Events migrate across different physical processors through inter-processor interrupts (IPIs). Event processing on the receiver side is always performed when running on behalf of the VCPU signaled, in case of IPIs on the return path taken from the IPI handler. Event processing allows control over any interaction with the from arbitrarily interrupted guest context. Examples are synchronous events such as hypercalls or exception handling, as are inter-processor events. Truly asynchronous events can thereby be determined and tagged with instruction counter readings. Synchronous versus asynchronous events will be revisited in section 5.8.5.

VLS signals already constitute much of the concept an *event* within the PWD model when carried forward to virtual machines: arbitrary changes in state of guest systems and their runtime environment. *Synchronization* then establishes the necessary binding between event execution and execution of the target system.

The downside with signals is that they migrate state changes by migrating their *execution*. For the remainder of this document, the overall technique to carry work items to a specific processing context will be referred to as *deferred processing*. As an example, a function performing guest system time updates would typically execute one of three different versions:

1. The original function, for regular guest systems not subjected to trace and replay, executes unconstrained.
2. In VLS *Trace* mode, the update will not be performed immediately, but signaled to the target VCPU. Deferred signal processing will execute the original update and generate an event log entry accordingly.
3. In VLS *Replay* mode, events are solely replayed from the event log. A typical implementation may generate signals, but discard them[3].

The result represents a workable solution, but one comparatively hard to maintain. Changes due to different modes of execution largely affect the original monitor. To achieve piecewise deterministic execution, indirections either via function pointers or conditional expressions need to be added. Keeping the VLS component separate from core VMM code makes control flow additionally hard

[3]Deterministic replay in semi-active replication will typically require signaling to be performed, in order to avoid event loss during failover.

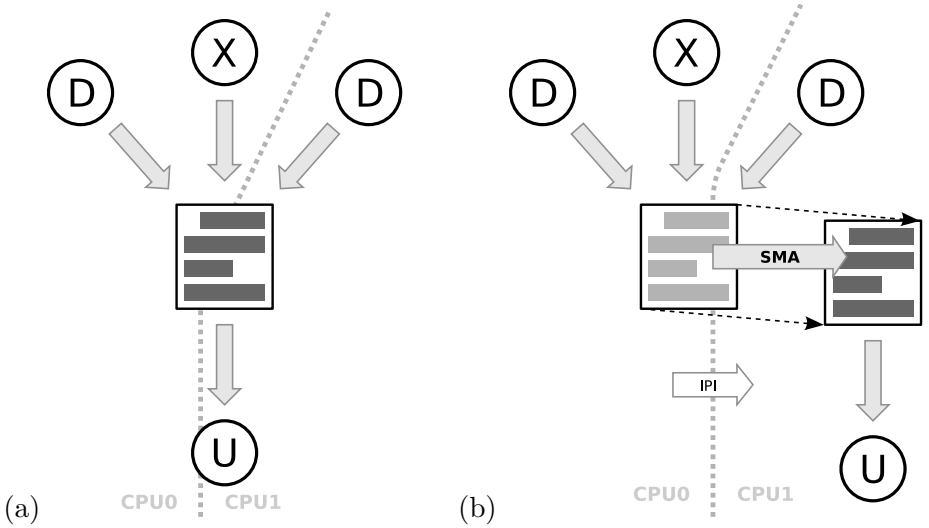


Figure 5.3: Original (a) memory accesses in Xen vs deferred, double-buffered (b) SMA updates.

to follow.

5.5.3 Deferred Memory Updates

SMA builds upon a processing scheme derived from the event model covered by signals, but does not migrate system execution into guest system context. Instead, state changes will remain to be performed largely unconstrained on the originating processor. In order to avoid inducing non-deterministic memory access interleavings with concurrently running guest VCPUs, they are not committed to guest state immediately, however. Instead, a large fraction of updates performed will buffered in memory private to the VMM. The state change is transferred in memory. Hence, the technique is said to replace deferred processing with deferred updates. The refinement follows a number of observations:

1. Shared memory is the predominant interface both within the VMM and external I/O. The same concept can be applied to both worlds, overcoming the issues with `shared_info` updates outlined in the previous section.

2. Data flow in shared items is typically unidirectional, i.e. most fields kept in shared memory have only one reader and one writer (see figure 5.1). Guest system *input* is written externally and not modified by the executing target VM.

Figure 5.3 shows external events as updates to page frames in guest system memory, where Xen (X) and driver domains (D) access memory of a guest system (U). A simple idea, but of considerable simplicity is double buffering of target page frames. Conceptually, shared memory is "unshared" by splitting individual pages into two separate instances. One instance, the back-buffer, is the one external input is redirected to. The primary instance is referred to by the guest system. Moving altered guest state from the back-buffer to the primary will only be performed under the constraints of the piecewise deterministic execution model, i.e. on a determinable instruction boundary.

Inter-processor event migration operates similar to deferred processing above. But in contrast to the original concept of signals, a work item migrated to a particular VCPU does not migrate execution of the original event. Instead, the result of executing it may be represented as a a tuple (m, o, l, d), where m identifies a page frame in the replica's pseudo-physical memory to be modified and d a byte sequence of length l, subsequently written to offset o in m. This level of abstraction constitutes the one chosen for the SMA layer: Instead of tracing state changes in terms of initiator activity (e.g. *Timer update to a given value*), the event log comprises a sequence of anonymous updates in shared memory limited to the presentation above. The tuple (m, o, l) will further on be referred to as an *I/O vector*.

Execution of original code can thereby remain largely unaltered. A synchronous memory access, as shown in figure 5.3 performs a number of operations in guest machine memory, such as copying data between two pages. The amount of change imposed to original VMM code is thereby reduced to conditional redirection to the back-buffer. Work items migrated to a virtual processor generate SMA commands, which are then to be executed by an SMA component integrated with the hypervisor. The SMA component integrated with the Xen hypervisor is not limited to copies in double buffering. The set of available operations in guest memory will be discussed in the following section.

It should be noted that this concept can *not* be applied to the entire machine interface. One exception to rule 2 are event channel activations and their representation in shared memory, as outlined in section 5.1.1. Another are grant table accesses. Both will therefore be revisited in section 5.7.3. Still, the effect on the monitor has been found to be much smaller than with event signaling alone. Most importantly, buffered memory updates committable to target memory in a deferred fashion are sufficient for all paravirtual device I/O performed by driver domains.

5.6 SMA Channels

In Xen/VLS, deferred updates are the primary mechanism employed by driver domains in support of deterministically replayable memory accesses. This section describes its integration into the paravirtual machine interface. The architecture features a small extension to the PVMI, following the assumption that while replica input remains to be initiated and controlled by guest systems, both update consistency and event logging are best performed by a single, central facility integrated the hypervisor. For this purpose, the Xen VMM is extended with a pseudo-device, called *SMAC*, performing updates to shared domain memory on behalf of arbitrary driver domains.

The device functions performing memory accesses functionally resemble a hardware DMA controller (DMAC). A DMA controller comprises a number of individual DMA channels, performing memory transfers between host memory and I/O device memory on behalf of a host processor [24]. This image refers to DMA in non-mastered bus architectures, as would be the case with some legacy-derived components found in IA-32 and derived system architectures [44]. This thesis is rather concerned with memory transfers are rather host-initiated. Host-integrated DMA engines recently spawned interest in system research, as an alternative to TCP offloading in 10-Gigabit networking. Similar opportunities for future research in scope of deterministic VM replay will be discussed in chapter 7.

For the purpose given, DMA-like memory transfers are to be performed consistently at a determinable instruction boundary during target guest execution. The abstraction chosen has therefore been dubbed *SMA Channel*.

5.6.1 Programming Interface

An SMA Channel exposes a programmable pseudo-device interface to driver domains. Each channel is controlled with request messages on a separate I/O ring associated with it. Different from customary driver architecture, the ring is shared with the VMM. Response messages are generated to indicate command completion and indicate transfer success or failure[4]. The present SMA channel implementation provides a sufficiently flexible interface comprising only four different commands[5].

[4] Failures are typically fatal and indicate program errors. Correctly initiated SMA transfers are not meant to fail, since no hardware beyond the host memory interface is involved.

[5] Note that while the notation shows command mnemonics and parameter names in function syntax, actual encoding and calling conventions are left out for brevity.

`SMA_CMD_target(dst)` Set destination page *dst*. The channel *destination* refers to a target page shared with a remote virtual machine. It is the only persistent element of channel state, an implicit parameter to all subsequently issued commands until the destination is reset to a different page, or the channel closed.

Like all shared memory in Xen, the destination is subject to validation and security enforcement by the hypervisor. Consistent with memory sharing concepts underlying Xen, different types of destinations result:

- Machine frame numbers referring to a page frame in (physical) machine memory. The ability to target machine frames directly is only available to sufficiently privileged domains (presently only `dom0`), but a necessity in support of some fundamental services, such as paravirtual console I/O and the XenStore interface, which do not use page grants.
- Grant handles and references, as introduced in section 4.7. Whether handles or references are to be used depends on the driver architecture and is largely a matter of convenience. Practice has shown that both variants can be of practical use. When used as SMA targets, both variants are additionally tested by the VMM according to their original semantics [32].

The majority of subsystems, such as block and network I/O drivers, utilize grant handles and references exclusively. Ultimately, all variants resolve to a single frame of machine memory owned by respective remote target domain.

`SMA_CMD_copy(src, offset, len)` Copy memory region, where parameter *src* refers to a page frame owned by the transfer initiator.

Memory copies facilitate back-buffering of up to page-sized amounts of data at the initiator. Since memory sharing is always performed at page granularity, the `copy` operation always assumes source and target machine memory to be congruent, i.e. the channel copies *len* bytes of data from offset *offset* in page *src* to the same offset in the present channel destination.

`SMA_CMD_write(offset, len, data)` Write datum to memory, where parameters *len* and *offset* have the same meaning as with the `copy` command. Presently, the parameter *data* may carry up to 4 bytes, as indicated by *len*.

In practice, writes facilitate updates to producer and consumer ring indices in shared memory, which explains why 4 bytes (a 32-bit integer) have

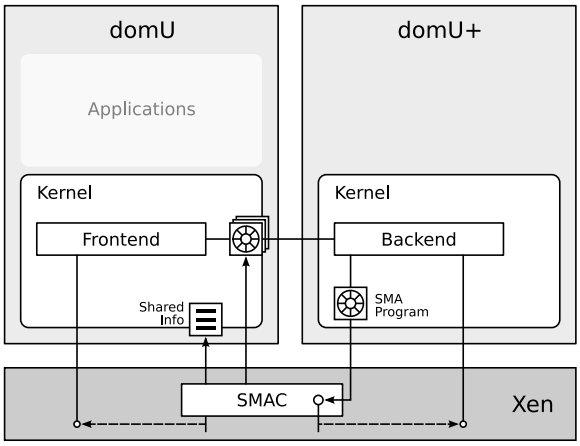


Figure 5.4: SMA Channels

proved to be sufficient.

Note that, different from memory copying and sampling, write transfer data provided as an immediate value encoded within the command. For some usage patterns discussed below, it is an important alternative to copying, since immediate values are not in danger of being overwritten before command completion if initiators and the SMAC proceed concurrently.

`SMA_cmd_sample(offset, len)` Sample information from the channel destination, where parameters *offset* and *len* have the same meaning as above.

Memory sampling is different from the above commands, as it does not modify destination memory, but only instructs the SMA channel to *read* the fragment in question.

Conceptually, sampling enables drivers to perform continued direct memory accesses to target domain memory, if the access is deemed safe to not affect control flow under the piecewise deterministic execution assumption.

Figure 5.4 shows the resulting communication pattern, comprising a piecewise deterministically executed guest system (`domU`) and and a driver domain (`domU+`). All changes to `domU` state are performed through one or a number of channels allocated from the SMA controller. This includes not only the memory shared between the backend and frontend devices, but control over event channel activations and the port map embedded into the `shared_info` structure as well. SMA Channels are not standalone resources, but implemented as an

extension to the event channel interface part of the Xen PVMI. More precisely, SMA channels are exclusively bound to *inter-domain* event channels (see section 4.4).

Conceptually, SMA channels aggregate the notion of shared memory access with the event channel interface. SMA channels thereby break with one design element of the PVMI: while both elements are typically used in combination, especially in the split driver model, they are usually maintained as separate, unrelated entities. However, implementing SMA channels as a functional extension to event channels contributed much to keeping changes to the existing driver implementations small:

- *Port* numbers (see section 4.4) not only refer to event channels, but the attached SMA channel as well. This avoids the overhead of implementing dedicated resource management for SMA channels in the VMM, as it does in both backend and frontend drivers.
- Operations on event channel and SMA channels are logically related. Specifically, frontend notification and a flush of the SMA command queue can be performed as part of the same operation. This avoids introducing additional hypercalls to backends.

Interplay between event channels and SMA channel will be revisited in more detail within the following two sections, which cover usage of SMA channels for event logging in common backend and frontend drivers.

5.6.2 Split I/O Rings

VLS-capable backend drivers operate in a dedicated mode of operation when tracing virtual machine execution. All input to frontend devices is routed through SMA channels, thereby through the hypervisor.

One way to facilitate synchronous updates to the control and status interface is to carry the concept of double buffering, as employed for the `shared_info` structure, over to the circular buffer shared with the frontend device. Figure 5.5 depicts the the resulting *split I/O ring*[6] architecture. The driver pair shares one I/O ring and one inter-domain event channel used to notify a respective remote end of queued messages. An SMA channel has been attached to control and log messages issued by the backend.

The ring split creates a second ring structure in driver domain memory, of size

[6]Not to be confused with the concept of split *drivers* introduced in section 4.5, which refers to the overall split into frontends and backends.

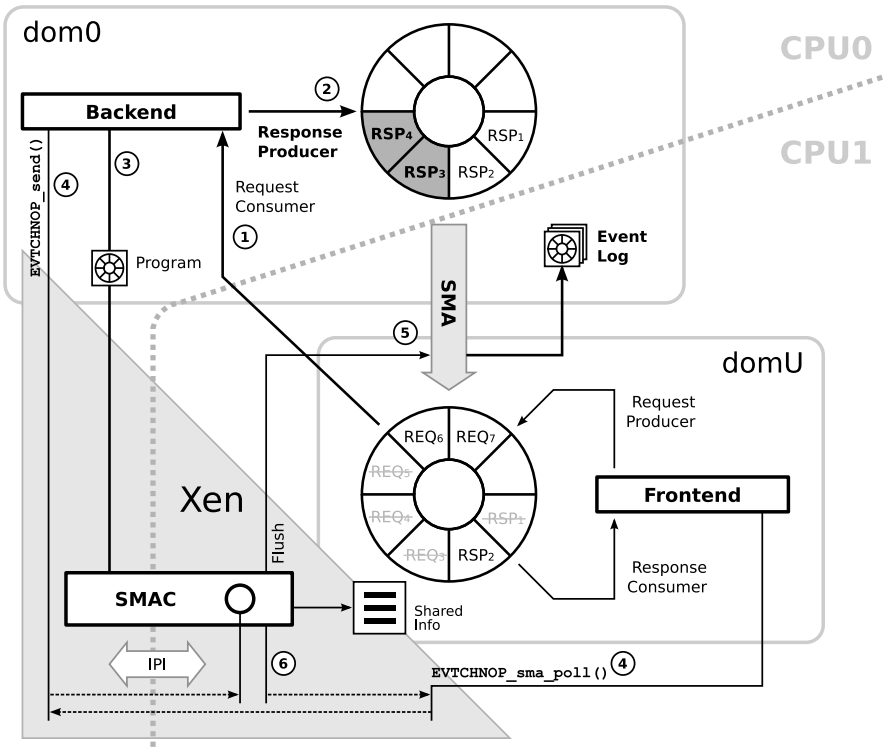


Figure 5.5: Split I/O rings, decoupling backends (`dom0`) from frontends (`domU`) entirely.

and layout identical to the primary structure, which remains shared with the frontend. While the shared I/O ring remains accessible to the backend driver, only reads are performed on it by the backend. It remains private to the backend and transparent to the respective frontend driver. Unless trace mode is enabled, the backend will revert to the original model of section 4.5.

Modifications to the ring are first stored to the back buffer. Source-level changes to the driver conditionally replace pointers to the shared memory with pointers to the back buffer. These changes have been fully integrated with the ring access framework introduced in section 5.3.2. Figure 5.5 exemplarily shows annotations for individual steps taken by a backend driver during request processing, thereby issuing two response messages ($\texttt{RSP}_3$ and $\texttt{RSP}_4$ to a shared ring structure. The backend driver therein proceeds as follows:

1. Request consumption operates equivalent to non-traced mode, in that messages queued by the frontend remain to be read from the primary.
2. Response generation issues $\texttt{RSP}_3$ and $\texttt{RSP}_4$, not to shared memory but to the same relative position in the back buffer.
3. To actually perform updates to the I/O ring, commands are issued to the SMA channel: Let B refer to the back buffer page frame, and S to the shared buffer, `offsetof`(f, p) denote the byte offset of field f in (page) p, and `sizeof`(f) denote the byte size of field f. For the example depicted, a program to issue both responses might comprise the following command sequence:
   a) `SMA_CMD_target(S)`
   b) `SMA_CMD_copy(B, offsetof(`$\texttt{RSP}_3$`, B), sizeof(`$\texttt{RSP}_3$`)+sizeof(`$\texttt{RSP}_4$`))`[7]
   c) `SMA_CMD_write(B, offsetof(rsp_prod, B), sizeof(rsp_prod), <5>)`
4. If the frontend is to be notified, the backend will activate the shared event channel. Forwarding of the notification will implicitly perform execution (*flushing*) of the channel program, i.e. all commands queued on the SMA channel are performed before the target domain is notified.

   If instead the frontend remains in progress, as determined from `rsp_event`, a deferred flush will be executed by the frontend, which performs a `poll()` operation on the SMA channel.
5. The SMA channel program is executed. Beyond committing memory accesses to the primary ring instance, this step interfaces with the VLS component for event log generation. This will be discussed in more detail in section 5.8.3.

[7] Or two `copy` commands, if the index of $\texttt{RSP}_4$ wraps around on the ring boundary.

6. If the SMA channel was flushed via a notification on the underlying event channel, the notification is forwarded to the frontend.

Updates to ring indices using the `write` command in step 3 are in support of asynchronous command execution by the SMA controller. Different from messages, indices may be frequently refined by backend drivers, possibly before pending commands have completed. Messages, by the producer/consumer protocol, are never overwritten before being committed to frontend-accessible memory for further processing.

5.6.3 Cooperating Frontends

Step 4 in the previous section, at which SMA program execution is executed by an explicit (thereby synchronous) operation needs additional explanation. As outlined in section 5.3.2, the ring protocol does not necessitate any message produced to be accompanied by a notification on the remote end. As long as the remote consumer remains in progress, gratuitous notifications can be avoided. This concept can be carried forward to SMA flushes triggered by cooperative frontends. Upon the last remote message consumed, drivers built upon ring access framework perform a "final check" for additional messages queued by a remote producer, which includes setting the `rsp_event` field to request notifications, unless additional messages are present.

When incorporating SMA transfers, the I/O ring protocol incorporates a two-way SMA interface, where flushes can be initiated both from the sender as well as the receiver side:

Asynchronous (push, backend-initiated) notifications trigger a flush of an attached SMA channel. Like any asynchronous event, memory accesses will be performed as part of interrupt delivery to the target guest OS. While an SMA channel and the entity programming it may execute asynchronously, all accesses are committed to memory before the notification is delivered to the remote end.

Synchronous (Pull, frontend-initiated) flushes are triggered via a dedicated event channel operation (`EVTCHNOP_sma_poll(port)`). Any pending SMA command queued on an attached SMA channel is executed upon return from the respective hypercall.

Both operations enforce the frontend's view on memory subject to SMA updates to become consistent with the backend's accesses. Ultimately, a cooperating frontend will poll the SMA channel at two different conditions during its execution:

- As a request producer, before deciding whether to notify to backend. The SMA flush produces a view on the consumer-written `req_event` mark to become consistent with backend state.
- As a response consumer, before performing the "final check" for incoming response messages, or requesting later notification by themselves. Again, this enforces memory consistency, this time with the `rsp_prod` index written by the backend.

5.6.4 Memory Sampling

The proposed SMA channel usage based split I/O rings as described in section 5.6.2, represents a workable solution, but can be significantly optimized. Split I/O rings, like back-buffering of the `shared_info` page shared with the hypervisor, represent the strongest level of separation between externally induced state changes to guest systems and their potential effect on control and data flow within them.

However, only a small fraction of the individual items written affects the guest system immediately. Split `shared_info` structures have been established as a construct which helps to keep the amount of change on respective parts of the VMM modifying shared information small. This however, is not the case with I/O ring accesses.

Instead, the process of transferring message bodies through back-buffers private to the backend device is not strictly necessary. A message written by a producer does not affect processing by a respective consumer before the corresponding producer index (`rsp_prod` within the I/O framework) has been adjusted accordingly[8]. Consequently, backends may equally well continue to store messages directly. Even with concurrently running frontends under a piecewise deterministic execution model, as long as the producer index is updated in a strictly controlled fashion, the PWD assumption. The same constraints persist on updates to the event marks, which determine whether or not notification hold-off is performed by the frontend.

Still, all changes to shared memory ultimately need to be carried through the SMA interface in order to accomplish a complete event log. For this purpose, the SMA `sample` command, as described in 5.6, was added to the SMAC interface. Memory sampling traces blocks of memory from a given channel *destination*, as opposed to a source buffer property of the respective channel owner. Porting

[8]This is the reason why, within the original ring access framework, much effort is spent on proper memory fencing on architectures which may perform reordering of store operations.

SMA channel usage for I/O ring accesses to the sampling technique has notable impact on the backend. The revised SMA command (3b) is the only SMA operation performing a `copy` from private memory. All other operations are implemented as `write`s, carrying ring indices as immediate values. Hence, no need for double buffering of the ring structure persists. Present revisions of the Xen/VLS prototype subject to this thesis include support for double-buffered I/O rings as well as memory sampling, mostly for experimental purposes. The performance evaluation (section 5.10) was derived from memory sampling.

Way more important than the comparatively small amounts of data accounting for ring messages are bulk data transfers on granted frames. Back-buffering of user data would largely degrade performance compared to regular operation of driver pairs. This is not only due to a larger volume: as pointed out in section 3.3.4, the paravirtual interfaces promote the use of hardware DMA and memory transfers to avoid copying data wherever possible.

Similar to messages, user data is not processed by the frontend before receiving the accompanying response message from the backend, hence such transfers performed asynchronously in hardware do not violate the PWD model. It is up to the backend driver to program SMA channels accordingly. Section 5.9 describes an exemplary implementation for the paravirtual block storage interface included with the XenLinux kernel.

Figure 5.6 shows the final SMA channel usage per memory region on the I/O ring, including data transfers to granted frames. Apart from the eliminated buffer split, all other elements of figure 5.5 remain as described.

5.7 Inter-Processor Synchronization

Memory writes and sampling via SMA channels is sufficient for device I/O. Inside the VMM, back-buffering works well to guest state accesses for part of the memory shared. However, whether double buffering and deferred committal is applicable, depends on the type of memory access performed.

Many updates are simple store operations, executable without further synchronization. Deferred updates then do not adversely affect the correctness of algorithms dictating control and data flow between Xen or driver domains on the one side, and the receiving domain on the other side. Unfortunately, this applies not to any piece information shared between guest systems and the VMM. A general indication is the presence of atomic read-modify-write (RMW) instructions [24] to synchronize

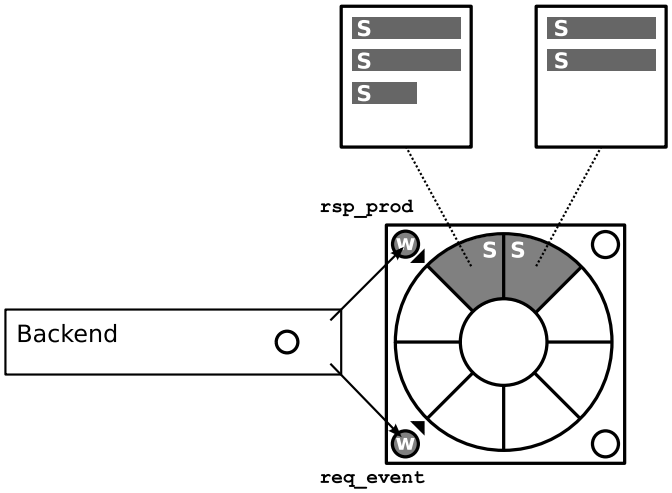


Figure 5.6: SMA access in shared I/O rings. (S) indicates memory sampling on message bodies and bulk data transferred, (w) 32-bit writes to ring indices.

Xen/VLS presently covers two cases where RMW operations require additional attention. One is the representation of event channel activations in shared memory, which rely on atomic bit operations (`set-bit`, `test-and-set-bit`), rendering deferred updates insufficient. The SMA layer features an alternative implementation of the original algorithms performed by the hypervisor during event channel activation. In summary, event channel activation resorts to deferred *processing*, to remain consistent with the original machine interface presented to guests.

The other exception are grant table processing performed by the VMM on behalf of remote domains. Different from event channel activations above, those require mutual exclusion of memory accesses of the target domain. That is, the traced domain has to be interrupted to be able to trace memory accesses in a fashion consistently replayable. Both types are discussed throughout this section.

5.7.1 Shared Info SMA

Section 5.1.1 outlined the fields allocated to the virtual event map. Figure 5.7 shows a graphical representation of the original algorithm. Without SMA, all fields depicted reside in shared memory. Between guests and the hypervisor,

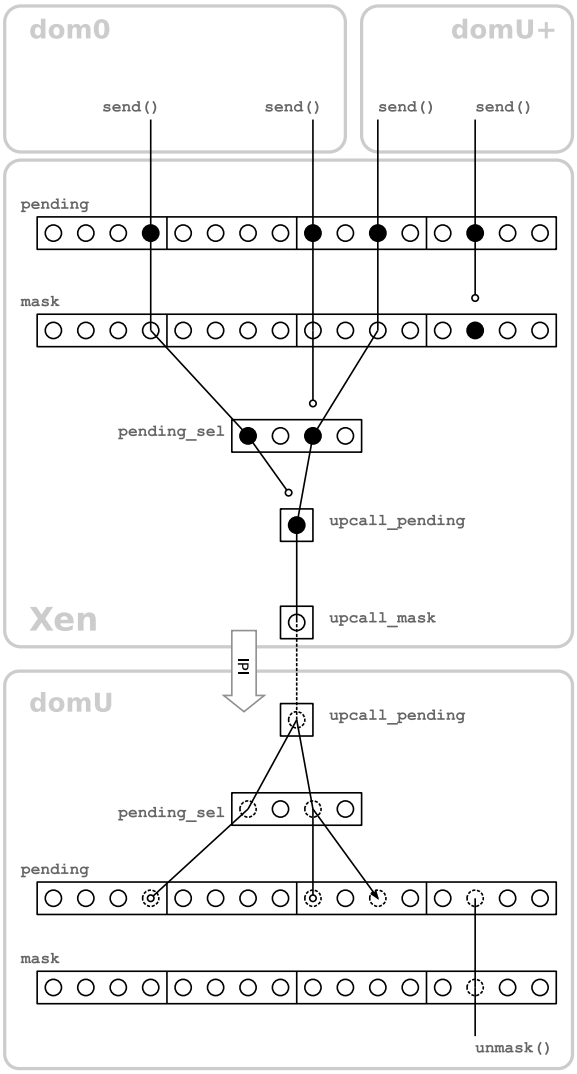


Figure 5.7: Event channel activations in Xen.

notification implies atomic bit operations on multiple fields in the `shared_info` page, to be read and written in a particular, strict order. The paths taken through Xen represent multiple threads executing a `EVTCHNOP_send()` hypercall. As shown, the state of the event map (`pending`), the event index (`pending_sel`) and notification flag (`upcall_pending`) thereby arbitrates between concurrent senders, which synchronize through RMW operations (`test-and-set-bit`). A bit already set when accessed will let the sender consider it's task already being finished elsewhere. While multiple channels may be activated within the same instant, only one thread will ultimately trigger a notification on the receiver side.

Event processing by the target system consults the same fields to identify pending events. It may interleave on a different physical processor, clearing the fields in opposite order during event processing.

Committing event channel activations to guest-visible state cannot be performed by overwriting respective fields within the primary structure. The operation to be performed is an atomic `set-bit` operation concurrent with the target. Instead, event state carried on the primary and back-buffer structure is merged into the primary. In summary, event channel activations are thereby split into two phases:

1. A preparation phase operating on the back buffer. This phase performs a variant of state migration, but cannot commit changes to target memory without further processing.

   This phase is performed by event senders, which can proceed without waiting for phase 2 to terminate.

2. A deferred processing phase performed by event senders. Different from deferred updates, additional processing is necessary in order to remain consistent with original machine interface semantics.

The present SMA implementation includes a custom variant of the original event activation algorithm. It is part of the SMA layer, but not performed by the SMAC. Instead, it rather represents an auxiliary component of a fully piecewise deterministic processing environment. As with regular double-buffered `shared_info` updates, the SMA layer has to decouple guest (`domU`) event processing from sender activity. The resulting change to the VMM can be outlined as follows:

- Event channel activation by senders operates on the `shared_info` back-buffer. The alternate implementation thereby maintains arbitration between concurrent senders as shown in figure 5.7.

- References to guest-written fields by the VMM are based on the primary structure. This includes the `mask` variables shown in figure 5.7.
- References to items read and written by both Xen and the guest system combine state from both the primary structure and back-buffer. For example, a test whether a given port is set would perform a logical `OR` on both versions of the `pending` field.

Note that the need for phase 2 only applies when performing an original domain execution conforming to the PWD model. As soon as all changes have been committed, changed state can again be sampled and replayed at the more coarse-grained level word-sized updates to shared memory. The custom event channel activation thereby only affects the guest during piecewise deterministic execution, not VLS as the logging facility, nor replay of resulting event streams.

Consequently, the SMA command set described so far can be readily applied to capture hypervisor accesses to the `shared_info` structure. Updates to the back-buffer page are marked and signaled as pending work items to a respective target VCPU. Event processing generates and issues commands to the SMAC instance allocated for the respective virtual processor. Note that no SMA channels are involved here, which only serve as device interface presented to driver domains. Updates to write-only data updated by the hypervisor, such as the virtual event system and wall-clock time representations, are copied and thereby traced according to section 5.8.

5.7.2 Grant Table Updates

The second exception to state migration are grant table accesses. Granted guest frames can only be released after a respective grantee dropped all of its mappings in virtual memory to them. Xen therefore indicates the state of in-use grant entries to their respective owner. This constitutes a potential race between granters and grantees. As an example, a granting domain may revoke a grant entry at the same point during system execution at which it is accessed by Xen on behalf of the grantee. Grant entry updates must therefore be performed atomically in memory.

Access to respective table column from either end is performed through a compare-and-swap (CAS) operation (`cmpxchg` on x86 hardware [45]). CAS is a read-modify-write instruction comparing the contents of the respective memory location with a supposedly up-to-date value and, if equal, replacing it with the new one [24]. If the value does not match the expected one, system software may recheck the datum and, if applicable, restart the operation. With grant

entries, an attempt to adjust frame state would either be reiterated by the VMM or ultimately signal a failure to the calling domain.

The update, like all memory store operations when executed asynchronously within the processing context of a remote domain, clearly induces a non-determinism in its effect in the system state of the frame owner. Different from the operations described so far, it cannot be managed by deferral: Since the result of the CAS instruction immediately determines success or failure to a grantee, it must be performed (i) by the initiator and (ii) on original target memory, to remain consistent with guest system state.

The general solution is to temporarily pause the target domain. The initiator will enter a critical section, blocking the domain from resuming execution until the operation is completed. In multiprocessor configurations, this is performed in a sequence depicted at the bottom of figure 5.8 (*Mutual Exclusion*):

1. The initiator enters the critical section, notifying – via IPIs – all processors running the target domain. It then spins (i.e. actively waits in a loop) until completion of step 2 on all notified CPUs.
2. Notified processors enter an ISRs, indicate termination of step 1 to the initiator, then proceed to step 4.
3. The initiator executes a shared memory update, marking one or a number of SMA trace operations in the target domain as pending. It then leaves the critical section, allowing completion of step 4 to the target domain.
4. The target spins until completion of step 3. Before returning to the guest system, subsequent sampling and logging of all memory updates is performed by the initiator.

The sequence induces wait phases on both initiator and target threads. First, the initiator needs to wait for the target domain to interrupt execution before performing any update operation. Second, the guest system cannot proceed until all updates have completed. Both make the technique relatively slow when compared to memory access deferral. As shown in figure 5.8, at least part of the SMA processing may interleave with critical section, as long as ultimately all updates performed are processed before resuming target system execution.

5.7.3 Summary

This section will complete the overview over the SMA layer. In summary, SMA comprises three different techniques, depending on individual needs of the type of update performed. To demonstrate the differences, the impact on control

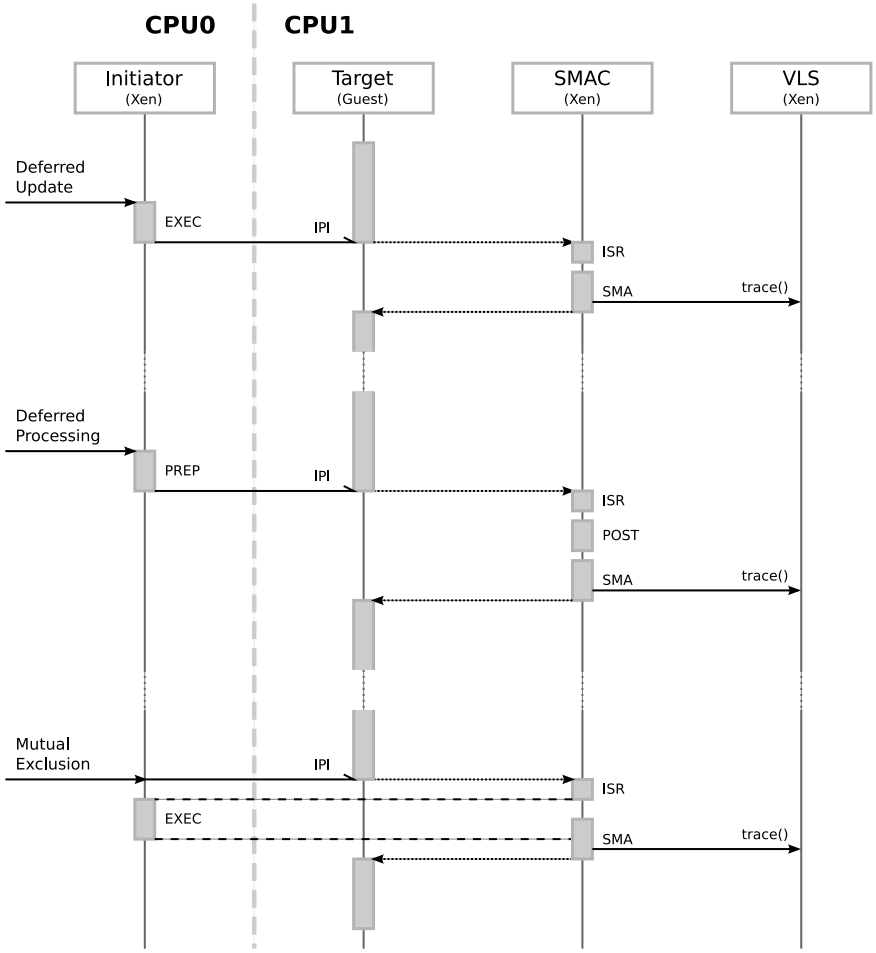


Figure 5.8: Summary of flow control during SMA updates.

flow within the initiator side of any update and the reception performed after migrating processing of state changes to the virtual processor of a target domain can be visualized by UML sequence diagrams, as shown in figure 5.8. Table 5.1 summarizes the resulting impact in overhead and complexity.

Deferred Updates are performed directly by the initiator unmodified (the activity labeled EXEC in figure 5.8). While sensitive to target control flow, such updates need not be performed in target memory but back-buffered in private memory. In shared-memory multiprocessors, updates are then signaled via IPIs.

EXEC does not need to be performed synchronous with execution of the

Type	Overhead	Compl.	Xen: Fields of use	Access Type
Deferred Updates	Small	Simple	Time, Device I/O	`store`
Deferred Processing	Small	Complex	Event Channels	`set-bit`
Mutual Exclusion	Notable	Simple	Grant Tables	`compare-and-swap`

Table 5.1: Comparison of memory access types in SMA mode and their usage in Xen.

target domain, thereby experiencing low performance impact. Except for additional statements to program the SMA controller on the receiver side, few changes to the original operation as performed on unconstrained receivers are required, yielding comparatively small impact on the existing VMM code base. Original algorithms are just provided with pointers to back-buffer space.

In Xen/VLS, deferred updates are by far the most common variant. The technique can be applied to timer updates in memory shared with the VMM as well as any remote memory access performed through SMA channels, thereby covering all paravirtual device I/O exercised by driver domains.

Precondition is that memory accesses are no RMW (*read-modify-write*) operations but only stores to memory, i.e. the initiator is not concerned with an original value subsequently overwritten. In practice, memory will typically have only one reader (the target system) and one writer (either the VMM or a remote domain).

Deferred/Split Processing may be performed on operations requiring read-modify-write access atomicity on fields shared with the target system (multiple writers) but have weaker constraints than those served with fully synchronous updates.

Different from deferred updates, operations requiring memory access atomicity racing with the guest system are migrated into target domain context. Figure 5.8 depicts the general case of "split" processing: part of the original operation may be performed by the initiator (labeled PREP) the part requiring access atomicity to target memory is executed by the target domain (POST).

Precondition is that any result of the operation on the initiator side can safely be determined prior to actual termination of the the memory access

on the target side. Hence, the initiator may finish PREP concurrently with the target system. Similarly, the target may proceed during initiator activity, again yielding relatively low overhead. Like deferred updates, state changes are back-buffered in private memory and their commitment to target memory migrated into the execution context of the receiving domain.

The downside compared to deferred updates is larger implementation complexity. Splitting execution into PREP and POST phases separated in time typically requires an alternate implementation of the original operation. It is therefore worthwhile for operations occurring performed at relatively high frequency. Rare operations not justifying the amount of change imposed to the VMM may turn to synchronous updates instead.

Mutual Exclusive Access is executed on target memory on the initiator side. For this purpose, the target is blocked from execution while the initiator performs all required updates.

No preconditions exist. Due to the mutual exclusion of initiator and target execution during critical sections, the approach would be general enough to replace any of the above techniques. However, synchronization between processors induces higher latency due to the additional wait phases when synchronizing execution across different processor cores.

Source-level changes to the VMM in order to access memory safely and replayable remain small. Additions comprise added function calls to enter and leave respective critical sections, and marking items written in target memory pending within, as outlined in section 5.7.2.

5.8 Event Log

As introduced at the beginning of this chapter, VLS is the event logging and replay facility built on top of the SMA layer.

To facilitate tracing of memory updates, the present SMA controller implementation provides a simple callback mechanism to which other parts of the VMM can register themselves. VLS thereby interacts with the SMA facility to essentially track all updates to guest memory, independent of the originator or type of data involved. Essentially, those of the SMA commands described in section 5.6 accessing target domain memory trigger an invocation of the tracing facility. Essentially, a callback to the VLS component carries the following pieces of information:

- A virtual machine frame number corresponding the the page frame in the target guest system's pseudo-physical address space.
- An *I/O vector*, comprising page offset and length of the update performed.
- A pointer to the data comprising the update, either in source buffer space for the `copy` and `write` operations, or target domain memory if sampling is used.

SMA transfers constitute one of two parts of what is considered an external event under the PWD model. The other part is the information necessary to replay the event accordingly. Section 5.3.1 described why I/O in Xen differs much from programmed I/O and the regular device model one would face with full virtualization. Due to the generally asynchronous I/O model established by the driver split and separation of platform I/O into domains of their own, one may suspect a comparatively large fraction of the resulting event sequence to retire in arbitrary processing context. Section 5.8.1 therefore examines the actual correlation between interrupt and potentially synchronous event context on a paravirtual machine interface in more detail.

Remaining events are truly asynchronous. In order to replay such events consistently, the original instruction at which they ocurred during capturing needs to generate a trap to the VMM, although remains innocuous to the virtualization layer. Both tracing and replaying such events require special treatment in software or hardware. Xen/VLS includes an instruction counter based upon x86 performance monitoring facilities. Due to the relative complexity of the PMU-based IC, description and evaluation if its design has been separated into chapter 6.

The remainder of this section is organized as follows: Section 5.8.3 discusses transmission of the resulting event log to a sufficiently privileged domain, typically `dom0`. Section 5.8.4 describes how congestion of the log transport is handled. Section 5.8.5 will summarize the elements defining the log format.

5.8.1 Context-Sensitive Logging

Not any non-synchronously induced change to guest system state, whether shared memory updates or control transfers by event channel activations, needs to be replayed asynchronously using an instruction counter. While all events subject to event tracing in VLS are ultimately externally induced, various system states may coincide with delivery. Different from event synchrony and asynchrony, virtual processor *context*, at which events retire in guest system state, can thereby be classified:

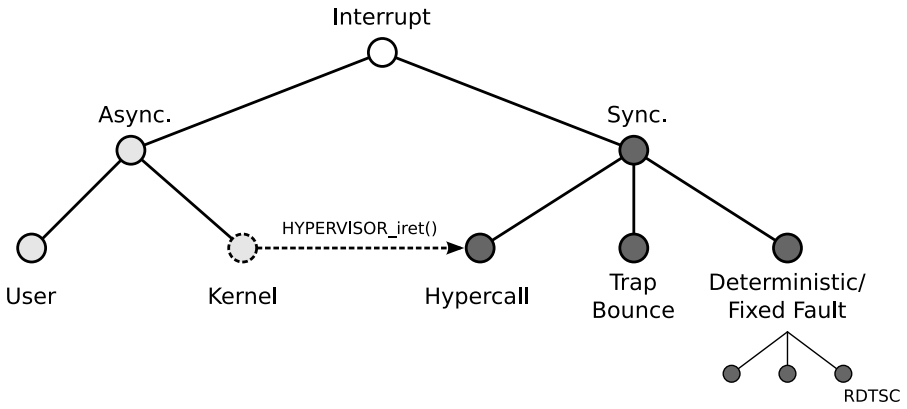


Figure 5.9: Events and synchronous control transfers.

Asynchronous External events which retire upon interrupts on arbitrary guest instruction boundaries. Those again can be partitioned into interrupts in user versus interrupts while operating in kernel space. The actual distribution depends on the amount of system interaction. I/O-bound applications have a higher probability being interrupted in guest kernel space than CPU-bound ones.

Synchronous External events which retire on synchronous entries into the VMM. If an otherwise non-deterministically occurring state change coincides with a synchronous one, the event becomes determinable by the synchronous one. The advantage of synchronous events is that they can be counted by themselves.

Discussion of asynchronous processing contexts in user versus kernel space will be delayed until the following section. The following discussion will study the practical relevance of coinciding synchronous context.

Synchronous context includes truly synchronous events (such as `rdtsc` emulation), but is not equivalent. The `rdtsc` instruction is a truly synchronous event, because the instruction itself induces non-determinism. However, like any other control transfer to the VMM, additional I/O events may coincide with privileged instruction emulation. The class of synchronous VMM entries comprises a number of different guest activities, which are depicted in figure 5.9:

Hypercalls mark voluntary entries into the hypervisor. Formally, they rather belong into a more general class of entries triggered by trap instructions (or *software interrupts* in x86 glossary). In principle, any instruction deter-

ministically transferring control to the hypervisor during execution belongs into this class.

System calls to the operating system *may* also belong into the same class of instructions[9].

Fixed Faults are processor exceptions which reliably reoccur under a piecewise deterministic mode of execution. Different from trap instructions, they mark *involuntary* entries into the VMM.

This class has been dubbed *fixed* faults because, different from forwarded (*bounced*) exceptions, they remain transparent to the guest. After the condition causing a fault has been cleared by the VMM, guest execution will resume without further notice.

One major subclass of fixed faults are privileged instructions emulated. The `rdtsc` instruction belongs into this class.

Trap Bounces Different from fixed faults, a large number of exceptions are not due to virtual machine map under control of the VMM, but forwarded to the guest system. Again, different from trap instructions, these are involuntary control transfers to a monitor and, subsequently, guest kernel.

Trap bounces will, under a piecewise deterministic execution model, reoccur reliably during repeated execution of the event log.

A large number of exceptions in Xen are bounced unconditionally. Examples are math coprocessor errors or a number of memory faults. Generally, trap bounces can be directly tracked by extending generation of the guest exception frame accordingly. The residual set of exceptions may be partly deterministic and qualifies as fixed faults. Table 5.2 lists the results for 32-bit and 64-bit x86 exceptions under Xen.

Trap bounces and fixed faults in combination represent a deterministic, proper subset of the sequence of all exceptions observable during guest execution. Contrasting trap bounces, identification of fixed faults can be comparatively time-consuming. Decision whether a fixed faulting condition occurs deterministically has to be performed on a case-by-case basis and careful analysis of the monitor. There is however obligation for correctness, but none for completeness. The

[9] On 64-bit x86 processors, the same instruction (`syscall`) may be used for both hypercalls and regular system calls issued from user space. Both transfer control to the VMM and therefore qualify as synchronous entries. System calls are trivially distinguishable by caller privilege levels and forwarded to the guest system. On 32-bit mode systems, in contrast, separate dedicated software interrupts are allocated to the VMM and the guest kernel. The system call gate allocated by Linux (vector `0x80`) would – unless altered – typically bypass the hypervisor.

prototype evaluated as part of this thesis presently identifies conditions for a number of general protection and page fault conditions (see table 5.2). Any event carried by an exception reoccurring deterministically but unnoticed will be (spuriously) identified as asynchronous and replayed accordingly.

The relative importance of context-sensitive replay becomes visible when studying their correlation with hypercalls. Asynchronous events coinciding with fixed faults and trap bounces may be considered truly incidental. The entire set of synchronous, piecewise deterministic entries into a VMM facilitate *opportunistic* techniques for synchronous event delivery.

Hypercalls, however, play a demonstrably more important role in practical event replay. The paravirtual VMI includes a number of calls which deal with event delivery from external I/O sources:

Scheduling in Xen offers a dedicated hypercall (`HYPERVISOR_sched_op()`) to yield a virtual processor cooperatively. They are equivalent in spirit to some POSIX functions for cooperative scheduling (`sched_yield`) or I/O multiplexing (`poll`) in operating system ABIs. As of Xen 3.0, a guest may perform a number of operations which synchronize with event delivery:

`block()` Block execution until event reception.

`poll()` Block execution until reception of one within a given set of events (ports).

Compared to native ISAs, blocking effectively signals virtual processor under-utilization to the VMM. In full virtualization, CPU sleep states would indicate the same condition. Especially under I/O-bound workloads, blocking is far from uncommon.

SMA Polling Section 5.6.3 described the role of the SMA `poll` operation on event channels as a method for enforcing memory consistency in face of deferred SMA processing. The state consistency gained in turn enables driver pairs to perform notification hold-off.

Hence, polling SMA channels on the receiver side entails additional advantage even beyond required memory consistency: It increases the number of asynchronously terminating I/O operations effectively retiring synchronous with guest system execution. This will be the case as long as event processing within a respective frontend device remains in flight.

Nr.	Mnem.	Name/Condition	Trap Bounce	Fixed Fault
0	#DE	Division by Zero	•	
1	#DB	Debug Exception	○	•[3]
2	#NM	Non-Maskable Interrupt	–[1]	–
3	#BP	Breakpoint Exception	•	
4	#OF	Overflow Exception	•	
5	#BR	Bound-Range Exception	•	
6	#UD	Invalid Opcode	○	•
7	#NM	Math Coprocessor Exception	○	×
8	#DF	Double Fault	•[2]	
9		Coprocessor Segment Overrun	•	
10	#TS	Invalid TSS	•	
11	#NP	Segment not Present	•	
12	#SS	Stack Exception	•	
13	#GP	General Protection Fault	○	○
14	#PF	Page Fault	○	○
16	#MF	x87 FP Exception	•	
17	#AC	Alignment Check Exception	•	
18	#MC	Machine Check Exception	–[2]	–
19	#XF	SIMD FP Exception	•	
		Software Interrupts	•	
		External Interrupts	○	
			• Always ○ Partly × Never – Not applicable	

Table 5.2: Guest exception determinism on Xen/x86 under the PWD assumption.

[a] NMI handling does occur on `dom0`, but not on non-privileged domains.

[b] Note that VM double-faults are always software-generated, and deterministic under a piecewise deterministic machine model. They have however no correlation to machine double-faults, which are always transparent to guests. Machine check exceptions are fatal.

[c] Debug exceptions which are not due to the guest system are exclusively due to the VLS instruction counter implementation.

5.8.2 Cooperative Synchrony

Synchronous VMM entries during guest system execution *implicitly* cooperate in the process of external event delivery. That means, it builds upon synchronous entries performed by guest systems unmodified. The question arises whether a larger degree of active (guest-driven) synchrony can (and should) be enforced *explicitly*, by altering guest systems accordingly.

There is a considerable number of practical high-availability platforms which pursue synchronous event delivery explicitly, by active cooperation, to a larger or lesser degree. One example is Delta-4 [14, 66], the first representative of semi-active replication. While the leader/follower model introduced efficient deterministic replay of asynchronous events, Delta-4 pursued the concept of predefined *preemption* points as part of the system architecture. Preemption points are executed recurrently as part of regular process execution. According to [14], they were installed within the application framework delivered with the hardware.

Hewlett-Packard's NSAA architecture [16] uses a technique called *Voluntary Rendezvous Opportunity* (VRO) to achieve synchronous interrupt delivery. VROs are embedded at various points in the operating system, at least transition points between different privilege levels. A hardware-aided rendezvous protocol is performed in order to agree on a common VRO embedded into the instruction stream. Once reached, the VROs schedule delivery of pending external interrupts.

As described in section 3.3.1, paravirtualization is based on the concept of virtualization-awareness and a general acceptance to adapt the OS to a surrogate interface different from the original machine. The system ABI, however, represents an important limit imposed by user demand. Applications and most system libraries remain unaltered or backward compatibility to the existing code base and thereby versatility is lost. The Xen hypervisor targets commodity operating systems and applications. Present examples include Linux, Windows or various UNIX-based operating systems. All these systems build upon a preemptive model for event processing and task switching. The VLS component, like Xen, ultimately has to conform to this model. Cooperation down to the application-level would therefore be out of place.

Kernel-level cooperation, in contrast, is particularly attractive. Privilege transitions as proposed by NSAA, either entries into guest kernels or the subsequent point of subsequent return, are especially simple to track from within a present system virtualization layer. As already pointed out in section 5.8.1, system calls typically enter the VMM before being forwarded to the guest kernel, partly

because the virtual privilege map has to be maintained by the hypervisor. Similarly, thread deprivileging upon return from the guest kernel may be intercepted by a VMM, unless performed in hardware. Xen's paravirtual VMI features a dedicated hypercall (`HYPERVISOR_iret()`) in place of the original `iret` instruction. The virtual `iret` can be used as a synchronous point of event delivery to substitute any event otherwise interrupting kernel space asynchronously. This is shown in figure 5.9: Upon event emergence, the privilege level of the current processing context is determined. If the present thread is running with kernel privileges, delivery may be delayed to the upcoming `iret`. No dedicated compile- or run-time instrumentation of the kernel needs to be performed.

Generally, cooperative synchronization represents a trade-off between two conflicting goals: VMM entries at high frequencies require careful instrumentation of the target system and, if induced in source code, may be considered a distraction. If the frequency is too low, interrupt latencies will adversely affect overall I/O throughput and system responsiveness. This may not be the case for delayed delivery in kernel space. Virtually all operating systems employ deferred processing of asynchronous I/O completion extensively. In Linux, *softirqs* (see section 5.5.2 or [18]) perform this task: Interrupt context only performs immediate acknowledgment of notifications, actual processing is deferred to a comparatively late point on the return path to a respective interrupted thread of execution. The difference between deferred or immediately delivered notification may therefore be small. Task prioritization performed in software may suffer, however.

Apart from diagnosing an apparently good match with customary virtualization infrastructure, measurable impact of kernel-level cooperation at regular intervals has not been extensively studied as part of this thesis.

5.8.3 Transport

To facilitate arbitrary postprocessing of the event log, transport to user space represents the most flexible solution. Possible uses include forwarding over a network interface, as would be the case with semi-active replication. Seeking deterministic replay at a later point in time one might rather save the stream to an attached storage system. Another potential use is performance monitoring and evaluation, as for the results presented in section 5.10.

Event log transport implies memory sharing directly with the hypervisor. This is different from most other components part of the Xen architecture, and therefore not covered by an existing VM management infrastructure present in the `dom0` kernel: Essentially, VLS log transport establishes an I/O interface directly

with the hypervisor. Memory management and interplay with the VMM cannot be safely performed by user applications alone: part of the process includes a translation back and forth between virtual addresses of the calling process and the underlying machine memory address space. Furthermore, buffer space exposed to the hypervisor needs to be locked from host memory management, such as demand paging. As a result, `dom0` support for VLS comprises two parts:

- A kernel space component managing machine memory dedicated to event log transport (`vlsfront` in figure 5.10). The module implements requisite control over the VLS layer and all data exchange with the VMM, including allocation of appropriate machine memory. The allocated buffer space is then mapped into the address space of the calling process for further processing, eliminating the need for repeated copying of the contained trace data.
- The architecture presented in figure 5.10 proposes a daemon program dubbed `vlsd`, as would be the case for semi-active replication, streaming the event log over an attached network interface. For the purpose of system evaluation as presented in section 5.10, a small command-line utility performed log analysis in a similar fashion.

Log transport builds upon the same generic framework for circular I/O buffers utilized by driver pairs[10]. For this purpose, one page frame is allocated as the shared I/O ring exposed to the VLS layer. Consistent with the design philosophy underlying split domain I/O, `dom0` is required to allocate all memory pages holding execution trace data from its own memory allocation. The message format is designed accordingly: each *request* message carries the MFN of a single page frame allocated by `dom0`. After being filled by VLS with log information, each page is returned to the calling domain with a single *response* message. Response messages do not necessarily follow the original order in which requests where issued. The message format of the present implementation leaves room for a maximum of 64 pages of memory queued simultaneously.

5.8.4 Flow Control

Apart from just forwarding the respective data to `dom0`, one major responsibility left to the VLS subsystem is flow control. Especially when the event log is ultimately to be forwarded over comparatively slow external links, such as storage or network interfaces (i.e. slow in comparison to the memory interface

[10]Without SMA extensions, obviously.

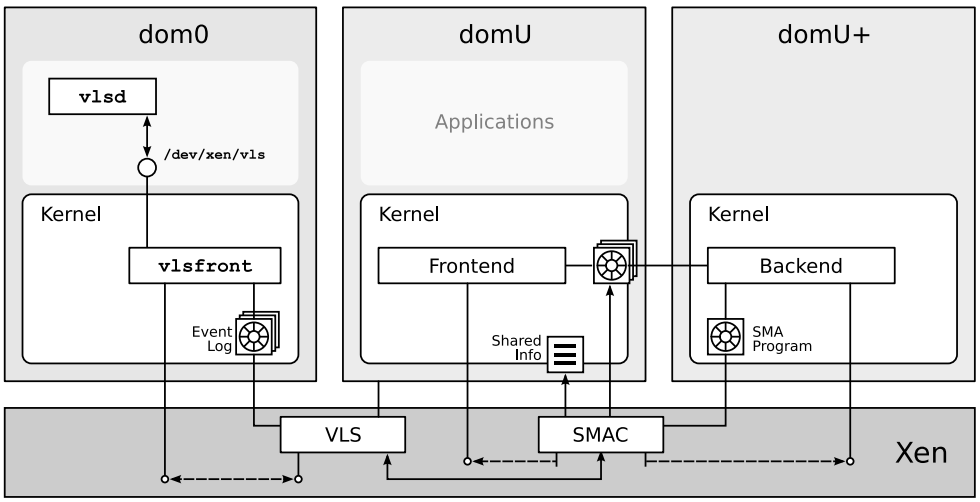


Figure 5.10: Log transport in Xen/VLS.

of the host systems), log transport is likely to suffer from congestion. Similarly, the log receiver may stay idle as long as no trace data is available.

Flow control in Xen/VLS could be built on top of a derivative of inter-domain event channels, which was originally introduced as part of the VMM support for full virtualization. Starting traced execution of a domain includes establishment of an event channel connecting the tracing domain on one end and Xen – on behalf of the target domain – on the other end. Both transmission ends signal message delivery via event notifications. Flow control then performs as follows:

- If event log generation cannot proceed due to lack of buffer space provided by the tracing domain, the replica domain is blocked from execution.
- Request submission by the tracing domain is accompanied with a notification, unblocking the traced domain if necessary.
- Similarly, log processing by VLS is controlled via event notifications received by the kernel space facility. Calls to the device interface may block the calling process interface until buffer space becomes available.

Flow control entails a bi-directional interface between the VLS component and the SMA subsystem. As long as a respective event cannot be logged, SMA cannot proceed. As soon as as the requisite event buffer space becomes available, a pending SMA update will be retried. Only unless events are pending, domain execution may return to the guest system.

5.8.5 Trace Data Format

The format of the execution trace emitted by the hypervisor closely follows the operation of the SMA subsystem as well as the transport facility discussed in the previous section.

Updates are grouped into *events*. Regarding their effect on target machine state, events resemble transactions: All changes to the traced guest system within a single event are performed atomically. A single event comprises all externally induced state changes at a single instruction boundary during the execution of the guest system[11]:

- A flush a flush of all pending SMA channels
- Updates to the target machine's `shared_info` frame.
- Any effect on machine state induced by the execution of non-deterministic instructions.

Given the relative size of bulk SMA transfers, a single event may thereby span multiple pages, which is incrementally forwarded to the logging facility. Like the transport protocol, the log format has therefore been organized into *frames*. Each frame comprises a response message in the event log and a memory page released to the logging facility.

In the present implementation, only two types of frames are differentiated:

Breakpoint control (`BRK`) frames include updates to the `shared_info` frame, plus control information required to replay the transaction.

I/O memory updates (`SMA`) frames include all remaining updates, exactly those induced by SMA channels allocated to driver domains.

The sequence of frames submitted to form one event invariantly adheres to the following regular structure.

1. Zero or more `SMA` frames.
2. Exactly one `BRK` frame. Presently, a single frame is sufficient to carry all `shared_info` updates and VCPU context information.

Consistent with the criterion of a sufficiently general interface postulated in section 5.4, the format of `SMA` frames is unspecific to the VLS component. The

[11] Note that future revisions of the SMA layer may relax the atomicity guarantee. As learned from section 5.6.4, only a subset of all SMA operations immediately affect control flow and therefore need to qualify as *events* to the VLS layer. Ultimately, only original access order needs to be maintained.

frame carries data as structured through the program generated by a respective SMA channel owner. I/O vectors are gathered into the response message. Section 5.9 will briefly discuss the implementation of SMA for the virtual block I/O backend.

The format of `BRK` frames is more strictly structured by the VMM than that of SMA frames. Like double-buffered memory transfers programmable on the SMA channel interface, frames are congruent to the target domain memory. Present versions of the Xen hypervisor leave space available at the end of the `shared_info` structure, which is reused in trace frames to store information about the computational state of the interrupted virtual processor. The resulting frame layout is thereby structured as follows:

- A `shared_info` structure, carrying updates the the shared information frame. Like `SMA` frames, individual regions updated are indicated by I/O vectors encoded in the response message.
- A data structure dubbed `vls_instant`, storing the VCPU context at which the event is to be replayed. *Instants* are organized hierarchically according to event context described in section 5.8.1:

  `TRAP,` i.e. a synchronous, under the piecewise deterministic execution assumption deterministically (re-)occurring processor state. This includes trap bounces, fixed faults, system- or hypercalls, or emulation of a non-deterministic instruction such as time stamp counter accesses.

  `INTR,` i.e. an asynchronous event interrupting an arbitrary guest instructions.

Both types of instants comprise all information necessary to replay the event precisely at the original instruction. Type `TRAP` is augmented with a software counter reading identifying the specific instant within the overall trap sequence executed under the PWD assumption. Type `INTR` is accompanied with an instruction counter value. Replay of asynchronous events and PMU-based instruction counters for x86 systems will be discussed in more detail within chapter 6.

5.9 Example: SMA-Channeled Block I/O

Present revisions Xen/VLS include full support for SMA in the paravirtual block (i.e. storage) I/O back- and frontends. The implementation contributed much to the present understanding on how the interface to synchronous shared memory access needs to be structured, and the cost of porting existing drivers

to revisions capable of replaying paravirtual guest systems. Both is provided here as an exemplary study on what is required from driver domains.

5.9.1 Implementation

Matching customary disk geometry, the virtual block I/O layer operates on individual sectors read or written from/to disk, where a sector always carries 512 bytes of user data. The paravirtual block I/O protocol carried out between driver pairs is structured as follows: A read request issued by the frontend may carry multiple (presently up to 11) *segments*, where each segment includes a reference to a target page frame, and start and end sector numbers to a range of up to 8 blocks (i.e. up to 4096 bytes). Indicated blocks are then to be buffered in the provided frame.

Frontend requests are be translated and forwarded according to the interface of the block I/O sublayer of the driver domain which carries the backend driver. Upon completion of the request, data has been readily written to the target frame[12]

In the SMA-capable backend driver, partial completion of a read request issued by a frontend generates one SMA channel program per segment, instructing the channel to sample the target frame, where the originally grant reference mapped by the driver domain serves as the channel destination. On Linux, target page fragments which need sampling may be identified not from the original frontend request, but the I/O request returned by the kernel block I/O layer for that purpose. The fact that the kernel merges aggregatable sector ranges into single I/O requests serves an additional convenience to keep the resulting SMA channel programs small.

There is a large degree of flexibility in tracking block I/O, the implications of which have not yet been fully explored. In theory, one could incrementally utilize up to 8 (i.e. the maximum number of sectors per page) I/O vectors, each upon read completion of any individual sector, which may still be saved to a single single event frame. The present implementation rather delays channel activation to the point where all individual sectors per segment (i.e. per page) have been gathered, which results in only a single I/O vector traced, which then covers the entire segment.

The trade-off here are larger updates, thereby larger latencies induced by

[12]If the storage is backed by a physical disk and storage controller, the process will typically employ DMA transfers in hardware. As pointed out in section 5.6.4, this is not in violation of the piecewise deterministic execution assumption.

channel program execution, versus larger, but incrementally executable channel programs. A full evaluation would require the need for a more advanced log generation infrastructure in the hypervisor than presently available. Future revisions of the Xen SMA layer may move to more fine-grained traces.

5.9.2 Maintenance Cost

Actual maintenance cost is hard to predict without foreseeing the future development roadmap of the original driver implementation. Focusing on block I/O, the design is proven, but alternatives to the presently Xen-proprietary protocol are in development. One example is paravirtual SCSI [41], promising better integration with enterprise hosting environments.

A more easily answerable question is how *intrusive* the present extensions in support of SMA-channel use are to the original code. A typical rule of thumb among system developers is "patch size", referring to the output presentation obtained by running the well-established UNIX *diff* utility (or a variant thereof) over the original and modified driver source. The following two sections pursue this approach for both the frontend and backend ends of the paravirtual block I/O pair introduced in the previous section.

Changes to the surrounding operating system kernel are not covered in detail here. Wherever possible, part of the needed functionality for both front- and backend support, e.g. details of SMA channel allocation, were not incorporated into individual drivers but rather into support libraries integrated with the host and guest system kernels. But different from drivers, core infrastructure is less critical because it may be maintained by a single party. Hence, driver maintenance requires more attention. Drivers (i.e. device classes) are generally various, even in the paravirtual case. Maintenance is up to different parties, and source code is subject to continuous change and improvement.

Backend VLS-capable driver builds are optional at compile time. The backend driver has been carefully crafted to avoid changes to the existing code, e.g. by resorting to C-language preprocessor macros nullifying SMA-specific statements if disabled. Unless activated at compile time, a driver identical in size and behavior to the original version will be built. If enabled, no functional changes to the original version are induced, i.e. as long as the frontend execution is not entering SMA mode, the driver will only be affected by a small number of conditional operations testing for its activation.

Including a retrospective code analysis without presenting sources, the results

	Changes		Xen Code base
Number of files	4		of 4 files
Host I/O (`blkback.c`)	7 chunks	+84 lines	-0 of 579 lines
Headers (`common.h`)	2 chunks	+13 lines	-0 of 139 lines
Initialization (`interface.c`)	3 chunks	+96 lines	-0 of 171 lines
XenBus Interface (`xenbus.c`)	6 chunks	+47 lines	-0 of 414 lines
Total	18 chunks	+250 lines	-0 of 1303 lines

Table 5.3: Change induced to a VLS-capable block I/O backend driver.

for the backend driver can be summarized as shown in table 5.3. A *chunk* is one consecutive range of lines subject to change, an estimate for the number of individual locations altered. Aggregate numbers of source lines (+ for additions, - for removals, e.g. due to replacements) demonstrate the amount change in relation to the original source.

The most important result is that in order to enable a full trace of guest I/O, no real interference with the the original code base was implied[13]. While all parts of the driver were subject to extensions, changes were mostly orthogonal to the existing code.

Host I/O includes both the driver pair protocol implementation as well as interfacing with the host storage interfaces. The comparatively large number changes was only due to a necessity to save grant references (as opposed to handles) in heap storage for later referral, in account for asynchronous channel operation on the receiver side. The rest was dedicated to issuing SMA operations the the hypervisor.

Like with so many pieces of software, more effort was spent on the surrounding infrastructure than core issues of SMA mode. Code incorporated into driver initialization includes memory and channel allocation and release. Backend drivers support entering and leaving SMA mode on the fly, i.e. independent of present connectivity of the backend interface with the frontend instance. For this purpose, the backend listens to a dedicated per-domain variable in the XenStore directory (`/local/domain/<domain_id>/vls/sma`) which is set to `1` (on) or `0` (off) by the tracing facility. If the value of this variable changes, a notification is received, which triggers channel allocation and usage, or a release, respectively.

[13] Even to the surprise of the author when the results shown were assembled.

	Changes		Xen Code base
Number of files	1		of 2 files
Headers (`block.h`)	1 chunks	+2 lines	-0 of 155 lines
Total	1 chunks	+2 lines	-0 of 964 lines

Table 5.4: Change induced to a VLS-capable block I/O frontend driver.

This mechanism accounts for extensions to the XenBus interface.

Frontend Table 5.4 contrasts the numbers derived from the backend port with those resulting from the frontend instance subject to repeated execution. The only mandatory change to driver operation at run-time is incorporation of polling event channels from the receiver (i.e. *response consumer*), in order to trigger channel flushes to shared memory from the receiver side at critical points during memory accesses potentially interleaving with the backend.

The necessary changes to response consumers on I/O rings could be integrated into the C-language ring access framework presently shared by all present paravirtual device classes. Changes thereby remained almost entirely backwards-compatible with the present code base. The only change necessary accounts for a mapping from SMA channels to the corresponding event channel number, for which definition of a single C preprocessor macro proved sufficient.

5.10 Evaluation

For the purpose of this thesis, there are a number of different properties to evaluate. One is performance impact. Since traced machine execution from unconstrained guest state access in multiprocessors, the resulting slowdown should be measurable. Another important question is overall log size and bandwidth consumption, i.e. log size per timer interval during execution, which varies with different workloads assigned. Tests performed include a number of synthetic and application-level benchmarks. Since a large fraction of execution traces is due to I/O operations, I/O-bound workloads are of major concern. Compute-intensive tasks should be lesser affected, but remain valuable in order to demonstrate some of the differences.

Measurements can be either relative to unconstrained execution on the same

virtualization layer, i.e. contrasting Xen/VLS with Xen, or relative to native performance. The Xen/VLS implementation on which experiments including a hypervisor were performed was based on Xen 3.0.2, running respective privileged and unprivileged builds of paravirtualized `dom0` and `domU` Linux kernels (version 2.6.16). The native Linux kernel run for comparisons carried version 2.6.20. Generally however, the relative performance impact imposed on paravirtual Xen guests due to virtualization has seen considerable research in the past and is known to be comparatively small [22].

5.10.1 System Configuration

All tests presented here were based on a two-socket SMP system carrying two Intel Xeon 5100 (Core2 Architecture) processors at 2.66 MHz, each with two processing cores. The machine had 5 GB of physical memory installed. Unless otherwise noted, the traced guest system received a physical memory reservation of 512 MB. Processors implement shared 4MB of level 2 cache.

Experiments in multiprocessor configurations target the common case of a two-CPU configuration, where one processor core is dedicated to the overall virtualization and monitoring facility, and the other the respective target system to trace. The virtualization layer therefore comprises two non-SMP domains with one respective virtual processor each. The `dom0` configuration carrying both virtual device backends as well as the monitoring facility in user space. Generally, `dom0` and `domU` ran on separate cores.

The Xen hypervisor allows *pinning* of virtual processors to physical processor cores. The given configuration therefore allows setups to experiment with different core assignments: By default, Xen would assign VCPUs to processor cores 0 and 4, i.e. to separate sockets communicating via the system bus. The remaining cores remain unutilized. This setup will further on be referred to as the SMP configuration.

Anticipating widespread adoption of multi- and manycore architectures, the alternative setup is to exploit CMP, i.e. pin `domU` to an immediate neighbor of `dom0`, which would be number 1 in above layout. The idea is that such a setup may gain additional benefit from physical colocation of the tracing and traced entity. Shared L2 caches potentiate a notable increase in log throughput. IPI latency should decrease, promising to lower the impact of thread synchronization, especially for mutually exclusive memory accesses such as during grant table updates. The downside are impending cache collisions of memory working sets which remain unshared.

Third, uniprocessor configurations (UP) let both the monitoring facility, device virtualization, and the traced guest share a common physical processor core. The increased demand for world switches between `dom0` and `domU`, in order to serve I/O requests from the paravirtual guest, has a measurable cost in guest system throughput and latency. The same detrimental effect may be expected from pinning trace consumption and replica execution to the same core.

5.10.2 Linux Kernel Build

Extensive compiler runs during build procedures for large software systems are usually accepted to represent a good combination of both disk I/O and CPU utilization. These tests therefore measure time and bandwidth taken to build the default kernel configuration of a Linux 2.6.23 source tree on a locally managed *ext3* file system with GCC 4.1. The experimental system shared one ATA disk among all guests. Virtual disk images assigned to the traced guest instance were loopback devices, i.e. backed by sparse files carried on `dom0`'s file system. The same configuration was used to all all later tests performed.

Analysis of trace log bandwidth progressions over the runtime can teach a lot about the inner workings of respective applications studied. Beyond a raw summary of trace packet sizes and frequency, investigation reveals information about software, the surrounding operating system and, last but not least the interworking between the two.

The chosen log format divided into SMA and BRK frames allows for a convenient arrangement to separate bandwidth dedicated to peripheral device I/O from control information solely affecting the virtual processor. As outlined in previous sections, SMA frames comprise device control interface data, user data as well as grant table accesses performed by peer driver domains. BRK frames include event channel activations, updates to the time abstractions as well as the information necessary for later replay events precisely as experienced during VCPU execution. Data presented across the remainder of this section will show CPU and I/O bandwidth as separate contingents.

The top graph in figure 5.11 shows a stacked bandwidth diagram for the Linux kernel build across the overall build period of about 5 minutes taken to complete the run of the make utility from a clean source tree. The numbers sampled report a total size of 60.8 MB, consisting of 54.1 MB SMA log and 6.7 MB in BRK frames. The default kernel configuration comprises 1091 object files built: 22 MB are C source and 8 MB of header files. More than half of the data volume loaded from disk thereby accounts for source code; the rest may be attributed to file system metadata and data sources external to the build tree.

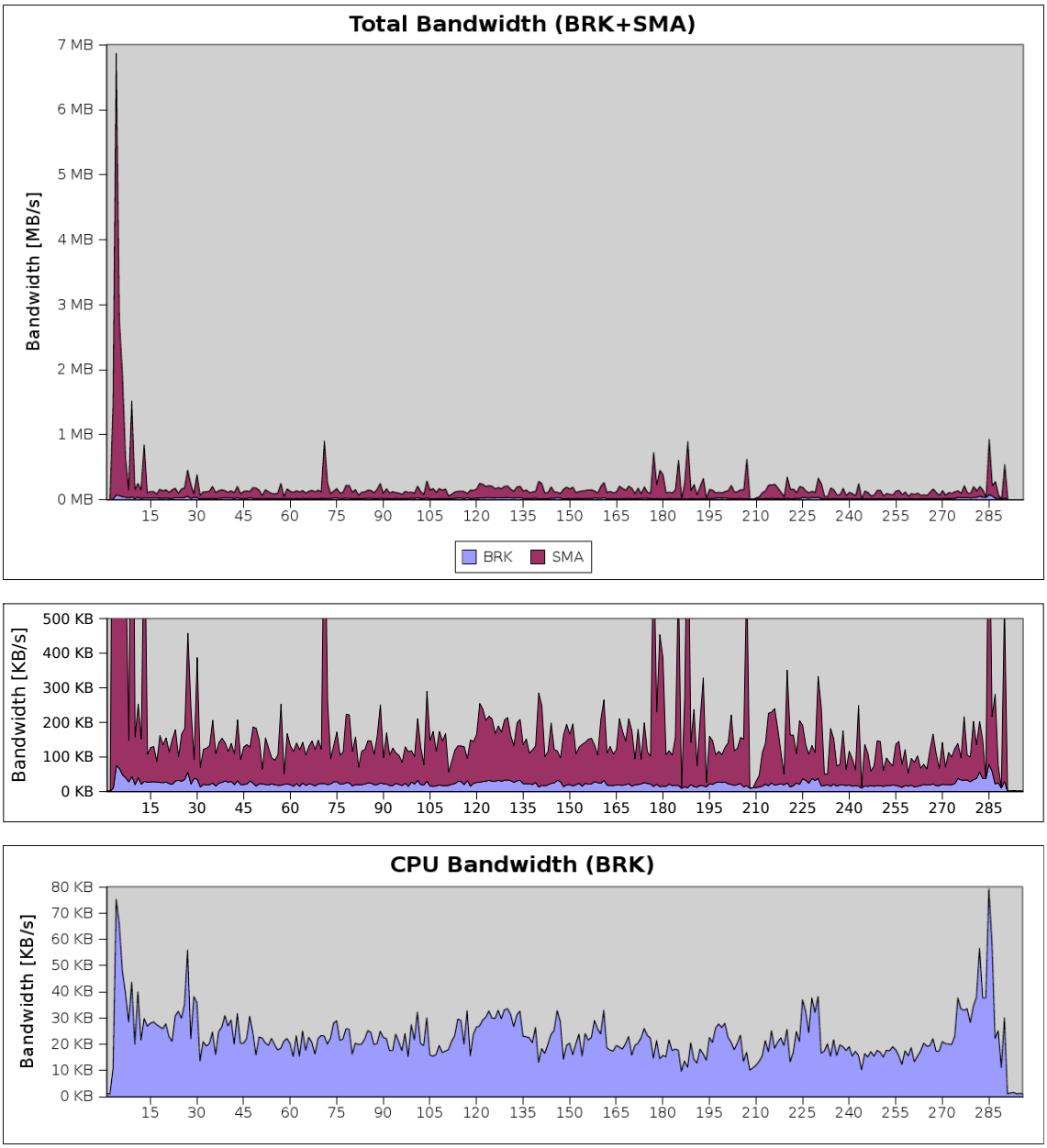


Figure 5.11: Trace bandwidth for a Linux 2.6 kernel build (256 MB RAM).

The figure demonstrates the importance of in-memory disk buffering for overall system performance. About one third (16.5 MB) of all data read from disk is loaded within the first 15 seconds of execution, presumably including a large fraction of the overall working set of header files involved. Linux, like any modern operating system, caches block storage contents aggressively to system memory. The traced guest system as of figure 5.11 received a physical memory reservation 512 MB. Since the processed input data volume is well below the memory allocation dedicated to the VM, files on disk are read at most once and served from buffer caches on subsequent compiler runs. The initial I/O bandwidth consumption of up to 6.8 MB/s is never achieved at a later point in time.

The middle of figure 5.11 shows the same graph limited to a lower range of up to 500 KB/s. The average compound bandwidth measured was 205.6 KB/s, of which 182.9 KB/s were SMA frames. Maximum CPU bandwidth was 79.5 KB/s, coinciding with a local I/O bandwidth of the execution trace, which is due to the larger number of interrupts received around that period. The bottom graph in figure 5.11 shows CPU bandwidth separately, which averaged at 22.7 KB/s.

In order to get a better estimate of trace impact, more I/O intensive applications are needed. Based on the same test, a simple way to achieve more disk usage is to significantly reduce guest memory. For this purpose, the same kernel build configuration was executed on a guest memory allocation of only 32MB, compensated with a sufficiently large swap partition. The vastly increased pressure on memory management leads to frequent flushing of kernel buffers and a fair amount of swap usage within individual compiler runs, deliberately reducing overall performance to only a fraction of available processor bandwidth.

Figure 5.12 shows the impact on total and CPU trace log bandwidth. Total execution time increased to 3844 seconds. The total log volume was 76.6 GB, an average of 19.7 MB/s. CPU bandwidth increases as the interrupt frequency rises, averaging 88.0 KB/s in BRK frame volume, at a total of 338 MB.

The left graph in figure 5.13 shows relative performance numbers derived from build execution time with 512 MB RAM, comparing native Linux with Xen traced and untraced execution times. The three groups of columns correspond to process time, i.e. *user* and *sys* as time spent in process and kernel space respectively, and real time elapsed, as output with the customary UNIX time utility.

As shown, the overall build process spends around 15% of its time in the kernel. Execution tracing in the SMP configuration results in a 12.2% slowdown

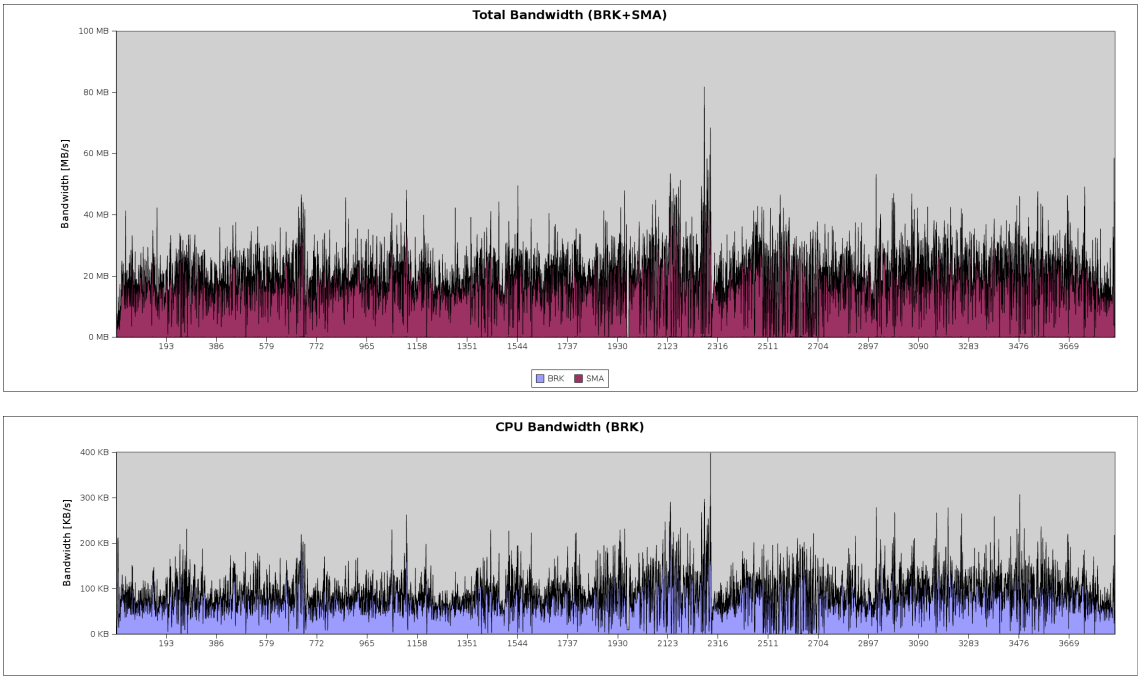


Figure 5.12: Trace bandwidth for a Linux 2.6 kernel build (32 MB RAM).

in raw kernel time, contrasted by user space performance affected only by 1.4%. The difference confirms intuitive expectations that tracing measurable would affect kernel space much more that application execution. A significant fraction of additional time spent in the system may be attributed to frequent polling of SMA channels. Slowdown measurable in user time is rather due to deferred interrupt delivery.

The overall slowdown due to execution tracing measurable is 4.1% in real time, compared to Xen without trace capturing. Comparing replayable with native application performance, the same benchmark with Linux on bare hardware reports an overall performance impact of 7.5% attributable to paravirtualization, or a total of 11.1%.

The right side of figure 5.13 depicts relative performance for the same experiment with VMs only assigned 32 MB RAM, as described above. The system spent about 90% of execution time swapping. Due to the higher I/O load, system time increased considerably, e.g. by 43.6% in the SMP configuration. However, system time only accounted for only 4% of total execution time. Compared to fully buffered I/O above, relative slowdown increased only slightly, to 5.7% with SMP.

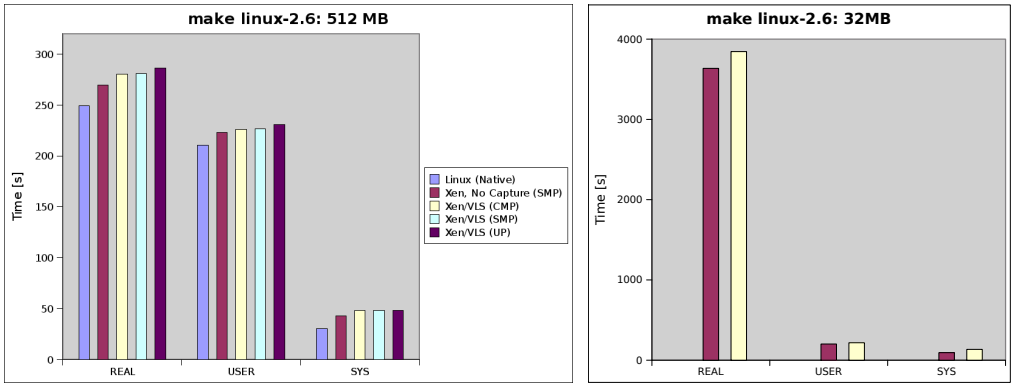


Figure 5.13: Relative Performance: Linux 2.6 kernel build.

5.10.3 SPEC CPU 2006

The large impact of device input in trace capturing can be contrasted with CPU-bound workloads, which are considerably more simple to trace. The SPEC CPU 2006 [78] suite is a "component-level" benchmark targeting solely processor and compiler performance. For the purpose of this thesis, only an integer test subset of the full benchmark suite has been evaluated. Since neither integer nor floating-point arithmetic induce instruction-level non-determinism, FPU applications do not promise any additional insight.

Figure 5.14 shows a sample bandwidth graph derived from an execution trace of the 401.bzip2 program executing under the `runspec` utility. 401.bzip2 is based on an implementation of the bzip2 compression algorithm and processes a reference workload consisting of six separate components in sequence (binary image data, program code and a combination thereof) [78]. Stressing processor and compiler performance, SPEC programs perform I/O only for initialization. SMA frames shown are due to loading of benchmark applications and their respective data inputs.

Tracing CPU-bound applications generates largely homogeneously composed event streams. The paravirtual Linux guest hosting the benchmark suite operates at an interval timer frequency of 100Hz. The experiment thereby demonstrates a special case where the log almost exclusively reflects interval timer ticks interrupting user space. Such periods typically comprise only BRK frames of two event types:

- `INTR` instances (100/s), carrying information about event channel activations and, since truly asynchronous events are encountered, the necessary

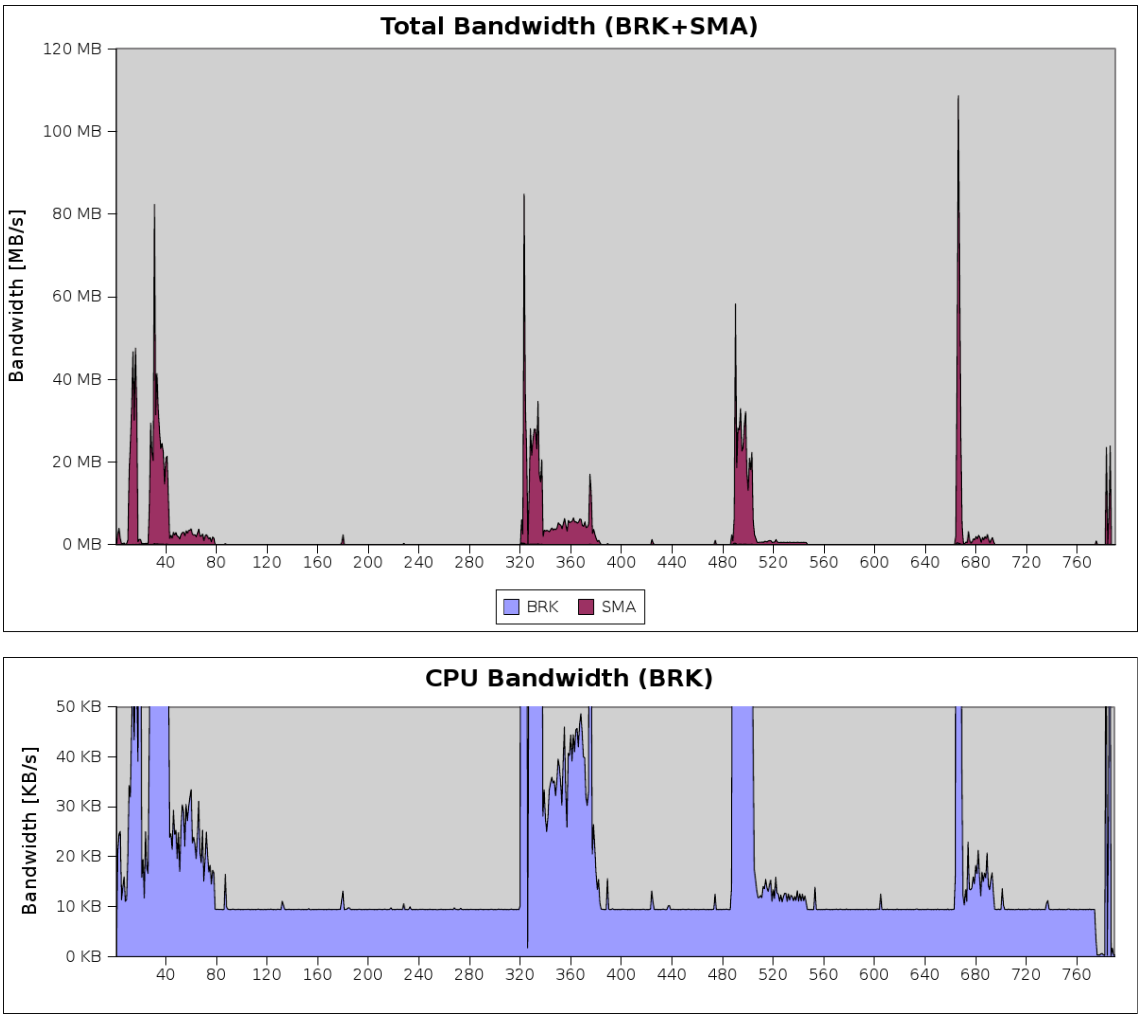


Figure 5.14: Sample CINT2006 execution trace for 401.bzip2

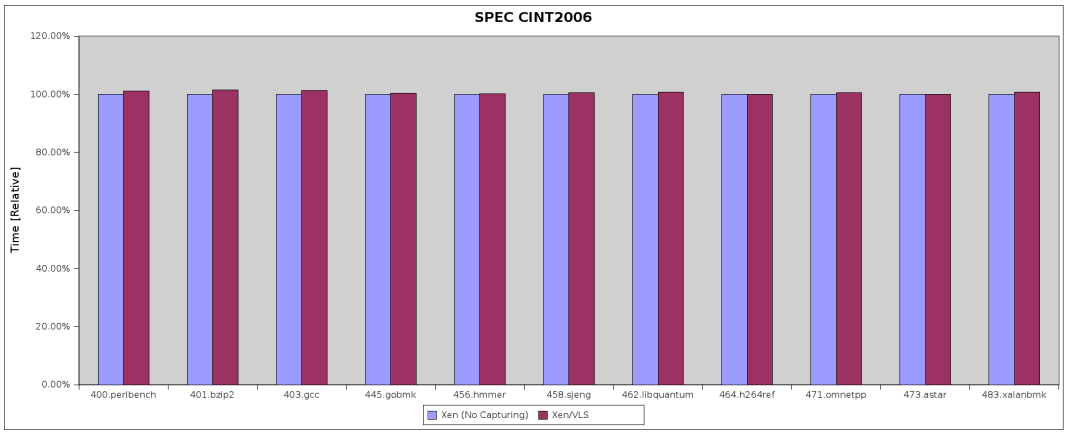


Figure 5.15: Relative Performance: SPEC CINT2006

.

instruction counter readings to enable consistent replay. These packets include regular updates to per-processor system time.

- For each `INTR` instance, a number of subsequent instruction emulations (i.e. type `TRAP`) determining `RDTSC` instructions (typically 4-5 per clock tick). Such time inquiries are typically due to processor time accounting by the guest scheduler.

Tracing periods of exclusive CPU utilization thereby generates a homogeneous event stream at a rate of 9.42kB/s. Performance impact is low, due to lesser event frequencies (averaging 518 Hz). Figure 5.15 shows normalized relative performance for the individual CINT2006 components run on the SMP configuration. In total, the performance impact due to trace capturing of 0.65% was measured.

5.10.4 Anticipated Bandwidth Limitation

The prototype subject to this thesis only provides fully deterministic block I/O to paravirtual guests. Network I/O, while potentially yielding a larger amount of interesting benchmarks, suffered from a defect limiting maximum throughput achievable. Compared to modern network interface over 1– and upcoming 10–Gigabit links, physical disks are comparatively limited in throughput. With the disk subsystem maintained by `dom0` in the experiments above, a maximum device read performance of ≈58.2 MB (Xen `dom0` was reported.

The constrained bandwidth at the physical layer, however, need not necessar-

ily apply to virtual device interfaces. To experiment with larger I/O bandwidth, virtual devices backed by machine memory instead of peripheral disks can be used. To this purpose, 3 GB of memory were allocated to a RAM-disk image (leaving 1 GB of memory to `dom0` and unprivileged guest systems), which backed an additional device exposed to the unprivileged domain.

The Linux `hdparm` utility [58] includes a simple benchmark program measuring raw device bandwidth by reading through the kernel buffer cache. It works by flushing in-memory caches, then performing sequential reads (each of size 2MB) on the device over a period of 3 seconds. Measurements were taken by calculating the arithmetic mean of 32 subsequent `hdparm` runs on the in-memory disk image.

Figure 5.5 shows the results, again comparing SMP, CMP and UP configurations with and without trace capturing. SMA mode apart, there is a notably disturbing effect when considering non-capturing operation alone: CMP peak throughput measured is reproducibly 13% less than on the SMP configuration. It remains unclear at this point why this is the case. L2 cache collisions may be the reason, but have not been encountered with the paravirtual network interface as shown in table 5.6, where CMP appears beneficial.

Of all configurations, tracing under SMP configurations reports the a relative performance loss of 47%. This is justified, since an overall peak loss of 50% or more *must* occur as bandwidth consumption on circular buffers approaches host system limits. Since tracing captures the same volume as transmitted from the SMA layer, the numbers should be rather understood as approaching an overall system limit of about 411.89 MB/s in the SMP case, or 439.88 MB/s for CMP setups. These numbers, however, should be assumed to be distorted by computational overhead within the block I/O layer.

Due to the functional similarity of paravirtual block and network I/O, SMA may be expected to affect intra-host communications over a replayable network interconnect. Table 5.6 shows TCP and UDP user data throughput measured for network RX and TX transmission paths (from `domU`'s perspective, i.e. RX corresponding to a `recv()` operation). The largest receive throughput measured with non-captured transmissions on the same system was 3446.64 Mbit/s on CMP configurations. While it is unlikely that original throughput can be maintained in SMA mode, it remains below practical limits experienced so far[14].

[14] Performance implications of circular buffers and memory sharing in Xen are obvious, but presently well understood. The problems is that the while the present protocol is sufficiently fast to saturate commodity gigabit network interfaces, CPU utilization exceeds those achieved with monolithic driver architectures such as native Linux installations by orders of magnitude [70]. Future I/O architectures are expected to partially remove this barrier [71].

	Xen	Xen/VLS	Loss
SMP	759.37 MB/s	411.89 MB/s	46%
CMP	661.99 MB/s	439.88 MB/s	34%
UP	490.76 MB/s	357.23 MB/s	27%

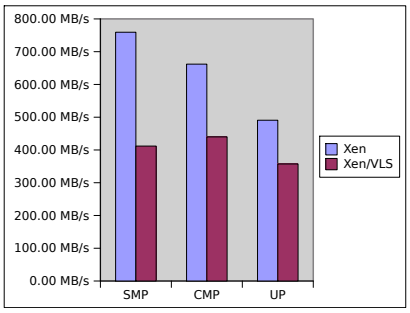


Table 5.5: Relative Performance: Circular buffer throughput on in-memory disk images.

	TX		RX	
	TCP_STREAM	UDP_STREAM	TCP_STREAM	UDP_STREAM
SMP	1711.92	2346.74	887.31	2344.54
CMP	2741.44	3811.54	1293.00	3446.64

Table 5.6: Uncaptured intra-host (`dom0`/`domU`) TCP bandwidth [Mbit/s].

6 Hardware Instruction Counting on x86 Machines

As described in section 2.2.5, ungoverned asynchronous event handling may produce arbitrary interleavings of state accesses between the preempted thread and the service routine. Chapter 5 recognized the need for an instruction counter within the augmented hypervisor, but did not discuss its realization.

The original idea of an instruction counter register implemented in processor hardware was originally described by Cargill and Locanthi [20], envisioning debugging and profiling purposes. The proposed programming model is simple: The feature comprises single, dedicated register per hardware thread, both readable and writable, which

- decrements by one on each instruction executed.
- triggers an interrupt, once the counter drops to zero.

In order to facilitate proper monitoring without counter interference by privileged system software, counting may be assumed to be scopable to user privilege levels.

Section 5.8 noted that the need for asynchronous event delivery can be significantly lowered with a combination of opportunistic synchronization and a modest degree of cooperation on the side of guest operating systems. Generally, however, preemptively driven event processing in guest systems and a typically non-cooperative process ABI makes consistent replay of arbitrary preemption points in commodity systems indispensable.

Unfortunately, few architectures ever featured a true instruction counter. Lacking a proper IC, an experimental counter implemented as part of the Xen/VLS project was based on x86 performance monitoring resources. A performance monitoring unit (PMU) built into a large number of available processors from different vendors. It has properties similar to a true instructions counter.

The remainder of this section is organized as follows. First, section 6.1 discusses additinal related work. Section 6.2 then introduces x86 performance monitoring features for a number of different processor models. While the fea-

ture set for the given purpose is similar across different vendors, there are some common, but non-obvious issues to be encountered when using these facilties when to implementing a suitable IC. These are either due to the microarchitecture or some properties of the instruction set, and discussed in 6.3.

Section 6.4 presents a suitable driver architecture integrating support for different processor types, as well as the individual solutions to above implementation issues. Section 6.5 will turn to an evaluation of the results, starting with a measurable performance impact when running monitored machine code at various interrupt frequencies. Section 6.6 describes improvements in counter precision archievable with different variants of event counting.

6.1 Related Work

A multitude of different techniques to determine and recover processing state at instruction granularity have been researched and developed in the past, in both hardware and software. Techniques implemented in hardware, such as instruction counter (IC) registers are generally the most convenient and ones, with insignificant effect on runtime performance. Solutions in software are feasible, but of lesser accessibility especially dealing with general-purpose processors featuring an instruction set which is complex by design, such as the x86 architecture.

The hypervisor-based system for active replication developed by Bressoud and Schneider was built upon HP's PA-RISC architecture Hewlett-Packard [19]. PA-RISC [39] comprised a *recovery register*, suitable for the purpose of replaying events deterministically. It operates functionally identical to the register interface described above.

There is a number of recent systems which utilize performance monitoring on Intel-architecture CPU families. The HP NonStop Advanced Architecture (NSAA) utilizes the PMU of Intel Itanium processors to consistently replay interrupts across different processor cores. Section 5.8.2 noted that NSAA exploits cooperation by applications. However, it supports *non-cooperative* processes as well, despite the kernel and run time support libraries rather targeting applications developed specifically to the given platform [16].

The ReVirt project was the first to combine deterministic replay with system paravirtualization. The implementation is based on FAUmachine (former UMLinux, see section 2.2.9). It includes drivers for AMD Athlon (K7), Intel NetBurst (Pentium4) and a partial implementation in support of Intel P6 and derived processors, such as the Intel Pentium M. Similar to the drivers

presented here, the instruction counter utilized debugging facilities such as execution breakpoints and single-stepping to work around some of the PMU event counts and favor branch counts over retired instructions where appropriate. In summary, the implementation provides insights comparable to the material presented here, but does not support a number of later processor models.

Lacking solutions in hardware, early work was done in the area of alternative control methods operating in software. Mellor-Crummey and LeBlanc showed that implementation of an efficient *software instruction counter* (SIC) by code instrumentation can be built [59]. There are alternative measures for program execution state, even at the granularity of single instructions. The definition of the original instruction counter as defined above is more strict than actually necessary for the purpose given. In fact, the *exact* number of instructions executed is of no particular purpose during event replay. It only represents the most straightforward enumeration of individual execution steps taken during a program state sequence.

The software instruction counter developed by Mellor-Crummey and LeBlanc exploits this fact by counting branches instead. Consider sampling the program counter (`PC`) during an arbitrary state sequence. It increments by the size of each instruction executed. The sequence of `PC` readings certainly cannot qualify as an IC, since the same code position may be revisited multiple times. However, the number of times this happens is bounded by the number of control transfers performed by the machine. Control transfers include jumps and branches. At a minimum, *backward* control transfers must be counted, since only those allow for individual code positions to be executed multiple times [59]. Subroutine entries must be counted for the same reasons. We idea of branch counting taken branches instead of individual instructions will picked up in section 6.2.

Given that the definition of what consistutes a precise instruction counter may vary, the discussion will be simplified by defining a number of simple properties any facility suitable to instruction have.

Let E be the set of all possible executions $e_0 e_1 \dots e_{n-1}, n \in \mathbb{N}$ of a system.

Uniqueness An instruction counter *count(e)* is *unique* if

$$(\forall e_0 e_1 \dots e_{n-1} \in E)(\forall i < n)(\forall j < n)(\text{count}(e_i) = \text{count}(e_j) \;\Rightarrow\; i = j)$$

Informally, for any execution sequence, $\text{count}(e_i)$ produces a different counter reading for any event e_i within that sequence.

Monotonicity An instruction counter is monotonically increasing $^{(\text{decreasing})}$ if

$$(\forall e_0 e_i \dots e_{n-1} \in E)(\forall i)(\forall j)(\text{count}(e_i) <^{(>)} \text{count}(e_j) \;\Rightarrow\; i < j)$$

Monotonicity implies uniqueness, but is a stronger property.

Determinism For any two independent executions a and b of the same instruction sequence $i_i e_1 \dots i_{n-1} \in I$

$$(\forall i)(\text{count}(a_i) = \text{count}(b_i))$$

Informally, if this property states all counter readings produced across multiple executions of one and the same instruction sequence are equal.

6.2 Performance Monitoring on x86 Processors

Many modern x86-family processors support a number of event counter registers. Designed to monitor machine events and aid code optimization, x86 *performance monitoring* extensions [76, 46, 5] provide a flexible means for counting both architectural and microarchitectural events during regular process execution. The major application is system and application profiling, e.g. as implemented by the Intel VTune performance analysis tool [76].

Unless auxiliary features are enabled, e.g. for event-based state inspection, hardware event counting is generally non-invasive, i.e. it does not affect regular program execution. Different from code instrumentation or statistical sampling employed by many classical profiling applications, e.g. the UNIX *prof* or *gprof* utilities [36], low runtime performance interference is implied.

Hardware performance monitoring facilities were originally introduced by Intel with the P6 microarchitecture, the foundation of the Pentium and PentiumPro processors [37]. They have been subject to redesign and enhancement with almost any later microarchitecture [46]. Generally, the interface comprises a number of model-specific registers (MSRs) for control and measurement. Performance monitoring implies *event* counting. For profiling purposes, events are typically microarchitectural, as opposed to architectural ones. With the exception of the Intel NetBurst microarchitecture (discussed below), the programming interface event comprises one selector register and one counter register.

Implementations commonly provide at least two independent counters. Figure 6.1 shows a custom architectural model of common x86 processor performance monitoring resources. Performance monitoring features at least two pairs of MSRs. Each pair comprises one *selector* register encoding events to be counted, and one *counter* register (PMC), accumulating sampled event occurrence.

A selector setting comprises an event identifier to count, plus additional control fields. Until 2006, performance monitoring extensions on x86 processors

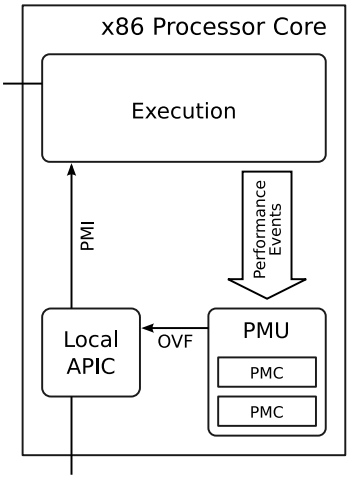


Figure 6.1: High-level microarchitectural model of x86 performance monitoring.

were non-architected, i.e. the implementation not only differed between vendors, but was subject to refinement with different revisions of the microarchitecture. More recent processor models released by Intel indicate that this is partly changing, with architected MSR and identifier assignments for a portable subset of countable events stabilizing in present future microarchitectures.

Consequently, each variant requires a different driver. The implementation subject to this thesis comprises the following model-specific drivers, unified under a common software interface:

PPro (Intel PentiumPro, Pentium II/III, Pentium M) The original P6 processor architecture dates back to 1995 [37]. From 2003 on, Intel revived the P6 architecture with the *Pentium M* product line of microprocessors.

P4 (Intel NetBurst) *P4* is a colloquial term for a large range of processors based on the P68 (*NetBurst*) microarchitecture [40]. Performance monitoring in the NetBurst microarchitecture [46] largely differed from the P6, regarding both features and the programming interface.

K7 (AMD K7/K8) The interface of the performance monitoring interface on AMD Athlon and Opteron processors is similar to the one introduced with the Intel P6, although with up to four PMCs [5] and different MSR assignments.

Core2 (Intel Core 2 Architecture) Intel *Architected Performance Monitoring* was ultimately derived from the the P6 interface, but represents a shift towards a fixed programming interface maintained with future revisions of the architecture. This process started with the release of the Intel *Core* microarchitecture.

There are two major event types which qualify instruction counting. An obvious choice is counting instruction *retirements*, i.e. instruction effect being finally committed to architected state[1]. The second option, as discussed above, is counting (taken) branch instructions retired, as a subset of the general instruction count. With a minimum of two counters available on any platform considered, both variants can be pursued simultaneously. In fact, section 3 will demonstrate that this approach can be beneficial on some systems.

Independent of the event class counted, monitoring execution is only practical if counter progression can be narrowed to the operating mode of the software monitored. Otherwise, execution of the runtime environment implementing the monitor will interfere with tracking the target instruction stream. Since the same requirements apply to application profiling, all performance monitoring implementations may be tuned to count user mode (privilege level 1-3) instructions exclusively.

Performance counters, in contrast to instruction counters, do not decrement but increment on each selected event encountered. A performance monitoring interrupt (PMI), if enabled, is generated on counter overflow, i.e. wrap to zero. Event replay may therefore set the counter to a negative value when initiating a state interval. As depicted in figure 6.1, counter overflow is signaled back to the same processor core through the local interrupt controller (LAPIC)[2]. The default setup typically generates a non-maskable interrupt (NMI) at the processor. For the purpose given, reprogramming the APIC to an unused interrupt vector is desirable.

Depending on the target application, available counter ranges need consideration. Typical counter register readings are 48 or 40 bits wide. However, on 32-bit architectures like the P6 family, writes are limited to the lower 32 bits, but sign-extended to the full counter width, i.e. initialization with negative values remains feasible. A portable driver may limit itself to a maximum interval length of 2^{31} instructions. Alternatively, larger intervals may be achieved by concatenation.

Many applications target event simulation at dimensions matching average interrupt frequencies found on commodity operating system platforms. This is

[1] With pipelined, out-of-order and speculative execution in modern superscalar microprocessor designs [37], the intuitive concept of instruction *execution* certainly does match its counterpart at the microarchitecural level.

[2] Microarchitectural details of the hardware implementation are solely vendor affairs. Hence, the concept of a separate PMU fully external from core execution units may not be accurate. However, the depicted reference scheme proved to sufficiently match the counter behavior experienced.

due to the fact that basic sources of asynchrony even at the application level, namely I/O and process scheduling, are ultimately driven by external interrupts. The maximum interval duration on such systems is therefore bounded by the timer interrupt. Typical timer frequencies are 100 Hz and above. On a hypothetical processor running at 1 GHz clock speed with one instruction retiring per cycle state sequences terminate after at most 10^7 ($< 2^{24}$) instructions.

6.3 Implementation Details

Regarding function properties as described above, x86 PMCs qualify well as instruction counters. However, this is not entirely the case in terms of counter reliability and precision. Section 6.3.1 addresses issues which are due to the fact that PMI generation is not driven synchronously by instruction execution. Section 6.3.2 discusses where performance monitoring counters are vulnerable to an observable form of non-determinism themselves. Section 6.3.3 discusses specific properties of the x86 instruction set which demand for additional attention.

6.3.1 Interrupt Latency

A fundamental difference between a PMC-based driver and a true instruction counter is the fact that PMIs do not generate traps, but interrupts. This can easily be derived from counter readings. Upon PMI reception, both instruction and branch counts will not equal zero, but a positive number, indicating execution beyond the state originally targeted.

The reason can easily be derived from the conceptual model shown in figure 6.1. Interrupt delivery through a local APIC is independent from thread execution. Architecturally, PMIs are delivered like external interrupts. In contrast, true ICs, like the one integrated into PA-RISC [39], would be architectural elements integrated with the control logic, generating *traps* at instruction granularity.

In the following the number of instructions executed beyond the programmed interval length will be referred to as *lag*. Practice reveals that the exact amount of lag is neither constant nor precisely predictable, even across repeated execution of the same state sequence. However, it is fair to assume that PMI lag must be bounded by some upper limit. Table 6.1 lists results for a number of different processors tested[3]. The values were derived from experiments and apply to the

[3]The ID column shows Family/Model/Stepping triples identifying the CPU revision.

	ID	max_lag
Intel P4	15/2/4	173
Intel P6	9/6/5	84
AMD K7	6/4/2	98
Intel Core 2	6/15/4	161

Table 6.1: PMI lag latencies for some x86 processors.

processor models as shown, but may easily be subject to small individual refinement for different processors. Drivers implementations add a small percentage for additional safety.

On the follower side, execution beyond the state dictated by the leader is not recoverable and must be reliably avoided. This works by adjusting the interval length in accordance with a given maximum lag. Construction of the interval length is shown in figure 6.2. Let i_l be the number of instructions given by the determinant of an asynchronous event. Then $i_f = i_l - \text{max\_lag}$ is a save number of instructions to adjust on the PMC register in order to follow. Upon PMI reception, the actual lag l experienced will in the range $0 \leq l \leq \text{max\_lag}$, hence $i_f + l \leq i_l$. As shown figure 6.2, this approach provokes a gap between an intermediate preemption point and the targeted point of event delivery.

Since the target IP is part of the determinant, a considerably simple method to recover from this state, and the one commonly employed on a processor models, is to terminate the remaining instruction sequence to execute with an instruction breakpoints. This approach causes slight additional overhead due to the exceptions generated. While the number of exceptions necessary depends on the control flow surrounding the target IP, the worst case of a tight loop involving two instructions will not exceed max_lag/2 exceptions before termination.

6.3.2 Interrupt Interference

Generally, instruction and – depending on the processor – branch retirement counts increment upon arbitrary control transfers. This includes exceptions and external interrupts, despite the fact that these events are typically transparent to application software, and intuitively would not be considered as separate instructions even when the event mask includes events at the operating system level.

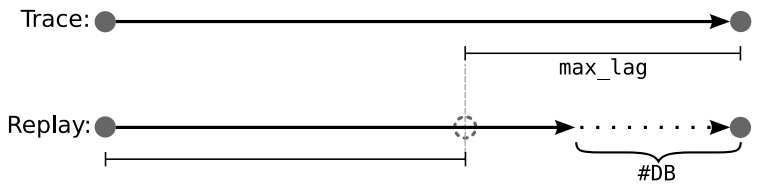


Figure 6.2: Construction of an instruction interval in follower mode

External interrupts are due to externally induced conditions and thereby non-deterministic. As outlined in section 5.8, the same applies to fair number of exceptions. To ensure consistent reading across repeated execution, non-deterministic increments must be compensated. Since the effect cannot be tuned in hardware, continuous readjustment in software is necessary. There are two methods to determine the number interrupts which occurred within a given time frame:

1. Utilizing the PMU to separately count interrupts. Most processors implement a separate performance event for control transfers due to interrupts and exceptions.
2. Counting control transfers from within the ISRs. This requires careful instrumentation for *any* possible vector.

Some drivers, such as P6, pursue method 2. The computational overhead to individual ISRs is negligible. Implementation complexity depends on the design of the surrounding system software. Most systems (including Xen and Linux) share common execution paths for exception and interrupt handlers, which then can be modified for that purpose.

There is some risk in counting external interrupts in software. The Intel architecture defines a dedicated *system management mode* (SMM)[5, 46], entered from reception of a dedicated interrupt (SMI)[4]. SMM mode is typically configured by firmware extensions, thereby difficult to alter without compromising the platform. Depending on the individual system configuration, SMIs occur on a regular basis. The impact of control transfer to SMM appears to be generally undocumented. Practice shows that control transfers to SMM mode do not always affect event counting. A more detailed investigation on a per-processor

[4]SMM has been designed to remain fully transparent to system software, much in contrast to modern ACPI-based systems. Typical actions performed by SMM include power management or emulation of platform facilities speed.

	Instructions			Branches		
	IRQ	Trap	Fault	IRQ	Trap	Fault
Intel P6	1			1		
Intel Pentium 4	0			1		
Intel Core2	1			1		
AMD K7/K8	1			1		

Table 6.2: Counter increment upon hardware interrupts and exceptions. With the exception of the Pentium 4, branch and instruction counters are equally affected.

has not been performed as part of the research presented here.

All recent revisions of both AMD and Intel processor manuals [4, 46] document that retirement counts are affected by interrupts, but do not quantify them. Table 6.2 shows the actual effect of interrupts experienced. Generally, counter increments vary with both different vendors and individual processor models. As shown, the interference may differ between instruction and branch retirements.

6.3.3 String Instructions

The x86 CISC instruction format comprises a variable number of prefix bytes [6]. Prefixes, if present, override some of the default properties of an instruction, e.g. address size, operand size or atomicity of complex read-modify-write instructions. An additional prefix variant are *repeat* prefixes (`REP`/`REPx`/`REPNx`), which turn certain load, store or I/O instructions into loop constructs. A `REP`-prefixed instruction will be executed in an implicitly loop, until some loop-terminating condition is reached. `REP` prefix bytes have a large number of different applications, including large I/O data transfers or yielding compact expressions for string processing procedures.

In contrast to "ordinary" retirement, `REP`-prefixed instructions generate not a single transition at the architected interface, but a whole state sequence traversed in multiple execution steps. This presents a major issue due to their specific effect on retirement event counts when combined with external interrupts.

- The state sequence neither counts as multiple instructions retired, nor

does it qualify as a number of branches performed (*atomicity*). This is uniformly the case with all processor models evaluated.

- However, repetition is non-atomic with regard to exceptions and external interrupts. All `REP`-induced loops solely operate on architected processor state individually committable, i.e. intermediate states are resumable after return from an ISR.

Since the single counter reading for all iterations fails to identify a precise point within the state sequence traversed, asynchronous events interrupting `REP`-prefixed instructions break the monotonicity property discussed in section 6.1.

Again, there exist different potential workarounds. All repeat-prefixed instructions decrement the `rCX` register during iteration [6]. One option is to reconstitute monotonicity by augmenting the IC with the value of `rCX` for any given state interval, then let followers resort to single-stepping individual loop iterations during replay. The downside is that the number of iterations is potentially large, which may lead to unacceptable numbers of individual iterations to be trapped. This implies opportunities for malicious workloads exploiting sensitivity to very long `REP`-sequences.

Another alternative, and the strategy performed by the VLS driver, is an additional rule to the event log, which disallows partially completed repeat-prefixed instructions to terminate an epoch on the leader side. The idea has been dubbed *REP-atomicity*. Leader mode therefore operates as follows:

1. Upon encountering an asynchronous event to be delivered, decode the prefix bytes of the instruction presently terminating the epoch at hand.
2. If a `REP` prefix is encountered, install an execution breakpoint at the instruction immediately following the present one, then resume execution. From the breakpoint trap, continue at 3.
3. Log and deliver the event.

Since the original IC semantics remain unaffected from this change, so does the follower side during event replay. Furthermore, an immediately following `REP` does not require additional treatment. The computational state preceding the first iteration is a valid target to the follower side facility, as described in section 6.3.2.

While `REP`-atomicity avoids excess single-stepping of loop iterations, it should be noted that it adds additional latency to external interrupt delivery. Partial decoding of any potential epoch boundary preceding an event is an $O(1)$ operation and does not need consideration. However, considering malicious workloads

as described above, atomic execution of very long `REP`-iterations may provoke notable impact on overall I/O performance.

6.4 Driver Architecture

Foregoing incorporation into the Xen hypervisor, a prototype IC driver was developed as a kernel extension on the Linux operating system. The prototype comprises a patch to a recent version 2.6 kernel release and a set of runtime-loadable modules. Sampled instruction streams are user space processes forked from a small utility mediating between user and kernel space. Necessary modifications to original kernel code include save and restore of PMC state upon task switches. The system implements two modes of monitored task execution to simulate leaders and followers: *Trace* mode generates a log of process interruptions at arbitrary regular intervals, *replay* mode reproduces the logged sequence on a respawned instance of the same program.

An experimental implementation on a modular commodity operating system proved highly recommendable, especially when considering integration into even lower-level monitoring facilities such as a Type I VMM. One reason are practical consideration during development: The module loading facility [18] implemented by the Linux kernel allows for replacement of custom subsystems at runtime. More importantly, operating system ABIs allow for arbitrary experimental programs (e.g. such involving string instructions) to be written as small user programs in a simple process execution environment. Performance evaluation could thereby be based on customary benchmark programs with various instruction mixes, isolating the potential contribution of the operating system kernel from elapsed benchmark times.

The resulting overall driver architecture is shown in figure 6.3. The driver could be carried from Linux to the Xen VMM without significant modifications. However, as discussed throughout previous sections, integration with the surrounding privileged software stack proved to be significantly more complex than a pure hardware counter would be. This is mainly due to the comparatively complex interaction with the trap and interrupt handling facilities. In order to facilitate compensation of interrupt accumulation, any control transfer to ring 0 is sampled in software. Similarly, execution breakpoints need to be managed separately from those on behalf of regular system tools.

At the lowest level are drivers for individual CPU models according to section 6.2. The IC incorporates a programming interface comprising two different modes of operation, corresponding to the two modes of a replay engine as would

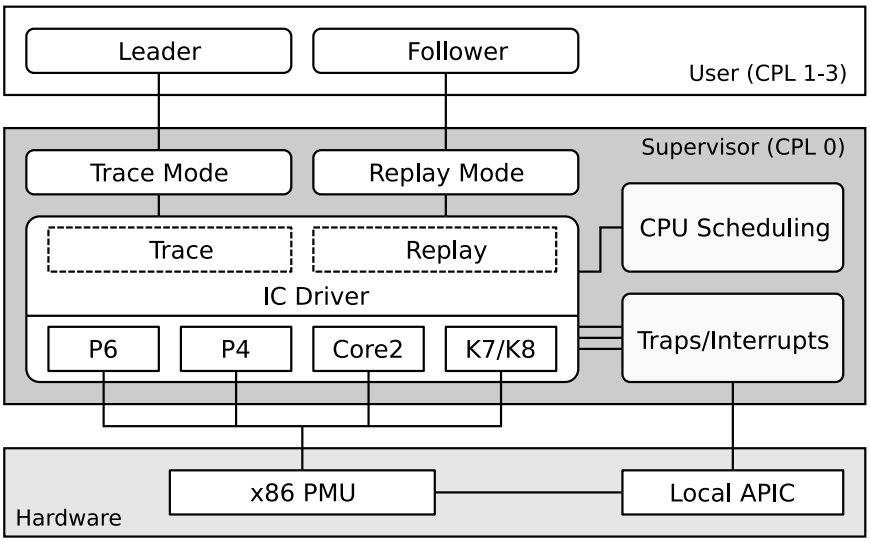


Figure 6.3: PMU-based instruction counter architecture for x86 systems.

be implemented as part of any overall monitoring facility: *Trace* mode treats an original program execution as a number of instruction sequences on a respective leader instance. Counter readings can be saved and reset at arbitrary points during execution. *Replay* mode controls follower execution from saved counter readings. A simple test module would interrupt any program after a selectable number of instructions.

6.5 Performance Impact

Solely focusing on the subject of asynchronous event replay, no attempt to trace and replay process I/O was made. Replayability of the event log therefore depends on user space code carefully crafted for full determinism. Fortunately, this is the case for a considerably large fraction of CPU benchmark programs available.

The customary mechanism for the *delivery* of asynchronous events on UNIX-like systems would be signals [42]. It should be noted that, for the purpose given, signal delivery was not performed. The purpose of the IC is to *potentiate* delivery of such signals. Since replay on a PMC-based IC on x86 is non-trivial, the following evaluation investigates the overall performance impact induced by the IC driver itself, in isolation from any in-process control transfers. To this

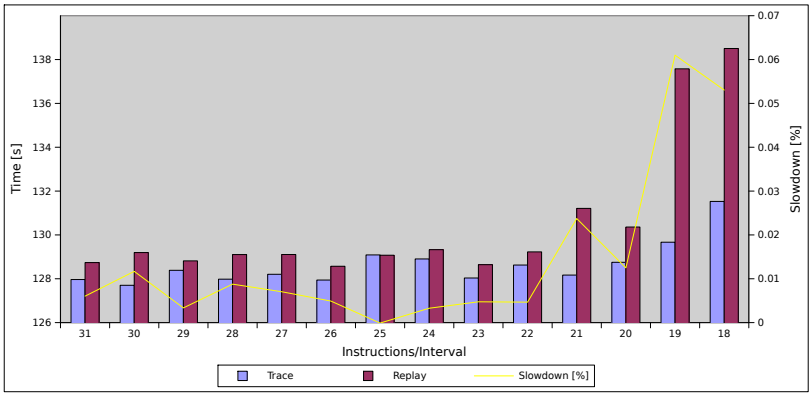


Figure 6.4: Performance downgrade by interval length

purpose, execution of fixed instruction sequences at various frequencies were investigated.

The performance evaluation testbed consisted of a selected subset of the SPEC CPU2000 suite of benchmark programs: `gzip`, `vpr`, `mesa` and `eon`. Each program was first run in leader mode, then replayed in follower mode. This process is iterated with interval lengths from 2^{31} to 2^{18} instructions decreasing by powers of two.

In order to avoid interference with system I/O, logging was performed to in-memory buffers. Execution runtime of the selected programs was intentionally shortened due to limitations in buffer space. Figure 6.4 shows CPU time for a test run of approximately 128 seconds, accumulating executions of all 4 programs by interval length. For each pair of bars, the left one shows time spent in leader mode and the right one shows follower mode.

Replay mode is considerable more costly than follower mode. This is due to the overall number of debug exceptions generated when compensating for PMI lag as described in section 6.3.1. Expressing exceptions in numbers would not be practical: The number of iterations trapped by execution breakpoints depends much on the length of iterated basic blocks, thereby individual code interrupted. In practice, followers would therefore dictate overall execution speed.

6.6 Counter Quality

Performance monitoring is not subject to the same overall requirements on precision and transparency to which core functional units must conform. After

eliminating the major, but fortunately observable sources of non-deterministic counter behavior, as outlined above, smaller inconsistencies, whose causes appear non-observable at the architectural level, may persist. However, one can observe that branch and instruction counts expose different precision, which is the major reason why the driver developed ultimately remains in support for both variants, even simultaneously, if necessary.

In order to analyze precision for different processor models, errors experienced when running programs in either leader and follower mode were analyzed. After each instruction interval performed on the follower, errors typically lead to measurable offsets between the counter value derived the leader execution and the counter reading after replay.

On Core 2 architecture machines, retired branch counts reveals no remaining imprecision, but counting individual instructions does. Figure 6.5 shows a bubble diagram which depicts an example distribution to demonstrate the difference. It shows counter precision for a run of 174.gzip at a relatively short interval length of 2^{18} instructions per interval, on an Intel Xeon 5100 2.66GHz processor (Core2 in table 6.1).

Similar patterns were obtained on the Pentium4 and K7 processors listed in table 6.1. Generally, success varies with with the given processor model. With the Pentium4 processor, stability of the branch counters was rarely sustained over periods of more than a few minutes. A more detailed analysis of reasons for failure than the experiences provided here cannot be obtained obtained from software-based analysis. As noted in section 6.3.2, one potentially interfering event source may be system management interrupts. These are however carefully shielded from system software by machine firmware.

Finally, inconsistencies can be made to compensate for each other, which is where counting both branch and instruction retirement is beneficial. Figure 6.6 shows counter precision for the same CPU2000 program and interval configuration as above, but on an Intel Pentium M 1700 MHz (P6 in table 6.1). As shown, offsets of up to an increment or decrement by 2 were observed on both the branch and instruction counts.

This case can be exploited in order to gain confidence in the counters, by applying the following rationale: There is no reasonable control flow construct which retires the same number of instructions and branches at the same time, then restoring the IP to the same original position. Any condition where the branch offset equals the instruction offset may therefore be considered a state match. A possible attack against this approach would be either a branch or jump instruction spinning in an endless loop or, similarly, a ring sequence of

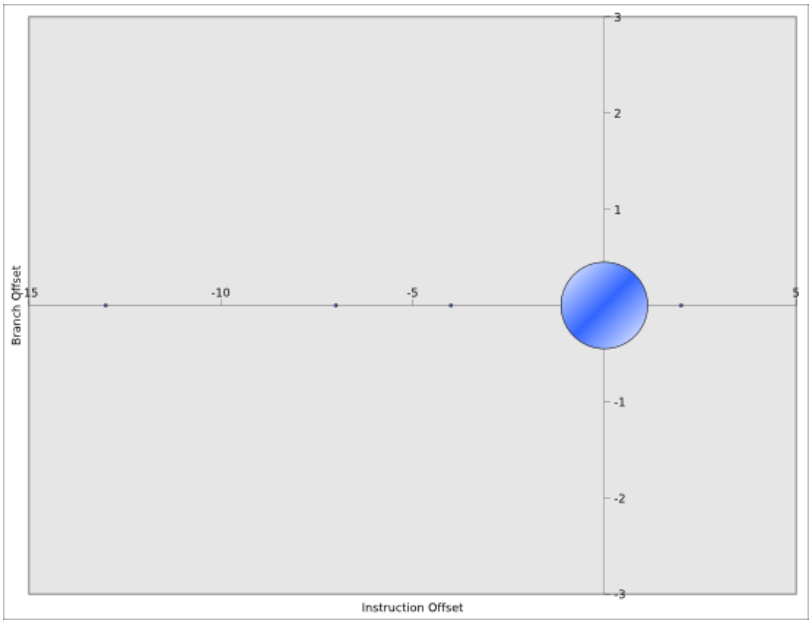


Figure 6.5: Branch and instruction counter precision on a Core2 processor.

an arbitrary number of branches and jumps. No such constructs, however, can produce any program state changes. In that case, event delivery would remain unaffected, even if the driver misinterprets the number of actual loop iterations performed. The driver can safely exploit this effect by deciding on a state match by a simple condition: An original state is reached during replay, if and only if an apparent branch offset equals the negative instruction offset.

6.7 Future Work

As an alternative to mechanisms implemented in hardware, binary recoding techniques might be applied in order to let code executed by the guest system maintain a software branch counter. The code generated would be similar in structure to the one created by compile-time instrumentation, as suggested by Mellor-Crummey and LeBlanc [59].

One of the original goal of the PMC-based instruction counter has been been to mimic overall performance properties of a true hardware instruction counter as closely as possible. This excluded binary recoding, in favor of executing original code unmodified.

Obviously, PMI lag and the resulting frequent need for breakpoint instrumen-

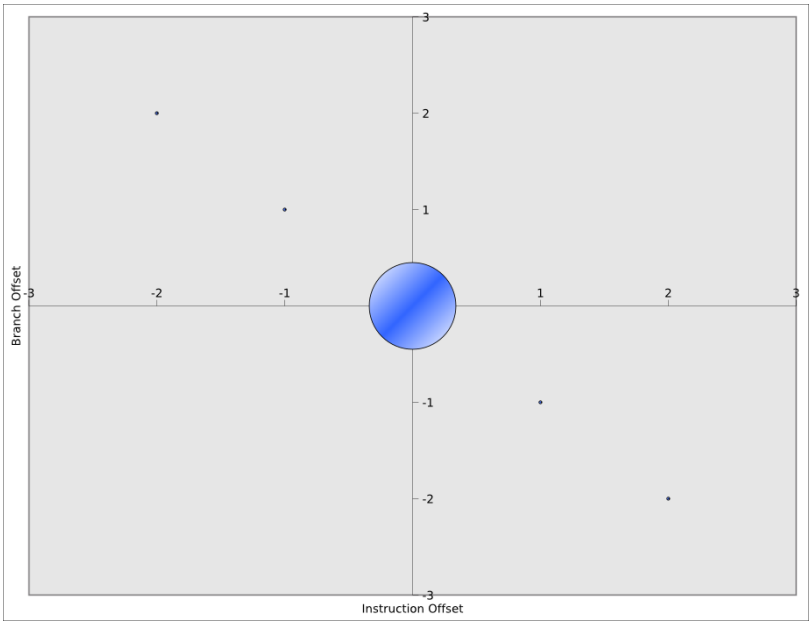


Figure 6.6: Branch and instruction counter precision on a Pentium M processor.

tation ultimately counteracted that goal. Furthermore, allocation of standard debugging facilities for the purpose of transparently replicated execution may potentially conflict with applications. One may argue that binary recoding could potentially eliminate overhead implied by repeating control transfers to external monitoring software. For this purpose, the original instruction marking the point of event delivery might be replaced by an instruction sequence performing both an interpretation of the original sequence as well as the comparison between PMC readings and the original point of delivery. Both can be performed from within user mode. Note however, that generating such an instruction sequence is not entirely trivial: executing instrumented code at the same privilege level will generate additional retirement events. Generally, any routine instrumenting leader or follower instance would need to account for its own instructions.

Additionally, binary recoding might eliminate the need to enforce atomicity of `REP`-prefixed instructions. As discussed in section 6.3.3, the approach taken is sufficient for evaluation purposes but potentiates large interrupt latencies, e.g. induced by malicious application code. Simulation of `REP`-prefixes, i.e. replacement with an explicit loop construct around the original instruction, can reconstitute interruptible execution. Navigation on the state sequence may match a logged `rCX` register value, as originally suggested.

7 Conclusion

7.1 Summary

Chapter 5 closed with an evaluation of performance impact and event log bandwidth experienced when capturing non-deterministic events in Xen's paravirtual machine interface. Especially regarding the runtime performance impact of generating determinant logs, the results appear very encouraging. For CPU-intensive applications, a performance downgrade of 0.65% appears almost negligible. Given the task at hand, 5.7% in SMP configurations for I/O-intensive workloads measured so far remains defensible.

Nonetheless, the task of tracing any input to guest systems generates data streams of considerable volume. More than performance implications, bandwidth requirements are of interest. Whether applied for debugging, profiling or high-availability purposes, the limits of determining non-determinism at runtime will significantly impact usability as soon as a captured determinant log volume per time approaches the limits of storage facilities or network interconnects to which it will be forwarded. As one would intuitively expect, the volumes reported in section 5.10 are largely dominated by guest I/O, and therein by user data. One may argue that many of the server system in use today are sufficiently equipped with fully networked I/O resources, such as network-attached storage. In such setups, guest system input bandwidth experienced and bandwidth available for event log transport may remain in balance.

Beyond quantitative measures, chapter 5 described general technique to integrate trace generation into the VMM and surrounding I/O virtualization infrastructure. Clearly, the architectural design principles presented throughout chapter 5 have mostly been influenced by the specific architecture of the Xen hypervisor, most notably the need for controlling shared state accesses across address space boundaries. SMA channels, which export DMA-like controllable data transfers to the guest system interface, clearly belong into this category.

On the other hand, some of the ideas developed appear applicable to a broader range of virtualization solutions. Most notably the concept of back-buffering state changes for deferred delivery inside the bare hypervisor, while actually a

simple idea, contributed much to a desirable degree of simplicity and comparative non-intrusiveness when integrated within the overall VMM architecture. Its applicability does not depend on I/O virtualization delegated to guest systems. This means that it could be equally applied to monolithic systems.

However, these techniques clearly depend on a paravirtual machine interface building upon data exchange in shared memory. This design is not an obvious one. The far more intuitive choice is an interface design rather inspired by OS ABIs, i.e. depending on hypercalls. Interestingly, both the competing VMware VMI and Microsoft's Hyper-V specifications [10, 60] belong into the latter class. Many resources which Xen mapped to shared memory are implemented as hypercalls in VMI and Hyper-V, including e.g. timer inquiries or interrupt control. A function-based interface to I/O facilities has to resort to bypassing original VMM functionality to achieve consistent replay among replica instances, thereby would add more unwanted changes to the core VMM than the architecture presented here.

Chapter 6 proposed to use customary x86 performance monitoring facilities as a compensatory instruction counter. Since this usage remains beyond specifications from processor manufacturers, the results could hardly be guaranteed to be sufficient for highly-available applications in a mission-critical environment. This does not mean that such usage is not possible, nor that the implementation described is insufficient. However, it should be noted that real-world scenarios would require long-term testing and quality assurance, far beyond the degree of stability required for experimental purposes, such as the ones pursued throughout this thesis. However, the implementation is based on insights largely beyond available processor documentation and reflects a time-consuming engineering effort even for prototypical system development. It therefore remains as a reference for future research in the same or similar areas.

Last but not least, one contribution of this thesis is a demonstration that a more reliable and versatile true hardware instruction counter in commodity processor architectures would pave the way for a considerable number of useful applications, effectively and efficiently. With virtual machines, the time for software-level deterministic replay has finally come.

7.2 Future Research

The overall tracing facility implemented and studied in 5 presently covers a major subset of the entire paravirtual machine interface defined by Xen's paravirtual machine layer.

Smaller "leaks" remain, but should not compromise on the overall reliability of basic performance and bandwidth measurements presented in section 5.10. One example of an I/O facility not included in above measurements is the XenStore configuration service described in section 4.8. However, the data volume typically transferred is used only for system and device initialization. It would only account for a few kilobytes of additional input data to the target system.

Processor models in support of concurrent guest instantiations were assumed to be entirely homogeneous, which need not strictly be the case in practice. Xen's binary guest interface includes emulated access to various control interfaces, exposing detailed information about real processor features and configuration items. One example is the `cpuid` instruction, which can be used to inquire about various ISA extensions available. In less experimental implementations, heterogeneous setups announcing strictly equal feature sets would be needed. A negotiation phase between leader and follower hosts may limit the interface to a least common denominator.

Some questions remain. Most notably, tracing network I/O in future revisions should provide additional insights. While chapter 5 showed that overall bandwidth limitations of device I/O capturable can also be derived for the storage interface, maximum bandwidth achievable cannot substitute the more flexible message-passing interface paravirtual network I/O implements. Most notably, the effect of SMA on communication *latency* remains to be evaluated in more detail. Request/response latencies for user datagrams of various sizes sent over a network link would fall into this category.

Beyond completeness, there are some new advancements in hardware and system software design which promise exciting opportunities to further improve the performance of the solution described here. This is discussed in the following two sections.

7.2.1 Time-Travelling Storage Systems

One recent field of research is storage system architecture in support of a virtual machine's capability to roll back and resume from past execution state. These typically target checkpoint/restore facilities, as well as the capability to resume ("fork") multiple instances of virtual machines [92]. In Xen/VLS, the present strategy of unilaterally capturing and logging all user data accounts for much of the log volume generated. Input from storage systems, however, outlives an execution unless data read is subsequently overwritten by the executing guest system, and then is recoverable at the original source.

Recoverability of past system state ("time-traveling") for storage systems has been subject of *Parallax*, a distributed storage system which allows creation of forks and snapshots on block device images [90]. Snapshots are independently accessible disk images sharing a common ancestor. Starting from a given snapshot, every subsequent write operation leaves original snapshot state recoverable. To remain efficient, the system employs copy-on-write techniques within its underlying block allocator.

When used for deterministic replay, initialization of a leader instance would create a snapshot of all associated disk images before resuming execution. Replay at a later point in time can then restart an original computation from the original device state. Especially for applications reading large volumes of data, but with low write performance, e.g. web servers, storage cached on both leader and follower nodes may enable a considerable fraction of guest input be omitted from the replay event log stream. Data could then be served from cached snapshots instead.

7.2.2 DMA Engines

Another potential field of research are hardware DMA engines. A considerable fraction of the downgrade in peak guest system performance and I/O throughput is likely due to sampling input from guest memory. Resorting to transfer mechanisms implemented in hardware may therefore save precious processor cycles and additionally reduce some of the overall cache footprint of the present solution, which is in software.

Some recent x86 chipset integrate a host DMA engine, envisioning to offload packet transfers between host and device memory in upcoming 10GB networks ("TCP onload" engines) [68]. The same DMA controller architecture could be used for memory-to-memory transfers during event capturing.

Similarly, network interfaces controllers (RNICs) capable of remote direct memory access (RDMA) [67], such as InfiniBand [43], can aid both in event capturing and network transport. RDMA is an industry standard whose main motivation is to permit zero-copy data transfers between applications on remote memory subsystems, bypassing not only a host processor for copying data, but the entire operating system when initiating such transfers.

To achieve true zero-copy event logging, data should be read from target guest memory. In theory, this is possible. However, the differences in how a suitable SMA controller implementation has to handle transfers on behalf of driver domains should not be underestimated. Any DMA variant, whether

host-bound or network-oriented, proceeds asynchronously from host processor execution. While the CPU cycles thereby saved are the sole reward to be gained, resuming guest execution before transfer completion risks correctness, if sampled input may concurrently be overwritten. Section 5.5.1 outlined page protection to synchronize guest access to shared memory region with concurrent update initiators. Similar mechanisms could be used to transparently write-protect guest input, synchronizing guest write accesses with transfer completion. But as already discussed in above section, this approach comes again at its own cost.

A tradeoff would be synchronous copies of the event log to avoid guest interference, as performed in the present version, then utilizing DMA for further processing of data becomes available to `dom0`.

7.2.3 Deterministic Multiprocessor Replay

While scalability on multiprocessor systems has been a major concern during development of the techniques proposed here, deterministic replay of multiprocessor guest systems on Xen has not been investigated yet.

Arguably, controlling non-determinism in multiprocessor VMs entails significant additional complexity. Section 2.2.5 identified arbitrary interleavings of memory accesses by concurring hardware threads as a major source of non-determinism on multi-processors. Hence, strict memory access ordering must be performed when tracing execution, then maintained during replay in order to guarantee consistent. Mechanisms in either software or, to speed up the determination of a particular ordering, processor hardware may be employed in order to perform such a task. The *Flight Data Recorder* was one research project which investigated an efficient mechanism implemented in hardware. FDR took advantage of present cache coherence hardware to generate and log replayable data about thread ordering [96].

Lacking similar facilities in contemporary commodity hardware, implementation and optimization of existing software mechanisms represents the foreseeable future. As noted above, shadow paging is a powerful basis in order to let the VMM take control over guest memory accesses in a transparent fashion. Memory access protection then may be used to determine and later enforce a thread order. A sufficiently general approach has first been described by Mellor-Crummey and LeBlanc [57]. The protection protocol they proposed is called *CREW*, short for *concurrent read, exclusive write*. It is defined similarly to a reader-writer lock in concurrent programming, but denotes a state assigned to entire page frames.

- Concurrent read: Arbitrary numbers of processors may read the same

page, but no other processor write to it.

- Exclusive write: Only one processor may gain read/write permission at any time.

Largely dealing with I/O, the SMA mechanism developed in chapter 5 will not interfere with this model. Part of the update types summarized in section 5.7.3, such as mutual exclusion, could be straightforwardly carried over to multiple target processors. In fact, what mutual exclusive SMA operations do – temporarily excluding *any* potential target processor from guest state access – essentially promotes same the single-writer policy demanded by the CREW protocol.

However, mutual exclusion remains the strongest and most costly technique available. Many data items, such as a relevant subset of the `shared_info` structure, are globally accessible but essentially belong to only a single virtual processor. Similar in spirit to the sampling technique employed for I/O memory updates, relaxation of memory access atomicity in favor of exploiting intrinsic knowledge and cooperation on the side of guest kernel software may remain key to a multiprocessor-capable SMA mode.

Bibliography

[1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer*, pages 93–113. USENIX Association, 1986.

[2] R.J. Adair, R.U. Bayles, L.W. Comeau, and R.J. Creasy. A virtual machine for the 360/40. ibm corp. *Report No. G320-2007*, May 1966.

[3] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, 2006. VMware, Inc., ACM Press.

[4] Advanced Micro Devices. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, February 2005. `http://developer.amd.com/`.

[5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Vol.2 – System Programming, Revision 3.13*, July 2007. `http://developer.amd.com/`.

[6] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Vol.3 – General Purpose and System Instructions, Revision 3.13*, July 2007. `http://developer.amd.com/`.

[7] Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h Processors*, December 2007. `http://developer.amd.com`.

[8] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some Constraints and Tradeoffs in the Design of Network Communications. *SIGOPS Operating Systems Review*, 9(5):67–74, 1975.

[9] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12(2/3):72–109, 1994.

[10] Zachary Amsden, Daniel Arai, Daniel Hecht, and Pratap Subrahmanyam. *Paravirtualization API Version 2.5.* VMware, Inc., February 2006.

[11] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[12] Michael Barborak and Miroslav Malek. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, 25(2), June 1993.

[13] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, 2003. ACM Press.

[14] P.A. Barret, A.M. Hilborne, P.G. Bond, D.T. Seaton, P. Verissimo, L. Rodrigues, and N.A. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *Digest of Papers, 20th International Symposium on Fault Tolerant Computing*, pages 481–488, June 1990.

[15] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[16] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop® Advanced Architecture. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 12–21, Washington, DC, 2005. IEEE Computer Society.

[17] Anupam Bhide, E. N. Elnozahy, and Stephen P. Morgan. A Highly Available Network File Server. In *USENIX Winter*, pages 199–206. USENIX Association, 1991.

[18] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 3 edition, November 2005.

[19] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[20] T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 82–83, Los Alamitos, CA, 1987. IEEE Computer Society.

[21] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determin-

ing global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[22] 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 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[23] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems – Concept and Design.* Pearson Education Ltd., Edinburgh Gate, Harlow, Essex CM20 2JE, England, 3 edition, 2001.

[24] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture – A Hardware/Software Approach.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[25] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[26] Jeff Dike. A user-mode port of the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference.* USENIX Association, October 2000.

[27] Fred Douglis and John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8):757–785, 1991.

[28] 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. *SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[29] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[30] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level resource management. *SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[31] Renato Figueiredo, Peter A. Dinda, and Jose Fortez. Resource Virtualization Renaissance. *Computer*, 38(5):28–31, May 2005.

[32] Keir Fraser, Steve Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System*

and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA, October 2004.

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

[34] Robert P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the Workshop on virtual computer systems*, pages 74–112, New York, 1973. Honywell Information Systems and Harvard University, ACM.

[35] Robert P. Goldberg. Survey of Virtual Machine Research. *Computer*, 7(6):34–45, June 1974.

[36] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, 1982. ACM Press.

[37] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2):9–15, February 1995.

[38] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS-X)*, June 2005.

[39] Hewlett Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 3 edition, February 1994.

[40] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Karmean, Alan Kyler, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001. `http://www.intel.com/technology/itj/q12001/pdf/art_2.pdf`.

[41] Matsumoto Hitoshi. Scsi support on xen. Talk at Xen Summit Fall 2007, November 2007.

[42] IEEE Computer Society and The Open Group. *IEEE Std. 1003.1, ISO/IEC 9945: Standard for Information Technology – Portable Operating System Interface (POSIX)*, 2004.

[43] InfinBand Trade Association. `http://www.infinibandta.org/`.

[44] Intel Corporation. *8237A High Performance Programmable DMA Controller (8237A-5)*, September 1993. `http://developer.intel.com`.

[45] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 2: Instruction Set Reference*, September 2006. `http://developer.intel.com/`.

[46] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3: System Programming Guide*, October 2006. `http://developer.intel.com/`.

[47] Intel Corporation. *Intel® Itanium® Architecture Software Developer's Manual, Vol.2: System Architecture, Revision 2.2*, January 2006. `http://developer.intel.com/`.

[48] Intel Corporation. *Intel® Trusted Execution Technology – Preliminary Architecture Specification and Enabling Considerations*, August 2007. `http://developer.intel.com/`.

[49] International Commitee for Information Technology Standards. Technical Committee T11 (FibreChannel). `http://www.t11.org/`.

[50] Richard P. LaRowe Jr. *Page Placement For Non-Uniform Memory Access Time (NUMA) Shared Memory Multiprocessors*. PhD thesis, Dept. of Computer Science, Duke University, March 1991.

[51] Dan Kegel, Doug Lea, Miles Sabin, and Matt Welsh. *JSR-000051: New I/O APIs for Java(TM) Platform Specification*, May 2002. `http://jcp.org/`.

[52] R. E. Kessler and Mark D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[53] Yousef A. Khalidi, Josep Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A Multi Computer OS. In *USENIX Annual Technical Conference*, pages 191–204. USENIX Association, 1996.

[54] Avi Kivity. Kernel Based Virtual Machine. `http://kvm.qumranet.com/kvmwiki`.

[55] Ruppert. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith. Transparent TCP Connection Failover. In *2003 International Conference on Dependable Systems and Networks (DSN'03)*, page 383, Los Alamitos, CA, 2003. IEEE Computer Society.

[56] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[57] T.J. Leblanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *ACM Transactions on Computers*, C-36(4):471–482, April 1987.

[58] Mark Lord, Tomi Leppikangas Benjamin Benz, Leonard den Ottolander, and OTHERS. Linux `hdparm` utility. `http://www.sourceforge.net/projects/hdparm/`.

[59] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 78–86, New York, 1989. ACM Press.

[60] Microsoft Corp. *Hypervisor Functional Specification v0.83*, March 2007. `http://www.microsoft.com`.

[61] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[62] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[63] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 391–394, Boston, MA, April 2005. VMware, Inc.

[64] David A. Patterson and John L. Hennessy. *Computer Organization and Design – The Hardware/Software Interface*. Morgan Kaufmann Publishers, San Francisco, CA, 3 edition, 2005.

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

[66] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, 4(1):36–47, 1994.

[67] RDMA Consortium. `http://www.rdmaconsortium.org/`.

[68] Greg Regier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):46–56, November 2004.

[69] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, 2000. USENIX Association.

[70] Jose Renato Santos, John Janakiraman, and Yoshio Turner. Xen Network I/O – Performance Analysis and Opportunities for Improvement. Xen Summit Spring 2007, April 2007.

[71] Jose Renato Santos, John Janakiraman, Yoshio Turner, and Ian Pratt. Netchannel 2 – Optimizing Network Performance. Talk at Xen Summit Fall 2007, November 2007.

[72] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. *Internet Small Computer Systems Interface (iSCSI)*. IETF Network Working Group, April 2004.

[73] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, 1990.

[74] Mendel Rosenblum Scott W. Devine, Edouard Bugnion. Virtualization System including a Virtual Machine Monitor for a Computer with a Segmented Architecture, October 1998. U.S. Patent No. 6397242.

[75] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[76] Brinkley Sprunt. Pentium 4 Performance-Monitoring Features. *IEEE Micro*, 22(4):72–82, July 2002.

[77] Rob Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[78] SPEC CPU Subcommittee and original program authors. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, 34(4), September 2006.

[79] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001. VMware, Inc.

[80] Sun Microsystems, Inc. *UltraSPARC Architecture 2005, Draft D0.9*, privileged edition, May 2007. `http://www.sun.com/processors/`.

[81] Sun Microsystems, Inc. *UltraSPARC Architecture 2005, Draft D0.9*, hyperprivileged edition, May 2007. `http://www.sun.com/processors/`.

[82] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc., Upper Saddle River, New Jersey 07458, 2 edition, 2001.

[83] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):38–56, May 2005.

[84] Max Walter und Winfrid G. Schneeweiss. *The Modeling World of Reliability / Safety Engineering*. LiLoLe Verlag, Hagen, Germany, 2005.

[85] VMware, Inc. *VMware ESX Server 2 Architecture and Performance Implications*, August 2005.

[86] John von Neumann. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, June 1945.

[87] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5 USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002. VMware, Inc.

[88] Bruce Walker. Open Single System Image (OpenSSI) Linux Cluster Project. Published on the Web. `http://www.openssi.org/`, 2004.

[89] Bruce J. Walker, Laura Ramirez, and John L. Byrne. Clusterproc: Linux Kernel Support for Clusterwide Process Management. In *2005 Linux Symposium*, pages 251–264, 2005. `http://www.linuxsymposium.org/2005/`.

[90] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS-X)*, June 2005.

[91] Joe Wetzel, Ed Silha, Cathy May, and Brad Frey. *PowerPC™ Architecture Book III: Operating Environment Architecture, Version 2.01*. IBM Corporation, December 2003. `http://www.ibm.com/chips/power/`.

[92] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, March 2007.

[93] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002. University of Washington.

[94] Thomas Wolf. Transparent Replication for Fault Tolerance in Distributed Ada 95. *Ada Letters*, XIX(2):33–40, June 1999.

[95] XenSource Inc. *Xen Interface Manual*, Xen 2.0 for x86 edition, 2004. `http://www.xensource.com/`.

[96] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003.

[97] Min Xu, Vyacheslav Malyugin, Jeff Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.