Technical Report
NWU-EECS-10-07
April 26, 2010

Comparing Approaches to Virtualized Page Translation in Modern VMMs

Chang Bae John R. Lange Peter A. Dinda

Abstract

Virtual machine monitors for modern x86 processors can use a variety of approaches to virtualize address translation. These include the widely-used software approach of shadow paging, with and without caching or prefetching, as well as the widely used hardware approach of nested paging. We compare and analyze the measured performance of these approaches on two different VMMs, KVM and Palacios, under a variety of different workloads on physical hardware (AMD Barcelona). We find that which approach is best is very dependent on the workload, and the differences can be quite significant. The difference between the worst and best approach can be as much as a factor of 53, with typical differences being a factor of 2. The difference between best software approach and nested paging can be as much as a factor of 3, with typical differences being small. These variations suggest that an adaptive approach to virtualizing address translation is likely to be beneficial.

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-0709168, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory on grant DE-AC05-00OR22725. John Lange was partially supported by a Symantec Research Labs Fellowship.

Keywords: virtual machine monitors, page translation, nested paging, shadow paging

Comparing Approaches to Virtualized Page Translation in Modern VMMs

Chang Bae   John R. Lange   Peter A. Dinda
Department of EECS; Northwestern University

Abstract

Virtual machine monitors for modern x86 processors can use a variety of approaches to virtualize address translation. These include the widely-used software approach of shadow paging, with and without caching or prefetching, as well as the widely used hardware approach of nested paging. We compare and analyze the measured performance of these approach on two different VMMs, KVM and Palacios, under a variety of different workloads on physical hardware (AMD Barcelona). We find that which approach is best is very dependent on the workload, and the differences can be quite significant. The difference between the worst and best approach can be as much as a factor of 53, with typical differences being a factor of 2. The difference between best software approach and nested paging can be as much as a factor of 3, with typical differences being small. These variations suggest that an adaptive approach to virtualizing address translation is likely to be beneficial.

1 Introduction

Virtual machine monitors (VMMs) must virtualize the address space of guest operating systems. The guest OS is presented with the illusion of controlling the translations from virtual addresses to physical addresses, however below the guest translations the VMM conceptually adds an additional layer of indirection. What the guest actually controls are the mapping from guest virtual addresses (GVAs) to guest physical addresses (GPAs), while the VMM controls the mapping from GPAs to host physical addresses (HPAs).

On x86 and x86_64 processors from Intel and AMD, VMMs implement two general approaches to these translations. In *shadow paging*, which does not require special hardware support, the VMM essentially flattens the GVA→GPA and GPA→HPA mappings into a GVA→HPA mapping that the VMM implements using the existing paging mechanisms. In effect, the GVA→GPA mappings are maintained as if they are contained in the TLB, an approach described by Intel as a "virtual TLB" [5, Chapter 26]. Guest operations, such as page faults, page invalidations, etc, cause exits from the guest to the VMM and are used to adjust the state of the virtual TLB. The shadow page tables maintained by the virtual TLB are used while the guest executes.

Shadow paging may have high overheads, particularly when using architected hardware virtualization such as Intel VT or AMD SVM since they have quite high costs for entering or exiting the VMM [2], and the virtual TLB may require many exits to maintain, especially when a complex, multitasking workload is running in the guest.

The second general approach used by VMMs is known as *nested paging*, which requires hardware support that is available (in different forms) from both AMD and Intel. In nested paging, the GPA→HPA mapping is made explicit and is supported, at the hardware level, by a second set of page tables. The guest controls the GVA→GPA translations by directly manipulating its own page tables and doing localized TLB invalidations, without causing exits. The VMM controls the GPA→HPA mapping by directly manipulating its page tables, which are visible only to the VMM. However, it should be noted that during a page-walk, the"physical" addresses stored in the guest's page tables must themselves be translated through the VMM's page tables to be meaningful. These translations "nest" inside of the steps of the guest's page-walk. If the page tables are n levels deep, an $O(n)$ page-walk from the guest's perspective can, in fact, turn into an $O(n^2)$ page-walk in the hardware. AMD (and probably Intel) have attempted to address this slowdown by extending page-walk caching (cf.[6]) to "two dimensions" [4].

Even given earlier work showing that with two-dimensional page-walk caching, nested paging has performance that is near-native, it is not obvious that nested paging is always preferable to shadow paging. In particular, shadow paging can also be optimized. One optimization, which has been widely adopted is called *shadow paging with caching*. A key performance hit in shadow paging is that some guest operations, such as a CR3 write, are defined as flushing the TLB, and thus the virtual TLB is flushed. In shadow paging with caching, a much more subtle view of the guest's paging structures is maintained, and this view allows many virtual TLB flushes to be avoided. In effect, the virtual TLB can maintain and switch among numerous guest contexts, selectively flushing entries only as needed. Due to the complex nature of the semantics of x86 paging [7], maintaining the correct behavior of caching given all a guest may do is a significant challenge. Shadow paging with caching has much more implementation complexity in the VMM than either plain shadow paging or nested paging.

Another potential shadow paging optimization, which is not widely implemented, to the best of our knowledge, is *shadow paging with prefetching*. Here, the idea is to maintain the simple virtual TLB model, but to refill it very quickly after any operation that flushes it. Essentially, this relies on spatial locality of reference. When we see a miss in the virtual TLB (a page fault that causes an exit to the VMM), we fetch not only the relevant page table entry from the guest, but nearby entries as well. Such prefetching is more complex that it appears since we must intercept guest changes to those entries that have been prefetched, but not yet used. The hope is that prefetching will provide many or most of the benefits of shadow paging with caching while avoiding some of its implementation complexity.

A natural question to ask is which of these four approaches is "best"?. The answer is unclear. For example, in previous work in which we considered the virtualization of a parallel supercomputer [10], we found a surprising result. For a particular benchmark, a high speed parallel conjugate gradient solver, shadow paging (without caching or prefetching) performed better than nested paging when a particular guest OS was used. When a different guest OS was used, nested paging performed better.

As far as we are aware, there has been no publication of a detailed performance evaluation of the four approaches to address space virtualization described above when driven by a wide range of workloads. The goal of this paper is to fill this gap. We consider implementations of shadow paging with and without caching, and nested paging in the KVM virtual machine monitor, and implementations of all four approaches in our own Palacios VMM. On both VMMs, we run six different benchmarks, ranging from microbenchmarks to high-level benchmarks such as TPC-H. We measure performance on real hardware (AMD Barcelona).

The primary contributions of our paper are the detailed performance measurements. However, based on these measurements, we can also draw several conclusions, as follows.

- The choice of the best approach for address space virtualization is heavily dependent on the workload. *There is no single best answer.*
- The differences in performance between approaches for a given workload can be significant. The difference between the worst and best approach can be as much as a factor of 53, with typical differences being a factor of 2. The difference between best software approach and nested paging can be as much as a factor of 3, with typical differences being small. *The choice matters.*
- The workload's CR3 write rate and TLB miss rate are strongly predictive of the best approach for a given workload. The CR3 write rate can be measured by the VMM and the TLB miss rate can be measured using the architectural performance counters.
- These differences strongly suggest that an adaptive approach to virtual address translation in the VMM seems like a good idea. Whether such adaptation should be done at startup time or continuously remains to be seen. *Perhaps the choice should be adaptive.*

2 Paging approaches

Paging is the process of converting a process-specific logical address into a system physical address. It is the primary mechanism by which the operating system can isolate the address space of processes and control the allocation of physical memory. In a virtualized environment, paging is complicated because there are essentially two levels of conversion that occur. We now describe the conceptual model that the VMM creates for the guest OS and some approaches to implementing it, shadow paging, shadow paging with caching, shadow paging with prefetching and nested paging.

The VMM uses paging to create what appears to be a physical memory map for the guest OS. On the x86 (or x86_64), for a non-paravirtualized guest, this guest physical memory map typically corresponds to that of a PC platform with a given amount of memory. We can consider the guest physical memory map to consist of *regions*, contiguous runs of pages starting at page boundaries. For historical reasons, a PC memory map is rather complicated, with numerous specialized regions occurring below the 1 MB "line", another specialized region from 1 MB to 16 MB, and others in at higher memory locations (e.g., PCI devices and APICs). For a paravirtualized [3] guest, the memory map can be made simpler as the guest OS has been modified such that backward compatibility is not required.

Some regions are mapped to physical memory, others to physical, memory-mapped devices (passthrough I/O), and still others to virtual, memory-mapped devices implemented by the VMM itself. For the former two kinds of regions, the VMM intends that any guest memory reference to the region be translated and handled by the hardware ideally without VMM intervention. For the latter, it is intended that an exit occurs so that the VMM can emulate the memory access.

Conceptually, in a non-paravirtualized VMM, the guest OS controls the translation from guest virtual addresses (GVAs) to guest physical addresses (GPAs) by manipulating page tables in its address space. In a paravirtualized VMM, these manipulations are made by explicit hypercalls, allowing the VMM to veto or transform the requested manipulations. Conceptually, the VMM controls the translation from GPAs to host physical addresses (HPAs) by manipulating some other structure that implements the mapping. For the first two kinds of regions on the guest physical memory map, mappings are stored in the structure. For the third kind, the lack of a mapping in the structure causes the desired exit.

2.1 Shadow paging

Shadow paging is virtualization support for paging that is implemented in software. To understand shadow paging, it is helpful to differentiate the privilege level of the guest page tables and the VMM page tables. The VMM is more highly privileged and thus has ultimate control over the control registers used to control the normal paging hardware on the machine. Because of this, it can always assure that the page tables in use are the page tables it desires. These page tables, the shadow page tables, contain mappings that integrate the requirements of the guest and the VMM. The shadow page tables implement a mapping from GVA to HPA and are in use whenever the guest is running.

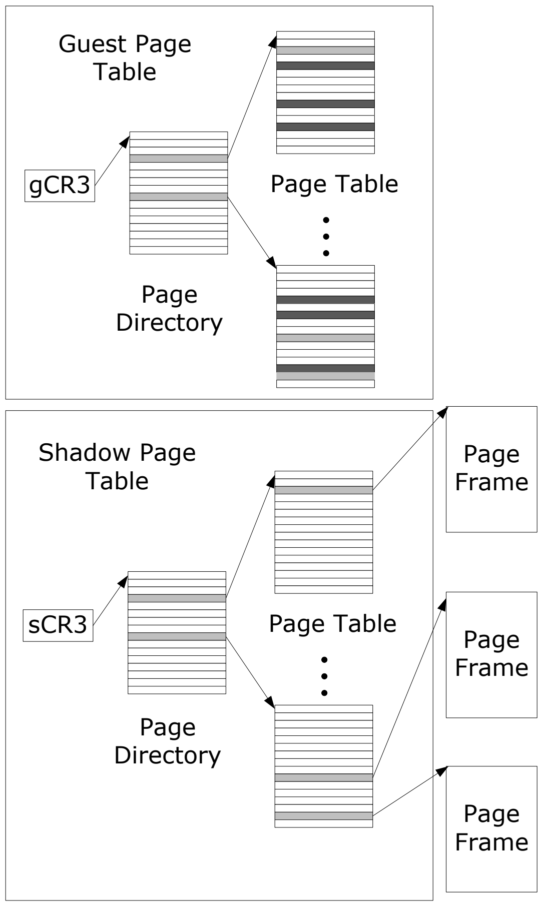


Figure 1: Shadow paging

The VMM must maintain the shadow page tables' coherence with the guest's page tables. A common approach to do so is known as the virtual TLB model [5, Chapter 26]. The x86's architected support for native paging requires that the OS (guest OS) explicitly invalidate virtual address (GVAs) from the TLB and other page structure caches when corresponding entries change in the in-memory page tables. These operations (including INVLPG and INVLPGWB instructions, CR3[1] writes, CR4.PGE writes, and others) are intercepted by the VMM and used to update the shadow page tables, in addition to the TLB and paging structures. The implementation must also use page protections so that it can update the accessed bits in the guest page tables as appropriate. The interception of guest paging operations can be expensive as each one requires at least an exit from the guest, an appropriate manipulation of the shadow page table, and a reentry into the guest. A typical exit/entry pair, using hardware virtualization support, requires in

[1]CR3 contains the pointer to the current page table.

excess of 1000 cycles on typical AMD or Intel hardware.

Figure 1 illustrates the state involved in using shadow page tables, using 32 bit, two-level page tables. 64 bit page tables are similar, but four levels are possible. Here, gray entries in guest page table and shadow page table are synchronized, while black entries in guest page table are not reflected in the shadow page tables. A guest access to an address mapped by one of these missing entries in the shadow page table will cause a page fault to which the VMM will respond by updating the entry. On a guest context switch, the guest will write CR3, and in response the VMM will flush the contents of the shadow page tables. As the guest executes, the shadow page tables will be rebuilt, one page table entry at a time.

2.2 Shadow paging with caching

The need to update the shadow page tables by page faults can make shadow paging, as described above, quite expensive, especially when context switches are frequent. Not only is the page fault cost incurred, but also the VM exit/entry cost, and either a partial (e.g., with AMD's ASID tagged TLB) or complete physical TLB invalidation.

Shadow paging with caching attempts to reuse shadow page tables. Ideally, the reuse is sufficient that a context switch can be achieved essentially with only one exit, to change the CR3 value (the shadow CR3, marked as sCR3 in the figures). The VMM maintains in memory both shadow page tables corresponding to the current context, and shadow page tables corresponding to other contexts. The distinction is not perfect—it is perfectly legitimate for the guest to share guest page tables among multiple contexts, or even at different levels of the same context. Furthermore, the guest kernel has complete access to all of the guest page tables, for all contexts, at any time.

While extending shadow page tables with caching is conceptually simple, the implementation challenge lies in being able to correctly invalidate shadow page table entries in guest contexts other than the one currently executing, including in the presence of sharing. When *any* guest page table is modified, the VMM must propagate those changes to the relevant shadow page tables in the cache. Usually, this change takes the form of an invalidation. That is, if the guest modifies a page table entry (PTE), the corresponding PTEs (there may be more than one, since the page may be shared) in the shadow cache must be invalidated. Further, the changes need to be propagated. For example, if a change to a first level page table entry (a "page directory entry" or PDE in x86 terminolgy), lower-level PTEs must also be updated.

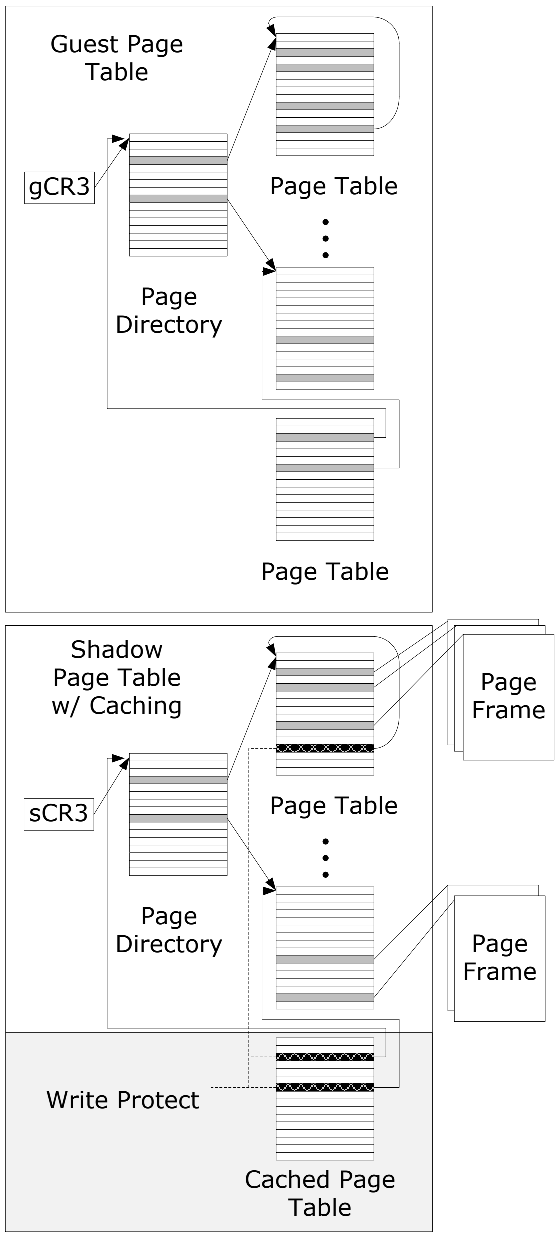


Figure 2: Shadow paging with caching

Notice furthermore that it is perfectly legitimate for the guest to modify a PTE for a context different from the one that is currently in use, *and not immediately invalidate it via an INVLPG instruction.* This is because the guest can assume that the needed TLB flush will happen when there is a switch back to the now modified context, as a CR3 write forces a TLB flush.

To keep the shadow page table cache coherent and synchronized with respect to the guest page tables, the

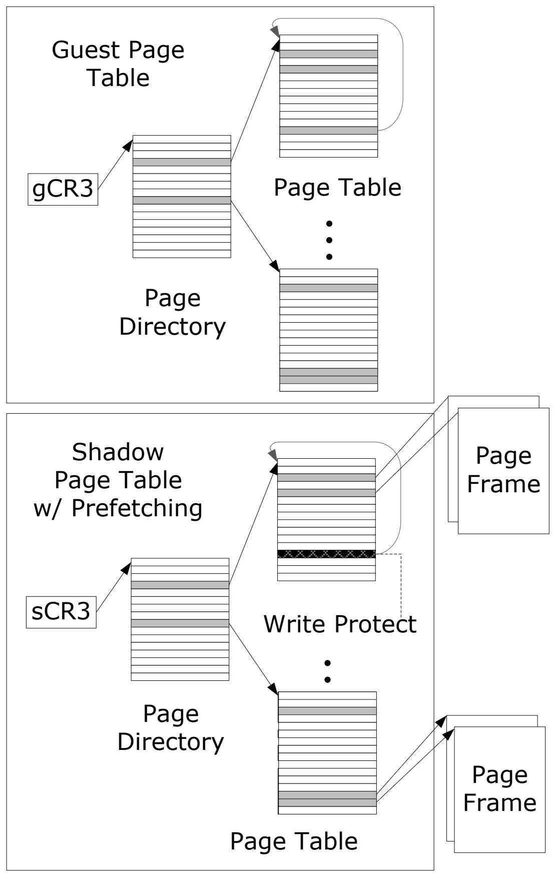


Figure 3: Shadow paging with prefetching

VMM needs to be aware of when any guest page table for which it has a cached shadow representation, is changed. To do this, we write-protect pages used as page tables in the guest by marking their corresponding entries in the shadow page tables as read-only. When the guest modifies one of its page tables, a page fault occurs, and the VMM now has the opportunity to invalidate (or update) relevant entries in the shadow page table cache. It also emulates the relevant write to the guest page table entry, or simply marks the guest page writable and restarts the faulting instruction. These operations are summarized in Figure 2.

Notice that it is not necessarily the case that caching will improve the performance of shadow caching. While it having cached shadow page tables will make context switch costs much lower, it is also the case that we are introducing more exits due to the monitoring of the guest page tables.

2.3 Shadow paging with prefetching

Shadow paging with management according the virtual TLB model updates at most one shadow page table entry per paging-related exit to the VMM. While this has the virtue of simplicity, it means that rebuilding shadow page tables, say on a context switch, involves many exits. Shadow paging with caching attempts to ameliorate this through reuse. We also consider ameliorating it through prefetching, by amortizing many shadow page table entry updates per exit. Figure 3 illustrates how this shadow paging with prefetching works. The basic idea is that when a page fault occurs due to a shadow page table entry being out of sync with the guest page table entry, we update not only the faulting entry, but also all the other entries (1024 entries total for 32 bit machines) on the shadow page table.

Because prefetching does not need to handle sharing from one guest context to another, the hope is that it will provide many of the performance benefits of shadow paging with caching, while having a lower implementation complexity. However, prefetching does need a more complex synchronization scheme. In particular, after a context switch (CR3 write), a guest OS need not execute INVLPG instructions because the write assures an empty TLB according to the architectural specification. Suppose we prefetch all of a guest page table's entries on the first access to a page table entry on it. The guest can then modify one of those other entries without invalidation, but the VMM would be unaware of the change. To handle such cases, the VMM must write protect shadow page table entries mapping guest page tables in order to be aware of these modifications. The infrastructure to capture these modifications is the same as for shadow paging with caching, but the work that is done in response to a modification is much simpler.

2.4 Nested paging

Nested paging is a hardware mechanism that attempts to avoid the overhead of the exit/entry pairs needed to implement shadow paging by making the GVA→GPA and GPA→HPA mappings explicit and separating the concerns of their control, making it possible to avoid VMM intervention except when a GPA→HPA change is desired. Both AMD and Intel have developed variants of nested paging.

In nested paging, the guest page tables are used in the translation process to reflect the GVA→GPA mapping, while a second set of page tables, visible only to the VMM, are used to reflect the GPA→HPA mapping. Both the guest and the VMM have their own copy of

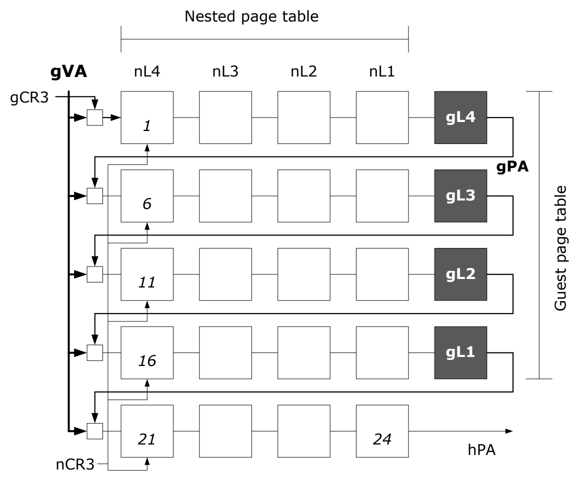


Figure 4: Nested paging

the control registers, such as CR3. When a guest tries to reference memory using a GVA and there is a miss in the TLB, the hardware page-walker mechanism performs a two dimensional traversal using the guest and nested page tables to translate the GVA to HPA. This process is shown in Figure 4, adopting the style of Bhargava, et al [4]. When the page walk completes, the result is that the translation is cached in the TLB. Intermediate results may also be cached in a structure referred to as a page-walk cache (PWC) by AMD and a paging-structure cache (PSC) by Intel.

We now explain the page walk in more detail, using Figure 4. The guest CR3 register provides a pointer to the guest page tables. This pointer is a GPA. Using this GPA, the page-walker goes through as many as 4 levels of nested page tables (nL1 to nL4) before the HPA is obtained. This is the actual physical address of the top level guest page table. This HPA along with the next chunk of the original address provides us with the appropriate entry in that table. This entry is again a GPA, and thus needs to be translated through a nested page walk to arrive at an HPA for the second level guest page table. This process continues through the next two levels of guest page tables. At this point, the original guest virtual address has been translated to a final guest physical address. This is mapped through the nested page tables to arrive at the final host physical address.

A TLB miss under nested paging incurs a very heavy cost, as compared to all forms of shadow paging. This is because of the two dimensional walk we have just described. A four level guest page table walk could invoke up to 5 accesses to the nested page tables for every level of guest page tables. This means, the cost for a TLB miss

	KVM	Palacios
VMM + shadow paging	47037	30458
VMM + shadow paging with caching	48368	31558
VMM + shadow paging with prefetching	-	31655
VMM + nested paging	46952	28593

Figure 5: Implementation complexity of different approaches in the KVM and Palacios VMMs. Measurement is in lines of code as measured using wc. Shadow paging with prefetching is not implemented in KVM.

is 24 memory references, providing no other support.

Recently, researchers at AMD have developed approaches to mitigate the cost of these multi-level nested page walks [4]. The concept of a page walk cache is extended to be "two dimensional", allowing for significant reuse in page-walks. The upshot of this is that the expected number of memory references in a nested page walk can be considerably lower than 24. The authors also introduced a nested TLB, which is a dedicated guest physical address to host physical address TLB. This nested TLB can "short circuit" the rows in the page walk shown in the Figure. Combining the two dimensional page walk cache and the nested TLB results in $86\% - 93\%$ of native (no virtualization) performance for a range of server, integer and floating benchmarks. While these improvements are impressive, in the worst case, a page walk can still result in up to 24 memory references. The authors do not consider the effects of context-switching among multiple guests.

3 Experimental setup

We now describe our experimental setup, beginning with the implementations of the four paging virtualization approaches used in the KVM and Palacios VMMs, and then describing the machines we used to measure performance and the emulator we used to get a finer grain view of behavior.

3.1 Implementations in KVM

The Kernel-based Virtual Machine (KVM) [8] is a Linux subsystem that allows virtual machines to be instantiated directly on top of Linux. KVM supports hardware virtualization extensions from AMD [1] and Intel [13]. It supports full system virtualization.

We started from the first version of KVM that supported AMD's nested paging, KVM 61. This version of KVM also supports shadow paging without caching in addition to shadow paging with caching, although not by default. Newer versions of KVM have retired support for shadow paging without caching. Using KVM 61 allows us to readily evaluate the paging virtualization approaches described in Section 2 in a modern, widely used VMM, with the exception of shadow paging with prefetching.

It is important to illustrate the complexity of the different approaches, which we do in Figure 5. Here, we show the lines of code using `wc` over the core KVM 61 implementation, excluding the QEMU-derived I/O back-end. As we can see, the least complex implementation is for nested paging, with shadow paging being close behind. Caching adds over 1300 lines of code.

3.2 Implementations in Palacios

Our implementation is in the context of the Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available type-I VMM designed as part of the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University and the University of New Mexico, with close collaboration with Sandia National Labs for our efforts in the virtualization of supercomputers. Detailed information about Palacios can be found elsewhere [10, 9, 17] with code available from our site.

Palacios achieves full system virtualization for x86 and x86_64 hosts and guests using either the AMD SVM or Intel VT hardware virtualization extensions and either shadow or nested paging. Currently, the entire VMM, including the default set of virtual devices is on the order of 47 thousand lines of C and assembly. When embedded into Sandia National Labs' publicly available, GPL-licensed Kitten lightweight kernel, as done in this work, the total code size is on the order of 108 thousand lines. Palacios is capable of running on environments ranging from commodity Ethernet-based servers to large scale supercomputers, specifically the Red Storm Cray XT supercomputer located at Sandia National Labs.

We have added shadow paging with caching as well as shadow paging with prefetching to Palacios. Previous versions of Palacios already supported plain shadow paging and nested paging. Our implementation of shadow paging with caching is similar to KVM's, and our implementation of shadow paging with prefetching uses the same invalidation detection logic that the caching implementation uses. Prefetches involve the entire faulting page table (1024 entries in 32 bit mode).

It is important to point out that KVM and Palacios are configured to handle TLB updates due to exits differently in this study. KVM uses AMD's *Address Space ID* (ASID) construct for tagged TLBs, including selective flushing on VM exit/entry. In this study, Palacios is configured not to use ASIDs, and to do TLB flushes on VM exit/entry.

Figure 5 shows implementation complexity as lines of code for Palacios. Virtual devices are not counted, similar our exclusion of QEMU back-end code from the KVM numbers. As with KVM, nested paging has the simplest implementation. Unlike with KVM, shadow paging requires considerably additional code (about 2000 lines). Adding caching requires an additional 1100 lines. Here, prefetching is implemented in the caching framework, so we cannot measure its complexity directly. However, it is interesting to see that only an addition 100 or so lines of code needed to be added to shadow paging with caching.

In Palacios, we also implemented an approach that, while not intended to be practical, helps us to measure costs. This is referred to as *shadow paging with synchronization*. This approach is essentially plain shadow paging but with the page table monitoring needed for shadow paging with caching or prefetching. This lets us decouple the costs of page table monitoring from the costs of maintaining the caching or prefetch structures.

3.3 Hardware testbed

Our experiments were performed on a Dell PowerEdge SC1450 that has an AMD Opteron 2350 processor with 2GB of RAM. This model of AMD CPU is based on the Barcelona architecture, which supports nested paging. Both KVM and Palacios are measured on this machine. We also conducted some measurements on a machine with an AMD Opteron 3276 processor, which implements the newer Shanghai architecture. We would like to compare nested paging with and without the two dimensional page walk caching described earlier. However, we have been unable to determine which specific AMD processors support this new optimization.

3.4 Simulation testbed

Beyond determining "which is best" for a given workload, we would also like to identify the characteristics of the workloads that correlate with performance in different approaches. To characterize our workloads, we examined microarchitectural metrics. Basically we tried to figure out the frequency of memory accesses by looking at cache transactions and miss rates. Furthermore, since MMU systems are virtualized in our work, we also

CPU	a generic 64-bit AMD Athlon 64 (Opteron processor w/o on-chip devices)
Cache	16KB (line number: 256, line size: 64, associativity 2, replacement: LRU)
TLB	DTLB: 64 entries 4K (associativity: 4) 64 entries 4M (associativity: 4) ITLB: 64 entries (associativity: 4) 64 entries 4M (associativity: 4)
Memory	1.5GB

Figure 6: Architectural configuration of SIMICS used to characterize workloads.

	TLB miss	CR3 write	Memory accesses per instruction
Worst case	High	Low	0.91
Boot	Low	-	0.28
SpecCPU VPR	High	Low	0.52
HPCC	-	High	0.58
TPC-H	Low	-	0.60
Kernel compile	-	-	0.52

Figure 7: Overall characteristics of the workloads.

examined TLB behavior such as TLB misses, fills, invalidates, and replacements. The rate of context switches at the guest level is also important.

To measure such microarchitectural metrics, we used the SIMICS full system simulator [16, 12]. The specific SIMICS configuration we employed is presented in Figure 6.

4 Workloads

We employed a range of different workloads to evaluate the virtualization approaches implemented in the VMMs. The same benchmark binaries were used in all cases, with the exception of kernel compilation. For KVM, the guest ran Fedora Core 5 (Linux kernel 2.6.15 for 64 bit), while for Palacios, Puppy retro 3.01 (Linux kernel 2.6.18 for 32 bit) was used. In both cases, workloads were scaled to available guest physical memory to avoid I/O operations as much as possible. It is important to point out that due to these differences, and others, performance cannot be compared between VMMs.

Our purpose in considering both 32 and 64 bit guests was to try to capture the different page walk behavior in nested paging, in particular the quadratic growth in page walk depth when we move from 32 bit (2 level page tables) to 64 bit (4 level page tables).

We used SIMICS to carefully characterize our work-

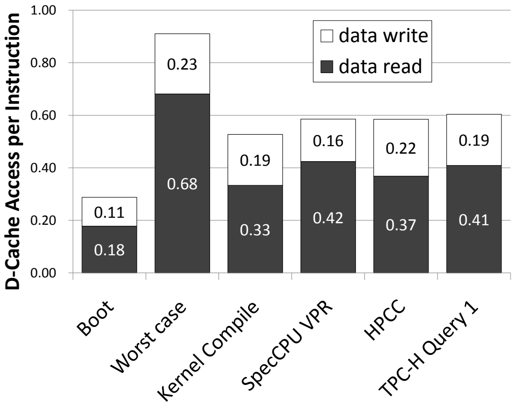


Figure 8: Characteristics of workloads: cache accesses per instruction.

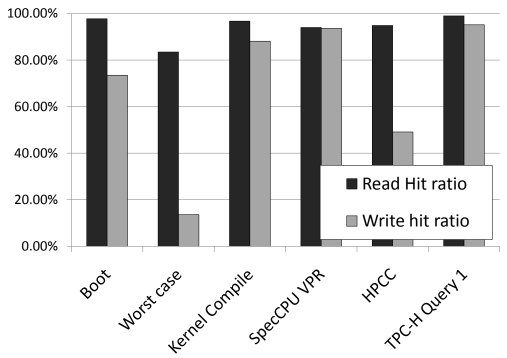


Figure 9: Characteristics of workloads: cache misses per instruction.

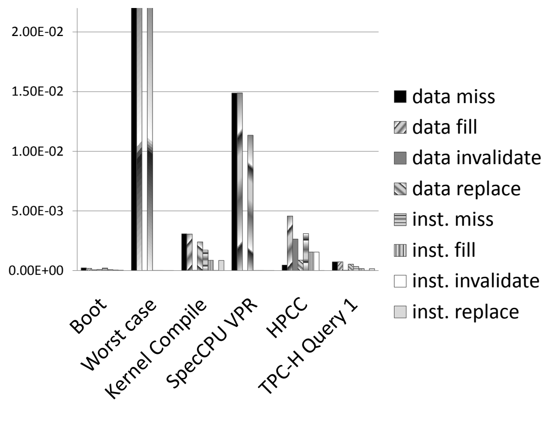


Figure 10: Characteristics of workloads: TLB misses per instruction.

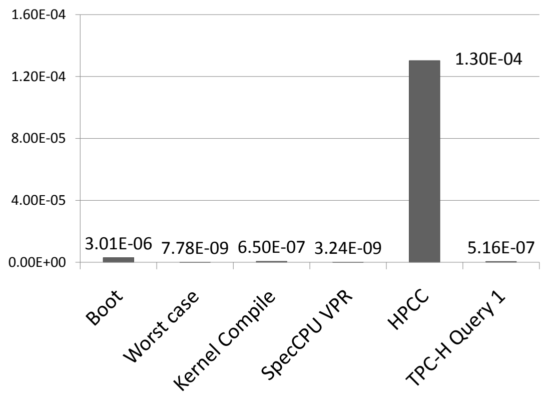


Figure 11: Characteristics of workloads: CR3 writes per instruction.

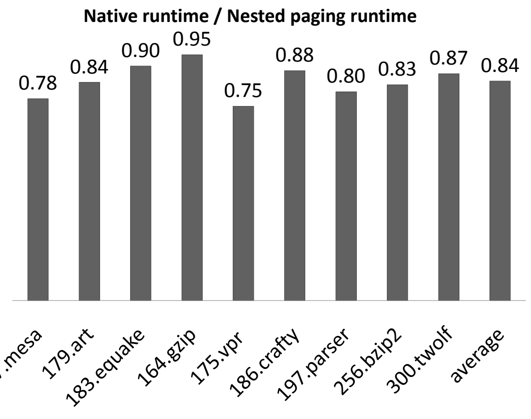


Figure 12: Performance comparison of SpecCPU 00: Native versus Nested paging

loads. Probably the most important figure summarizing the characteristics of the workloads is Figure 7. As we will see the characteristics shown there (in particular the characterizations of the TLB miss rate and the CR3 write rate) are predictive for which paging approaches are likely to work well.

Figure 8 shows the overall ratio of cache accesses per instruction, broken down by data reads and writes, while Figure 9 presents the cache hit rates. Memory access operations are dominant, except for kernel compilation.

Figure 11 shows the frequency of CR3 writes per instruction, which is essentially the context switch rate. Figure 10 shows the TLB misses per instruction. Note that CR3 writes and (virtual) TLB misses are what drive shadow paging approaches.

4.1 Worst case

Worst case is a microbenchmark that we wrote to strike a pain point for nested paging. It has a reference pattern that touches data such that there is minimal spatial and temporal locality both for the accessed data and for the page tables that support it. That is, it tries to maximize both data cache miss rate and TLB miss rate.

4.2 Boot

The boot benchmark measures the time to boot the guest OS into interactive mode, including both kernel bootstrap and init. Different from the other workloads, except kernel compilation, this workload results in the guest using large pages. Also, the memory access rate during execution is quite low compared to the other workloads. This helps us to consider the effect of memory access rate on the performance of the different approaches.

4.3 SPEC CPU 2000 VPR

The SPEC CPU 2000 1.2 suite [14] contains computationally intensive benchmarks based on real-life applications, some of which have large data footprints which stress memory hierarchy. We evaluated various SpecCPU benchmarks, comparing native execution (without any virtualization) and KVM-based virtualized execution using nested paging. Figure 12 shows the performance of the virtualized benchmarks compared to native execution. The benchmark 175.vpr shows itself one of most affected benchmarks under nested paging, and thus we selected it.

Versatile Place and Route (VPR) is an integrated circuit computer-aided design program, automatically placing and routing logic blocks and I/O pads in a Field-Programmable Gate Array chip. This benchmark has a very low context switch rate, as shown in Figure 11.

In addition to exhibiting the most significant decline in performance when run with virtualization, VPR is also distinguished by its microarchitectural characteristics, as illustrated in the earlier figures. It has a moderate working set size, and memory access rate per instruction. The TLB miss rate is quite high. On the other hand, the CR3 write rate indicates that context switches are infrequent.

4.4 HPC Challenge

HPC Challenge 1.3 [11] is collection of eight benchmarks, each of which measures performance focusing on a different part of the memory hierarchy. HPL and

DGEMM stress computation performance. PTRANS measures data transfer rate. FFT computes discrete Fourier Transform with stresses on interprocess communications. STREAM targets measurement for memory bandwidth and its computation rate. RandomAccess updates some memory location randomly selected and stresses interprocess communication. Section 5 details how this characteristic affects performance results. LatencyBandwidth measures latency and bandwidth of communication patterns. In most of the benchmarks, the memory system is measured with high stressful patterns of accesses.

In contrast with the other workloads, the context switch rate is quite high in the HPC Challenge benchmarks. We expected therefore that this benchmark will best showcase the benefits of shadow paging with caching and nested paging.

4.5 TPC-H

TPC-H is a benchmark that emulates a decision-support system (DSS) [15]. We run it with the scale factor set to one (1 GB) which significantly scales down the working set size to fit into main memory. DSS workloads run at low concurrency level, and DSS queries are table scans or clustered index scans. From the 22 available queries, we selected the pricing summary report query. Prior measurements [18] show that this query is dominated by read and write system operations.

4.6 Kernel compile

Kernel compilation is compute-intensive work and it generates many different processes. Our benchmark measures the elapsed time to do a complete build of minimal configuration of a Linux kernel (2.6.14 for KVM and 2.6.17 for Palacios) tree stored on local ram disk. This workload performs a moderate amount of accesses to small files as well as spending time on process management, hampering performance. As Figure 8 shows, the kernel compile has moderate memory access rate per instruction. It also features moderate TLB miss and context switch rates, as shown in earlier figures.

5 Results

As described in Section 4, our evaluation focused on a set of workloads with largely different behaviors. We ran each benchmark on the hardware testbeds described in Section 3 and measured their performance under each of the combinations of VMM and paging approach. The evaluation resulted in the average, min, and max times from five runs of each workload. We note again that the structure of the experiments does not allow direct comparisons between the two VMMs. We now describe the most important points we found from our measurements, and present the raw data as well.

There is no one best virtual paging approach. Figures 13 (KVM) and 14 (Palacios) summarize our results, showing the best approaches for each combination of VMM and workload. As we can see, the best virtual paging approach is most strongly dependent on the workload.

Expanding on the summary figures, Figure 15 shows the detailed runtime results for each workload in seconds, comparing the four different MMU virtualization approaches for both VMMs. The error bars indicate the minimum and maximum run-times.

The performance differences between the approaches are significant. The summary figures (Figures 13 (KVM) and 14 (Palacios)) also indicate the relative speedups between the best and worst paging approaches for each workload, and the relative speedups between the best shadow paging approach and nested paging. These speedups range from 0.5x to 3x for comparing the best shadow paging approach to nested paging, and from 1.0x to 53x for comparing the best approach and the worst approach overall.

It should also be noted that generally the performance of nested paging versus the best shadow paging approach is much more consistent than the overall best approach versus the worst approach. Nested paging never outperforms the best shadow paging approach by more than a factor of 3 and never under-performs by more than a factor of 2. However when the overall best and worst approaches are compared, the performance range is much larger. As an extreme case the best approach for the HPCC workload demonstrates a speedup of 17(KVM) and 52(Palacios) over the worst approach. This clearly demonstrates how certain workloads can result in radically different performance profiles depending on the type of virtual paging being used.

A workload's CR3 write rate and TLB miss rate are predictive for the best performing virtual paging approach. Figure 7 shows a high level view of how each workload interacts with the memory system. As a reminder, we derived the table from a study where we used SIMICS to collect a set of low-level microarchitectural measurements for insight into the memory access behavior of each workload. When Figure 7 is compared and contrasted with the high-level summary of the results of our hardware performance measurements, given in Fig-

Workload	Best Approach	Speedup (Best / Worst)	Speedup (Best shadow / nested)
Worst case	shadow/shadow+cache	2.56x	2.56x
Boot	nested/shadow/shadow+cache	1.06x	0.94x
SpecCPU VPR	shadow+cache	1.19x	1.19x
HPCC	nested/shadow+cache	17.52x	0.98x
TPC-H	all same	1.03x	0.98x
Kernel compile	nested	1.59x	0.75x

Figure 13: High level comparison of KVM workload results showing the best paging approach, speedups between the best and worst approaches, and speedups between the best shadow approach and nested paging

Workload	Best Approach	Speedup (Best / Worst)	Speedup (Best shadow / nested)
Worst case	shadow/shadow+cache/shadow+prefetch	2.86x	2.86x
Boot	nested	1.57x	1.16x
SpecCPU VPR	shadow/shadow+cache/shadow+prefetch	1.08x	1.08x
HPCC	nested	52.79x	0.75x
TPC-H	nested/shadow	1.73x	0.88x
Kernel compile	nested	3.49x	0.57x

Figure 14: High level comparison of Palacios workload results showing the best paging approach, speedups between the best and worst approaches, and speedups between the best shadow approach and nested paging

ures 13 (KVM) and 14 (Palacios), it is clear that CR3 write rate and TLB miss rate are highly correlated with the preferred paging approach(es) for each workload.

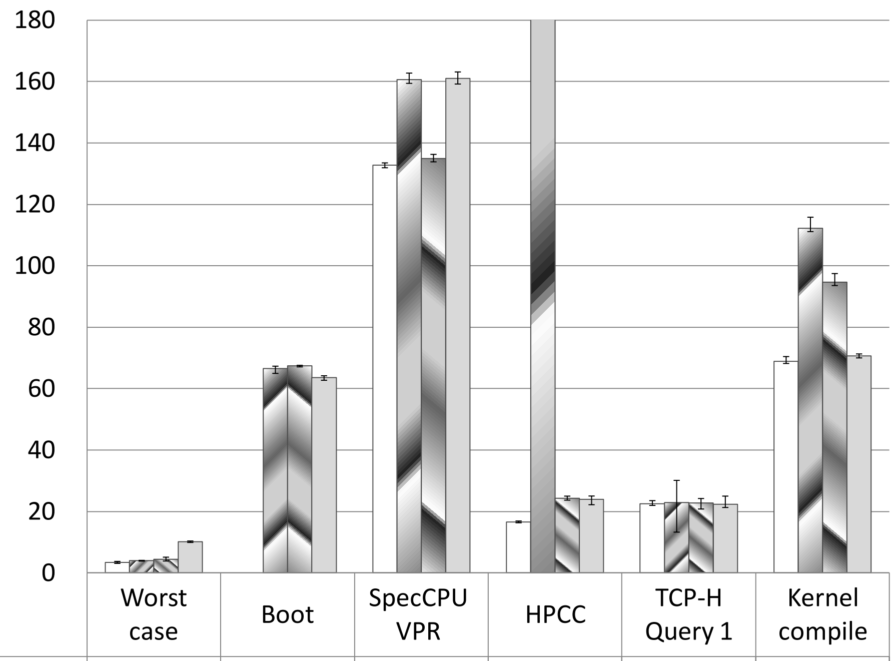
We now consider why this is likely the case. First, nested paging is well documented to perform poorly when the TLB hit rate is low. As we mentioned earlier this is due to the nested page table walks requiring n^2 as opposed n memory accesses to walk the page table hierarchy. This would imply that workloads such as WorstCase and SpecCPU VPR, that experience a high TLB miss rate, would prefer shadow approaches over nested paging. However, workloads such as Boot and TPC-H which experience low TLB miss rates would perform better using nested paging.

Second, all shadow paging approaches are extremely sensitive to shadow page table flushes (guest CR3 writes), because in the worst case it leads to a very large number of exits that are needed to populate the shadow page tables. It is this behavior that caching and prefetching are designed to mitigate. However workloads that do not often perform CR3 writes do not suffer the same penalty, and will actually experience less virtualization overhead using a naive shadow approach than any of the other approaches. This would imply that shadow paging approaches without caching or prefetching would be the best approach to use for the worst case and SpecCPU VPR workloads. Conversely, HPCC, which has a high rate of CR3 writes, should experience the best performance with nested page tables.

The SpecCPU VPR and Worst case workloads, with have high TLB miss rates and few CR3 writes, perform better using standard shadow paging approaches. Also the HPCC workload, which contains a large number of CR3 writes, clearly performs better using nested paging than the standard shadow paging without caching. Finally TPC-H and Boot also perform better using nested paging due to the low TLB miss rate. However, it should also be noted that in most cases shadow paging with caching is able to provide performance that is comparable to nested paging.

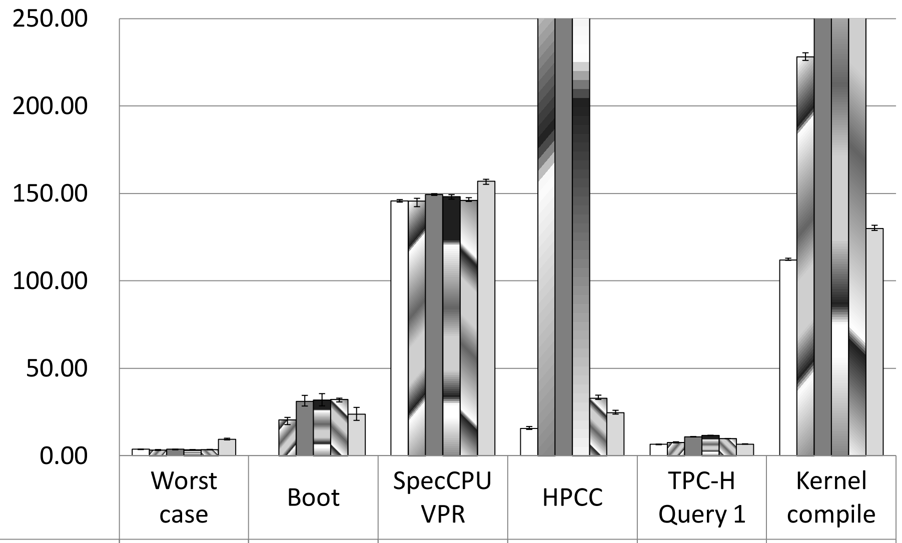
The kernel compilation workload provides an example in which CR3 writes and TLB misses are insufficient to easily predict the best choice of virtual paging approach. Based these metrics we might expect that the paging approaches would perform equally well. However, according to our results, nested paging is clearly preferable on both VMMs, and, on Palacios, shadow paging *without* either caching or prefetching is the best shadow paging approach.

The VMM architecture has a less significant impact than the choice of virtual paging approach. The results show that while the VMM architecture and actual implementation of the virtual page translation mechanisms do have some impact on workload performance, there is still a large correlation between the different architectures. For example, for the HPCC and SpecCPU VPR workloads both KVM and Palacios exhibit comparable relative performance for the same set of ap-



	Worst case	Boot	SpecCPU VPR	HPCC	TCP-H Query 1	Kernel compile
□ Native	3.44		132.71	16.63	22.46	68.81
▨ Shadow w/o Caching	3.97	66.53	160.63	418.10	22.88	112.21
▧ Shadow w/ Caching	4.40	67.41	134.93	24.28	22.75	94.66
□ Nested Paging	10.19	63.54	161.04	23.87	22.29	70.72

(a) KVM



	Worst case	Boot	SpecCPU VPR	HPCC	TPC-H Query 1	Kernel compile
□ Native	3.77		145.75	15.69	6.51	112.10
▨ Shadow w/o Caching	3.27	20.54	145.68	798.92	7.60	228.18
■ Shadow (with Synchronization)	3.69	31.07	149.42	1249.60	10.91	453.22
▤ Shadow (Prefetching)	3.43	31.89	148.20	1305.63	11.65	368.65
▧ Shadow w/ Caching	3.53	32.22	146.07	32.91	9.84	419.36
□ Nested Paging	9.34	23.74	156.93	24.73	6.72	130.05

(b) Palacios

Figure 15: Benchmark run-time.

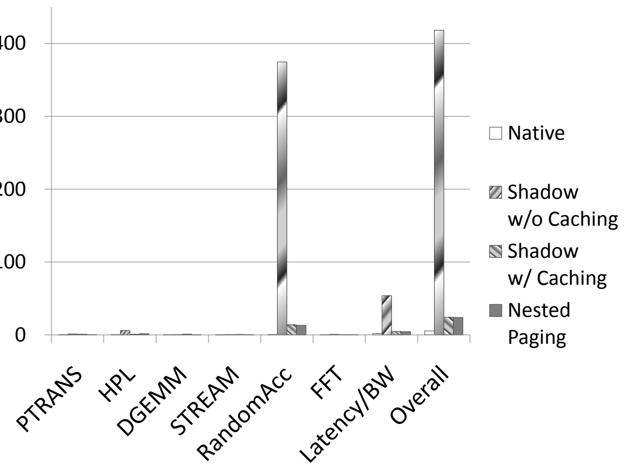


Figure 16: Breakdown of HPCC results (KVM).

proaches. Nested paging is clearly the best approach in both VMMs for the kernel compilation workload. In Palacios, the preferred approaches for both the TPC-H and Boot workloads are a subset of the best approaches for KVM. It should also be noted that no workload demonstrates a preference for completely separate approaches between the VMMs. This indicates that the workload and the paging approaches are the dominating factors for performance.

Breakdown of HPCC workloads. To determine the reason for the large performance disparity between shadow paging without caching and all other approaches for the HPCC workload, we break down the results for the component sub benchmarks (running on KVM) in Figure 16. As can be seen the runtime for the RandomAccess benchmark clearly dominates the total run time. This is to be expected since it is writing to a large number of memory locations located at different regions of the page table hierarchy. Combined with a high CR3 write rate this benchmark not surprisingly achieves terrible performance with shadow paging without caching. However it is important to note that the shadow paging with caching negates much of the performance penalty and results in a runtime similar to nested paging.

6 Conclusions

We have studied the performance of four different approaches to virtualized paging in virtual machine monitors for modern x86 architectures. Our results show that despite the conventional wisdom, there is no single best virtual paging approach, and that the differences in performance between approaches can be significant. The best approach is strongly dependent on the workload. We further found that two workload characteristics, the CR3 write rate, which can be readily measured in a VMM, and the TLB miss rate, which can be readily measured with a architectural performance counter, are strongly predictive of the best virtual paging approach.

Based on our results, it appears that a VMM would ideally dynamically select the virtual paging approach based on these easily measurable characteristics of the workload and its microarchitectural interactions. This could be readily done in our Palacios VMM. Palacios, and we expect most other VMMs, must maintain a representation of the guest physical memory map that is independent of the active paging approach. The necessary interactions with the map are identical for both shadow and nested paging approaches, which means the paging approach could be switched with little overhead. In this paper, we have shown that the prospects for such adaptive paging appear quite good, and have identified the inputs to an adaptation control algorithm. We are now constructing such a system.

References

[1] AMD-V Nested Paging. Tech. rep., AMD Inc., 2008.

[2] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 2006).

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[4] BHARGAVA, R., SEREBRIN, B., SPANINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008).

[5] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide Part 2*, November 2006.

[6] INTEL CORPORATION. *TLBs, Paging-Structure Caches, and Their Invalidation (revision 003)*, December 2008.

[7] KARGER, P. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)* (June 2007).

[8] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Linux Symposium* (2007).

[9] LANGE, J., AND DINDA, P. An introduction to the palacios virtual machine monitor. Tech. rep., Department of Electrical Engineering and Computer Science, Northwestern University, October 2008.

[10] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios: A new open source virtual machine monitor for scalable high performance computing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[11] LUSZCZEK, P., BAILEY, D., DONGARRA, J., KEPNER, J., LUCAS, R., RABENSEIFNER, R., AND TAKAHASHI, D. The HPC Challenge (HPCC) Benchmark Suite. In *SC06 Conference Tutorial* (2006).

[12] MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. Simics: A Full System Simulation Platform. *IEEE Computer 35, Issue 2* (Feb 2002), 50–58.

[13] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10* (2006), 167–177.

[14] SPEC. SPEC CPU 2000. http://www.spec.org/cpu2000.

[15] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC Benchmark H (Decision Support) Standard Specification Revision 2.9.0. Tech. rep., Transaction Processing Performance Council, 2009.

[16] VIRTUTECH, I. Virtutech Simics. http://www.virtutech.com.

[17] XIA, L., LANGE, J., DINDA, P., AND BAE, C. Investigating virtual passthrough i/o on commodity devices. *Operating Systems Review 43*, 3 (July 2009). Initial version appeared at WPIO 2008.

[18] ZHANG, Y., ZHANG, J., LIU, C., AND FRANKE, H. Decision-support workload characteristics on clustered database server from the os perspective. In *Proceedings of the International Conference on Distributed Computing Systems* (2003).