Accelerating Two-Dimensional Page Walks for Virtualized Systems

Ravi Bhargava
Computing Solutions Group
Advanced Micro Devices
Austin, TX
ravi.bhargava@amd.com

Benjamin Serebrin
Computing Solutions Group
Advanced Micro Devices
Sunnyvale, CA
benjamin.serebrin@amd.com

Francesco Spadini
Computing Solutions Group
Advanced Micro Devices
Austin, TX
francesco.spadini@amd.com

Srilatha Manne
Advanced Architecture & Technology Lab
Advanced Micro Devices
Bellevue, WA
srilatha.manne@amd.com

Abstract

Nested paging is a hardware solution for alleviating the software memory management overhead imposed by system virtualization. Nested paging complements existing page walk hardware to form a *two-dimensional (2D) page walk*, which reduces the need for hypervisor intervention in guest page table management. However, the extra dimension also increases the maximum number of architecturally-required page table references.

This paper presents an in-depth examination of the 2D page table walk overhead and options for decreasing it. These options include using the AMD Opteron™ processor's *page walk cache* to exploit the strong reuse of page entry references. For a mix of server and SPEC® benchmarks, the presented results show a 15%-38% improvement in guest performance by extending the existing page walk cache to also store the nested dimension of the 2D page walk. Caching nested page table translations and skipping multiple page entry references produce an additional 3%-7% improvement.

Much of the remaining 2D page walk overhead is due to low-locality nested page entry references, which result in additional memory hierarchy misses. By using large pages, the hypervisor can eliminate many of these long-latency accesses and further improve the guest performance by 3%-22%.

Categories and Subject Descriptors C.0 [*General*]: Modeling of computer architecture; C.4 [*Performance of Systems*]: Design studies; D.4.2 [*Operating Systems*]: Virtual Memory

General Terms Performance, Design, Measurement, Experimentation

Keywords Virtualization, TLB, Memory Management, Nested Paging, Page Walk Caching, Hypervisor, Virtual Machine Monitor, AMD

1. Introduction

Virtualization allows multiple operating systems to run simultaneously on one physical system. These operating systems run as *guests* on the virtualized system and have little or no knowledge that they no longer control the physical system resources. The *hypervisor* is the underlying software that inserts abstractions into a virtualized system: an operating system (OS) becomes a *guest OS*, physical addresses become *guest physical addresses*, and, in general, system elements that the OS presumed were real or physical are converted into virtualized resources under control or manipulation of the hypervisor [14, 16].

Ideally, a virtualized guest system will have comparable performance to an equivalent native, non-virtualized system. This can indeed be the case for compute-intensive applications. For example, the performance overhead for a virtualized system running SPECint®2000 benchmarks can be less than 5% because the hypervisor is infrequently invoked [2]. However, as the number of operations requiring hypervisor intervention increases, performance can degrade substantially. While tolerable in many server consolidation environments, these longer run times are unsatisfactory for performance-sensitive applications.

Operations intercepted by the hypervisor in a virtualized system could consume thousands of cycles of overhead to trap the condition, exit the guest, emulate the operation in the hypervisor, and return to the guest. These costs lead Adams and Agesen to state that "reducing the frequency of exits is the most important optimization for classical [hypervisors]" [2]. More specifically, one of the primary sources of virtualization exits is software memory translation management, which is required to maintain the guest page tables.

AMD has implemented *nested paging* to greatly reduce the overhead of hypervisor intervention in memory management [4]. Under nested paging, the guest controls its unmodified page tables. However, what the guest considers to be real, or *system*, physical addresses are in fact virtualized by the hypervisor. Each guest physical address in the guest page table is looked up in the *nested page tables* by hardware to obtain the system physical address. The end result is a *two-dimensional (2D) page walk* that translates the guest virtual address directly to the system physical address.

Although nested paging removes the overhead of hypervisor intervention, it increases the maximum number of page entry references architecturally required to generate a system physical address. If a guest page walk has n levels and a nested page walk has m levels, a 2D walk requires $nm + n + m$ page entry references. For example, a 2D page walk with four-level guest paging and four-level nested paging has six times more page entry references than a four-level native page walk. Therefore, the overall performance of a virtualized system is improved by nested paging when the eliminated hypervisor memory management overhead is greater than the new 2D page walk overhead.

Translation look-aside buffers (TLBs) can limit the nested paging overhead by caching the full 2D translation and reducing the frequency of page walks. For applications with a high TLB hit ratio, the additional 2D latency will have a negligible impact. However,

ASPLOS'08 March 1–5, 2008, Seattle, Washington, USA

workloads such as databases and web servers, which incur a high TLB miss rate due to their large working sets and frequent TLB flushes, are sensitive to the latency of a 2D page walk.

Contributions. This work discusses the AMD Opteron *page walk cache* (PWC) for the first time. The PWC is designed to reduce the latency of native page entry references by storing page entries in a small, fast cache to avoid memory hierarchy accesses. The 2D page entries are then studied for their cacheability and the highest levels of the nested and guest dimensions are shown to have the most potential for caching.

This paper goes on to evaluate extending the PWC beyond the native implementation to include the nested dimension of 2D page walks. This strategy improves the performance of the virtualized guest workload by 15%-38%. Combining a *Nested TLB* with the PWC further improves guest performance up to 7% by leveraging redundancy and spatial locality to eliminate nested page walks.

These techniques recover more than half of the guest's unvirtualized native performance, but the remaining 2D page walk overhead proves difficult to address with a PWC. Further investigations reveal that 2D page entry references that miss in the PWC often miss in the memory hierarchy as well. However, if the hypervisor uses a larger nested page size, many aspects of the 2D page walk overhead decrease and overall guest performance can improve by an additional 3%-22% depending on the characteristics of the guest workload.

2. Background

This section provides an overview of the native x86 page translation mechanism and a explanation of memory management for virtualized systems, including a detailed description of nested paging. The impact of page size on native and nested page walks is also discussed.

2.1 x86 Native Page Translation

Virtual memory is an abstraction of real system physical addresses, and is typically set up by system software to provide separate address spaces for each process or application [4]. The x86 page translation mechanism uses hierarchical address-translation tables referred to as *page tables* to translate from process-specific *virtual addresses* (VA) to system-specific *physical addresses* (PA). The x86 page tables are hardware-walked, and their layout is specified by the x86 architecture. There are multiple modes of x86 paging, varying primarily in the number of levels and in the ranges of VA and PA that can be mapped.

The page table namespace can become overloaded. The overall hierarchy is sometimes referred to as the "page table," and each level of the tree is a table with a particular name, with the bottommost level of the tree also called the "page table." Figure 1(a) shows the AMD64 *long mode* page translation hierarchy for a 4KB memory page. For brevity and clarity, we will refer to the levels of the page table as L_n, where $1 \leq n \leq 4$ and 1 is the bottommost level. Each level of the table is 4KB in size and is 4KB-aligned. To avoid conflicts with x86's "page table entry" (L_1 in our terminology), we generically refer to a table entry at any level as a *page entry*.

A page walk is an iterative process where a physical memory address is used as the base of an L_n table and nine bits of the VA per iteration are used as an index into that table to retrieve the base address of the L_{n-1} table, until level 1 is reached. The address of the page table base is stored in the architectural CR3 register. Each successive level of the walk maps a smaller VA range: L_4 entries map 512GB, L_3 maps 1GB, L_2 maps 2MB, and L_1 maps 4KB. The L_1 table entry provides the physical address of a 4KB physical page, which is combined with the lowest 12 bits of the virtual address to generate the final physical address of the referenced data.

2.2 Memory Management for Virtualization

Without hardware support, virtualizing the virtual memory system is complex and one of the primary sources of hypervisor overhead. The hypervisor must maintain a separate mapping of guest physical addresses to system physical addresses and use the existing native hardware page walk mechanism for translations from guest virtual address to system physical address. To accomplish this translation, the hypervisor can create a *shadow page table* that maps the same domain of guest virtual address into output ranges of system physical memory [2, 8]. The hypervisor must intervene in every attempt by the guest to install or update a page table.

To avoid the software overhead of shadow paging, hardware mechanisms have been proposed to avoid the intervention of the hypervisor for memory management [4, 7, 8]. One such technique is *nested paging*, in which the guest page table converts guest virtual address to guest physical address, while a new table, the *nested page table*, is introduced to map guest physical address to system physical address. The guest remains in control over its own page table without hypervisor intercepts. Paging control bits and CR3 are duplicated to allow the nested paging table base and mode to be independent from the guest. When an address translation is required, the 2D page walk hardware traverses the guest page table to map guest virtual address to guest physical address, with each guest physical address requiring a nested page table walk to obtain the system physical address.

Figure 1(b) shows the steps required for a 2D walk. The numbers within each circle or square in Figure 1(b) show the ordering of memory references that take place during an end-to-end 2D page walk. The final "SPA" indicates a system memory reference to the referenced datum once the translation has been created. The boxes represent the stages of the guest page walk, and the circles represent the stages of the nested page walk. Each circle or square contains a label showing the level of the page walk, gL_4 to gL_1 for guest page table walk and nL_4 to nL_1 for nested page table walk. Guest physical addresss are indicated by dotted lines.

When referring to a specific access, the notation $\{\mathsf{column,row}\}$ is used. For example, $\{\mathsf{nL}_1,\mathsf{gPA}\}$ refers to step 24 in the 2D page walk. A nested page walk is required to map each guest physical address reference, including those generated during a guest page table access. A single four-level guest table walk invokes the nested page walker five times, once for each guest page entry (gL_4 to gL_1) and once for the final translation of the guest physical address of the datum itself (gPA).

2.3 Two-Dimensional Page Table Walk

Consider a standard x86 page table walk where a physical address is generated at each stage of the walk. In the first step, CR3 and VA[47:39] combine to produce the required physical address in the L_4 table, and the memory read for the L_4 entry comes directly from memory and provides the base of the L_3 table, as in Figure 1(a). In nested paging, each gL_n entry cannot be read directly using a guest physical address; a nested page table walk must translate the guest physical address before the gL_n entry can be read. The guest physical address for gL_4 serves as an input to a recursive call to the page table walker, this time with nCR3 as the base of the page table. The page table walker reads the four $\{\mathsf{nL}_n,\mathsf{gL}_4\}$ entries (steps 1-4 in Figure 1(b)) to translate the guest physical address into a system physical address that can be used to read the desired gL_4 entry (step 5).

The walk proceeds to the next level of the guest page table, which corresponds to the second row in Figure 1(b), and again reads four $\{\mathsf{nL}_n,\mathsf{gL}_3\}$ entries to find the required system physical

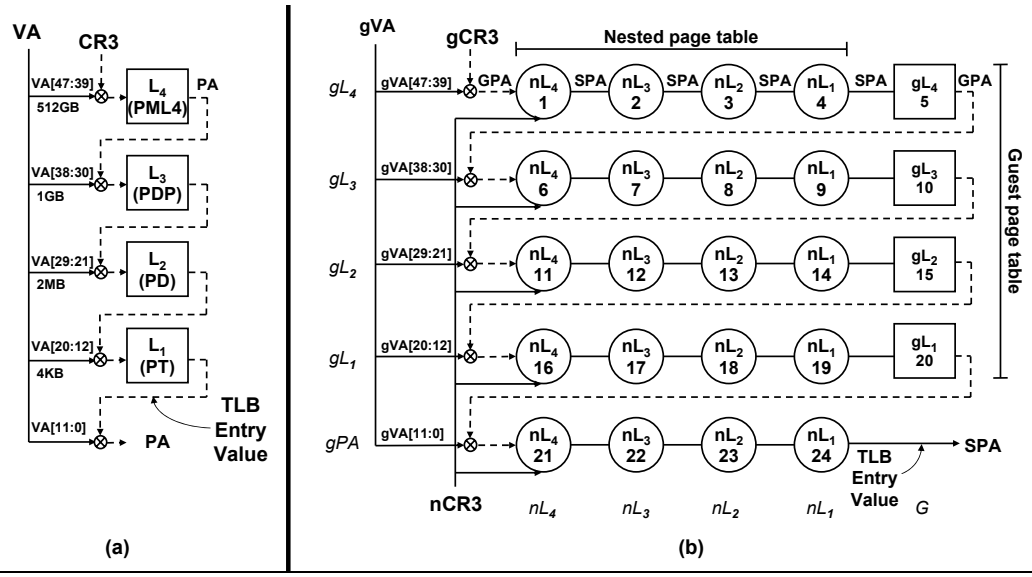


Figure 1. (a) Standard x86 page walk. (b) Two-dimensional page walk. Italics indicate column and row names; notations such as {nL1,gPA} and {G,gL1} indicate entries in the indicated columns and rows.

address. This portion of the walk repeats for gL2 and gL1. The gL1 entry at step 20 determines the guest physical address of the base of the guest data page.

At this point, the guest page table has been traversed, but one final nested page walk (steps 21-24) is required to translate the guest physical address of the datum to a usable system physical address.

2.4 Large Page Size

While the diagrams in this paper show four levels of long mode translation, some workloads have accesses which use only a subset of them. The most important such case is large page support.[1] Large pages provide several advantages in both the native and nested paging scenarios, including memory savings, a reduction in TLB pressure, and shorter page walks.

With 4KB pages, an OS must use an entire L1 table, which occupies 4KB of memory, to map a contiguous 2MB region of virtual memory. If the OS can place all 512 4KB pages of that 2MB region into one contiguous, aligned 2MB block of physical memory, then the OS can substitute a single large page mapping and thus save the 4KB of memory used by the L1 table.

In addition to the memory savings, large pages can reduce TLB pressure. Each large page table entry can be stored in a single TLB entry, while the corresponding regular page entries require 512 4KB TLB entries to map the same 2MB range of virtual addresses. Large page use allows the page walk hardware to skip L1 entirely and use the L2 page entry directly to map a 2MB page, reducing page walk latency due to the number of page entry references. A large page entry encountered at L2 causes an early exit from the standard walk shown in Figure 1(a) and a bypass from {G,gL2} to step 21 in Figure 1(b).

In a nested paging environment, large pages can potentially provide the same benefits in both dimensions of the 2D walk. However, most large page benefits are neutralized if a guest uses a large page to map a block of memory that the nested page table maps with smaller pages. For correctness, the TLB must consider the page size for a given translation to be the smaller of the nested and guest page sizes, referred to as *splintering* [4]. This has important performance implications (discussed further in Section 6.5), as a splintered 2MB page in the guest could require as many as 512 4KB TLB entries.

[1] While AMD64 now adds support for a 1GB page size, this paper uses *large page* interchangeably with *2MB page*.

3. Page Walk Characterization

This section discusses the performance cost of page walks and shows that guest and nested page entries exhibit both a high degree of reuse and a reasonable amount of spatial locality, making them good candidates for caching.

3.1 Page Walk Cost

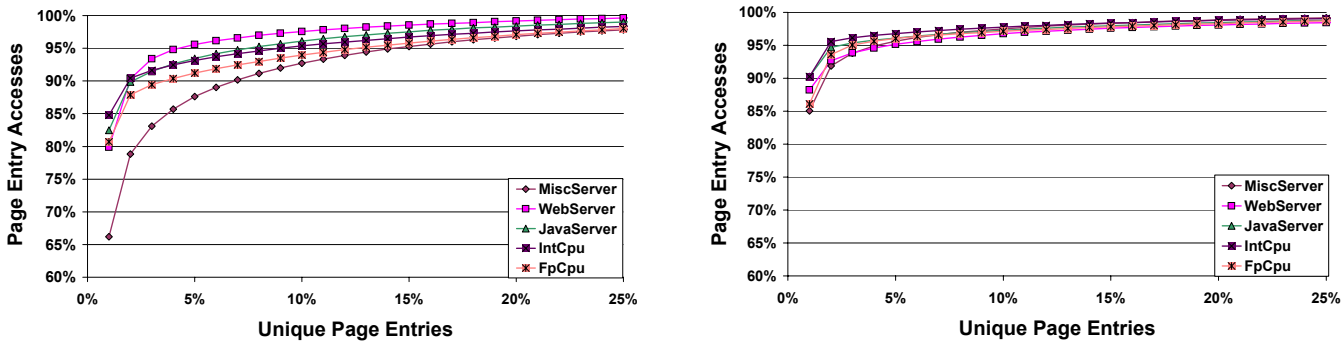
Translation requests that miss in the TLB can degrade performance. Thus, understanding the characteristics of how the TLB behaves in virtualization workloads is key to improving paging performance. Table 1 provides some basic information about TLB behavior. The simulation parameters, methodology, and benchmarks used to produce this data are discussed in detail in Section 5.

Table 1. TLB miss frequency, latency, and performance impact

Instruction and Data Translations				
	TLB Misses (Per 100K Inst.)	Walk Latency 2D/Native	Perfect TLB Opportunity	
			Native	2D
MiscServer	294.3	4.01X	14.0%	75.7%
WebServer	129.0	3.90X	4.7%	44.4%
JavaServer	257.0	3.91X	13.5%	89.0%
IntCpu	70.4	4.57X	11.4%	48.6%
FpCpu	18.2	4.43X	5.7%	27.5%

These statistics were gathered on a model with no specialized page caching hardware other than standard TLBs. *Native* refers to unvirtualized execution. The geometric mean is used within the benchmark suites.

The *TLB Misses* column shows the average number of TLB accesses that result in a page walk per 100,000 retired instructions in each suite. This value applies to both native and virtualized guest execution. The *Walk Latency* column shows the relative slowdown of a 2D page walk with no page walk caching as compared to a native table walk with no page walk caching. The slowdowns



(a) Guest page tables

(b) Nested page tables

Figure 2. Page entry access coverage by a given percentage of unique page entries

are significant, with nested walks being on average 3.9-4.6 times slower than their native counterparts. While a page table walk does not automatically stall progress in an out-of-order machine, it will often stall multiple operations to the same page in memory, and can very quickly stall the entire machine while the TLB miss is resolved.

The last piece of data in Table 1 is the performance improvement that could be theoretically achieved with a perfect TLB, which eliminates cold misses as well as conflict and capacity misses. While native performance can improve 5%-14%, nested paging shows much larger potential gains in guest performance – up to 89% on *JavaServer* and 75% on *MiscServer*. Even CPU-intensive suites, which are generally less sensitive to TLB behavior, show guest improvements of 27%-48%.

3.2 Page Entry Reuse and Locality

Page Entry Reuse. Page entries are cached most effectively when most of the accesses are confined to a small working set of unique entries. While the complete working set of page entries is large for the workloads investigated, they show considerable reuse.

Figure 2 shows the degree of reuse for the guest and nested page entries, where reuse is measured as the percentage of unique page entries needed to account for a given percentage of all page entry accesses. The larger workloads such as *MiscServer* and *WebServer* require more guest pages to cover a given percentage of their page entry accesses than smaller workloads such as *IntCpu* and *FpCpu*. However, even in *MiscServer*, which has the least reuse, less than 10% of guest page entries would need to be cached to cover 90% of accesses.

Nested page tables have much higher reuse than guest page tables, in part due to the inherent redundancy of the nested page walk. This is critical for performance since there are many more nested accesses than guest accesses in a 2D page walk. As seen in Figure 1(b), each level of the nested page table hierarchy must be accessed for each guest level. Since typical guests require a very limited number of nL_4 and nL_3 entries, in many cases the same nested page entries are accessed multiple times in a 2D page walk.

While the large overall page entry working set of some workloads makes caching difficult, the working set is not distributed equally across the 24 references in the 2D page walk. Figure 3 shows the percentage of all unique page entries encountered at each step of the walk for *MiscServer*. Note that this is a measure of the unique page entries seen rather than total accesses. One might expect to see an equivalent number of $\{nL_1,gPA\}$ and $\{G,gL_1\}$ entries in Figure 3, since they are both mappings of the guest data into their respective address spaces. The difference arises when the guest uses large pages, which remove $\{G,gL_1\}$, while the hypervisor configuration for this section always uses full four-level nested paging.

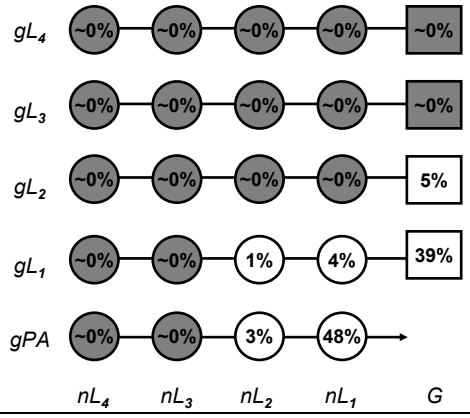


Figure 3. Percentage of all unique page entries for each 2D page walk reference in *MiscServer*

Figure 3 clearly demonstrates that six of the references account for nearly all of the unique page entries. The greyed references accounted for significantly less than 1% of the unique page entries. This has two important implications for caching: all the greyed references should be easily cacheable, since very few entries are needed to capture their working sets; and, the $\{nL_1,gPA\}$ and $\{G,gL_1\}$ references together account for nearly 90% of the page entry working set and may prove difficult to cache effectively.

Page Entry Spatial Locality. Many common microarchitectural techniques can take advantage of spatial locality. The spatial locality of page entries is heavily correlated to that of the guest's data stream, since striding through data regions larger than 4KB will generally involve striding through $\{nL_1,gPA\}$ page entries. The same rationale also applies to $\{G,gL_1\}$, since it only differs from $\{nL_1,gPA\}$ in that it maps the guest data stream to the guest physical address instead of the system physical address. Exploiting the spatial locality of these steps in the 2D page walk is particularly important since Figure 3 shows they are the least reused.

Figure 4 shows the distribution of unique $\{nL_1,gPA\}$ page entries accessed per cache line from the set of lines that service at least one $\{nL_1,gPA\}$ page entry access. Since L_1 page entries are eight bytes each in long mode, the maximum number of page entries per 64-byte cache line is eight. The individual workloads have very different spatial locality profiles. For *IntCpu* and *FpCpu*, more than

70% of lines have at least two {nL$_1$,gPA} page entries, with more than 30% having the full eight. On the other end of the spectrum, the larger workloads have few full lines. Even for *MiscServer*, however, more than 30% of lines have at least two {nL$_1$,gPA} page entries. While not shown here, {G,gL$_1$} entries exhibit a similar distribution.

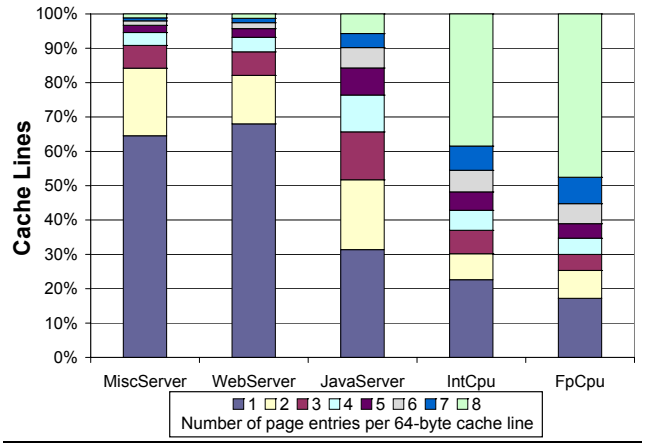


Figure 4. Spatial locality of {nL$_1$,gPA} page entries

This graph is based only on the last page entry reference in a nested page walk. These page entries are eight bytes wide, so a 64-byte cache line can contain up to eight valid page entries.

4. Page Walk Acceleration

Building on the page entry properties in the previous section, the AMD Opteron's page translation acceleration techniques for native page walks are described and extensions for 2D page walks are discussed.

4.1 AMD Opteron Translation Caching

AMD Opteron processors have facilities to accelerate native address translation and page walks. For instance, like most general-purpose microprocessors, the AMD Opteron TLB stores the full virtual address to physical address memory page translation.

The AMD Opteron processor further benefits from the frequent reuse of page entry references by accelerating native page walks with a *page walk cache* (PWC). The PWC is a small, fast, fully-associative, physically-tagged page entry cache. A PWC hit prevents a page entry reference from accessing the memory hierarchy.

The PWC stores page entries from all page table levels except L$_1$, which is effectively stored in the TLB. All page entries are initially brought into the L2 cache and treated like any other data. Therefore, on a PWC miss, the page entry data may reside in the L2 cache, L3 cache (if present), or main memory. Not caching page entries in the L1 caches is a design decision and not a fundamental requirement for page entry caching.

4.2 Translation Caching for 2D Page Walks

To accommodate nested paging, designers can easily modify the TLB to store the full guest virtual address to system physical address translation, which is the equivalent of the entire walk down to the {nL$_1$,gPA} page entry. The design choices for caching 2D page walk entries, however, are more varied than native and potentially more critical, as indicated by the perfect TLB speedups in Table 1.

Figure 5 illustrates the three 2D page walk cache designs outlined in this section. Four-level paging for guest, nested, and native page tables are assumed for illustrative purposes. Each page entry reference in the figure is represented as a circle or a square, as in Figure 1(b). The following describes the caching schemes in more detail.

One-Dimensional PWC (1D_PWC). This design operates almost identically to the native environment. Only page entry data from the guest dimension are stored in the PWC and the entries are tagged based on the system physical address. As indicated by the unshaded references in Figure 5(a), all nested page entries require a memory hierarchy access and cannot benefit from the lower-latency PWC. Also, the lowest level guest page table entry {G,gL$_1$} is not cached in the PWC, similar to the AMD Opteron processor's decision to exclude L$_1$ in native page walks.

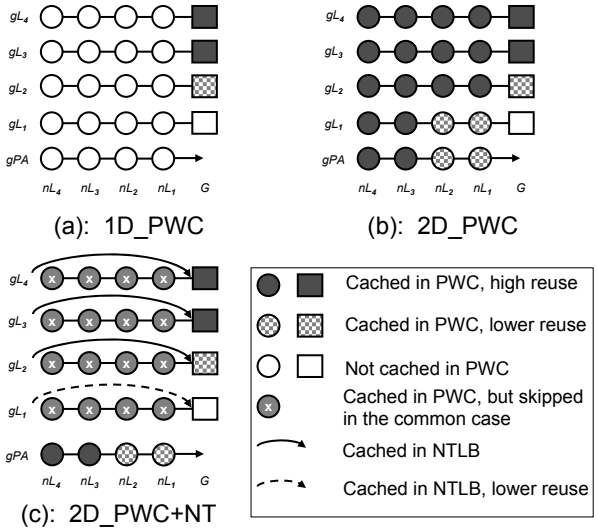


Figure 5. 2D page walk caching designs

Two-Dimensional PWC (2D_PWC). The 2D_PWC design extends 1D_PWC into the nested dimension of the 2D page walk. This caching strategy stores data for all 24 page table references of the 2D page walk, turning the 20 unconditional cache hierarchy accesses of 1D_PWC into 16 likely PWC hits (dark-filled references in Figure 5(b)) and four possible PWC hits (checkered references), where the likelihood of hitting in the PWC is shown in Figure 3. Like 1D_PWC, all page entries are tagged with their system physical address and {G,gL$_1$} is not cached.

Two-Dimensional PWC with Nested Translations (2D_PWC+NT). The 2D_PWC+NT configuration augments 2D_PWC with a dedicated guest physical address to system physical address translation buffer, the *Nested TLB (NTLB)*. The primary goal of the NTLB is to reduce the average number of page entry references that take place during a 2D page walk. A separate structure is used to simplify the design of the PWC and isolate the nested page table translations. The NTLB uses the guest physical address of the guest page entry to cache the corresponding nL$_1$ entry. Caching nL$_1$ page entries is preferable to caching G entries because of the superior reuse characteristics of nL$_1$ demonstrated in Figure 3. In addition, storing nL$_1$ page entries allows one NTLB entry to exploit spatial locality and provide translations for all page entries that reside in the same page of memory.

The common-case 2D page walk with 2D_PWC+NT is depicted in Figure 5(c). The page walk begins by accessing the NTLB with the guest physical address of {G,gL$_4$}. The top solid arrow indicates that this access is expected to hit in the NTLB and produce the

data of $\{nL_1, gL_4\}$, allowing nested references 1-4 (Figure 1(b)) to be skipped (shown as an X in a filled circle). On an NTLB hit, the system physical address of $\{G, gL_4\}$ needed for the PWC access is calculated. Similarly, the system physical addresses for the gL_3 and gL_2 rows are calculated.

The NTLB translation for the gL_1 row has reuse equivalent to $\{G, gL_2\}$ and therefore has an increased chance of missing in the NTLB (shown with a dashed arrow). The NTLB translation from gPA level guest physical address to system physical address occurs at the same frequency as a TLB miss. Therefore, there is no attempt to cache this translation in the NTLB (analogous to the $\{G, gL_1\}$ entry in the PWC). Finally, note that accessing the new NTLB structure imposes latency in the page walk unrelated to the PWC that is not present in the other schemes.

5. Methodology

This section describes the simulation environment for evaluating the described page translation caching strategies. This discussion includes hypervisor-independent 2D page walk modeling, the baseline microarchitecture, and the analyzed workloads. The baseline microarchitecture remains constant throughout the paper and closely resembles that of the AMD Opteron processor.

Hypervisor-independent Simulation. This paper presents simulated 2D page walk caching results in a hypervisor-independent environment. The configuration and implementation variabilities of a specific hypervisor are removed and only the isolated guest performance is reported. This is equivalent to using a theoretical system with absolutely no hypervisor overhead and guest virtualization overhead due only to the 2D page walk.

These simulations use pre-generated nested page tables based on AMD64 long mode with 4KB pages to map guest addresses to the system physical address space. The guest paging mode and page size are properties of the virtualized guest and therefore vary across the analyzed workloads. Note that the nested page size does not need to match the guest page size. Many existing hypervisors use 4KB nested pages for mapping guest physical addresses due to the complexity of eliminating sub-2MB fragmentation. As described in Section 2.2, this forces all TLB entries to be 4KB translations, regardless of the guest's use of large pages. In Section 6.5, the impact of exclusively using a nested page size of 2MB is evaluated.

Baseline Architecture. Page walk caching is evaluated in the context of the latest generation AMD Opteron processor microarchitecture. A cycle-accurate simulator is used to evaluate the page table caching tradeoffs. The simulator has been validated against silicon and the simulated microarchitecture is similar to the native Quad-Core AMD Opteron microprocessor [15, 17]. Table 2 outlines some of the important microarchitectural parameters. Note that TLB flushing due to requirements of the x86 ISA and microarchitecture is modeled. The majority of instructions that flush are writes to control registers that dictate paging properties.

All simulations in this work are performed with a single thread of execution, which is similar to examining a single guest on a single core in a single socket. This may not be appropriate for all virtualization tradeoffs because it ignores multi-processor and multi-guest effects. For this work, we assume that the tradeoffs analyzed for page walk hardware are mostly immune to these effects in both the server consolidation and high-performance virtualization scenarios. In both of these cases, a high-performance hypervisor can use affinity-based scheduling of virtual machines such that the frequency of TLB and PWC flushes easily surpasses that of guest core migrations.

Table 2. Default Microarchitectural Parameters

Memory Hierarchy (AMD Opteron processor)
64 KB L1 instruction cache & 64 KB L1 data cache 512 KB L2 Cache & 2 MB L3 Cache & DDR2-800 memory Average L2 miss latency†: 100 cycles
Data TLB (AMD Opteron processor)
64-entry, fully-associative L1 D-TLB (any page size) 512-entry, 4-way L2 D-TLB (4KB pages) 128-entry, direct-mapped L2 D-TLB (2MB pages)
Instruction TLB (AMD Opteron processor)
32-entry, fully-associative L1 I-TLB (4KB pages) 16-entry, fully-associative L1 I-TLB (2MB pages) 512-entry, 4-way L2 I-TLB (4KB pages)
Page Walk Cache (experimental)
24-entry, fully-associative, LRU PWC 2-cycle PWC access 11-cycle PWC-miss-to-L2 cache hit Flushed on each TLB flush
Nested TLB (experimental)
16-entry, fully-associative, LRU NTLB 2-cycle NTLB access Never flushed during guest execution

† - The *average L2 miss latency* is the average number of cycles across all workloads to receive any type of data from either the L3 cache or memory.

Page Walk Configuration. Table 2 also presents the base configuration of the page walk. The page walk parameters are experimental and are not chosen to be consistent with AMD microarchitectures. In addition to the page walk configurations presented in the results of Section 4, the following configuration is also evaluated in Section 6. The *no page walk caching* configuration has no PWC or NTLB, forcing all 24 nested and guest page entries to be retrieved from the memory hierarchy.

Benchmarks. The simulation inputs are suites of retired instruction traces. Similar to techniques such as SimPoint [6], tracing regions are selected and weighted to represent the entire run of the benchmark. Each simulation consists of a warmup phase and a full simulation phase that produces the results seen in this work. Due to the warmup phase, many of the cold-start microarchitecture effects (including TLB misses) are not present in these runs.

Table 3 lists the workloads presented in the analysis sections. The *MiscServer* suite has database and services benchmarks, including OLTP and SAP. The *JavaServer* and *WebServer* suites have multiple versions and configurations of the corresponding SPEC benchmarks. These three suites represent server workloads which are often virtualized. For example, the VMmark benchmark [18] includes SPECjbb2005, SPECweb2005, and a database component. While all programs are potential virtualization candidates, the SPEC CPU2006 suites are presented primarily to serve as reference points.

Table 3. Benchmark Description

MiscServer	OLTP with three different back-end databases; SAP; stock market analysis; MAPI Messaging Benchmark 3 (MMB3); terminal services
WebServer	SPECweb® 2005, SPECweb® 99, SPECweb® 99 SSL
JavaServer	Three runs of SPECjbb® 2005, each with a different JVM; SPECjbb® 2000
IntCpu	All SPECint® 2006 benchmarks
FpCpu	All SPECfp® 2006 benchmarks

6. Results

This section first presents the performance of the described page walk caching schemes and shows that both of the 2D PWC designs significantly outperform 1D_PWC. The performance advantage of 2D_PWC+NT over 2D_PWC is then analyzed in more detail by examining the PWC access characteristics.

Overall, page walk caching allows the guest to reclaim half of the performance lost to 2D page walk overhead. However, further analysis reveals that lower level nested page entry references are difficult to cache in the PWC and miss frequently in the L2 cache. The results section concludes by exploring hardware and software modifications to further improve 2D page walk performance. Sizing and policy alternatives are studied for the PWC and NTLB structures, and a larger nested page size is tested.

6.1 Guest Performance Improvement

Figure 6 compares the performance improvements offered by the presented page walk caching schemes and a hypothetical hypervisor change to the performance of the unvirtualized guest (i.e., native). The results show that reasonable page walk caching hardware can deliver 51%-78% of the native speedup and, when combined with large nested page sizes, 86%-93% of native performance is possible.

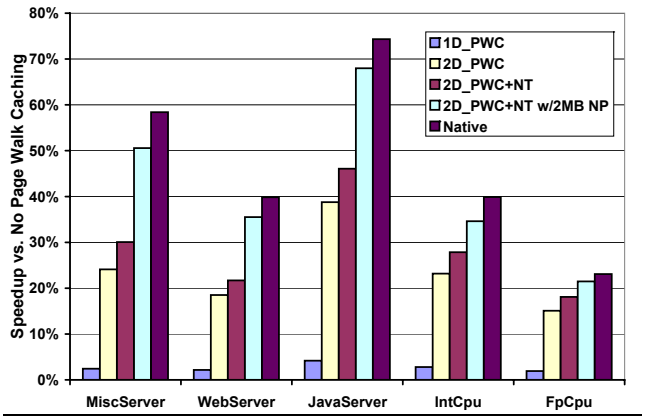


Figure 6. Guest performance improvement

The guest performance is reported as speedup over the baseline microarchitecture with 2D page walks but no page walk caching.

The three hardware-only page walk caching schemes improve performance by turning page entry memory hierarchy references into lower latency PWC accesses and, in the case of 2D_PWC+NT, skipping some page entry references entirely. 1D_PWC, which caches only references from the guest dimension of the 2D page walk, yields little performance relative to the baseline. This is expected, as 21 of the 24 page entry references still require memory hierarchy accesses. For 2D_PWC, which caches all but the $\{G,gL_1\}$ entry in the PWC, the absolute guest speedups range from 18% on *WebServer* to 38% on *JavaServer*. The 2D_PWC+NT configuration, which adds a separate NTLB structure to cache page-level guest physical address to system physical address translations, performs the best. It improves *WebServer*, *JavaServer*, and *MiscServer* guest performance over 2D_PWC by 3.2%, 7.3%, and 5.9% respectively.

The unvirtualized performance (*Native*) is 23%-74% better than no page walk caching and still 5%-28% better than 2D_PWC+NT. Reasons behind this performance difference are presented in Section 6.3. The 2D_PWC+NT w/2MB NP configuration combines the 2D_PWC+NT caching with a 2MB nested page size. This improves on 2D_PWC+NT by up to 21% and is examined further in Section 6.5.

6.2 Page Walk Caching Comparison

The 2D_PWC+NT configuration improves on the 2D_PWC by as much as 7.3% because it can skip nested page walk references. Each NTLB access adds two cycles, but eliminating accesses that would otherwise have hit in the PWC saves two cycles and eliminating memory hierarchy accesses saves many more cycles.

To determine where 2D_PWC+NT differs from 2D_PWC, Figure 7 examines the page walk translation cache accesses and page entry caching miss characteristics in more detail using the *MiscServer* suite. The left side of the figure shows the total PWC and NTLB (where applicable) accesses for 2D_PWC and 2D_PWC+NT, while the right side shows the total page entry references that miss in all caching structures and require memory hierarchy accesses. Each bar is segmented into references based on page entry location in the 2D page walk. The heights of all four stacked bars are relative to the total number of PWC accesses in the 2D_PWC scheme. NTLB accesses are represented by the *NTLB All* segment and are unique to 2D_PWC+NT.

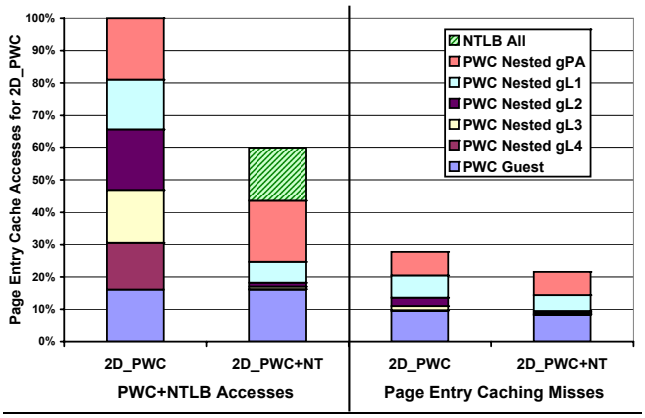


Figure 7. Page entry caching characteristics for *MiscServer*

The stacks are normalized to the number of PWC accesses for 2D_PWC and segmented based on the caching structure and the page walk level of the access. The 2D_PWC+NT scheme adds a second caching structure, the NTLB, to assist the PWC. There are no NTLB-related page entry caching misses because a NTLB miss results directly in PWC accesses instead of memory hierarchy accesses.

The first thing to note is that 2D_PWC+NT has 40% fewer total PWC and NTLB accesses than 2D_PWC and the accesses are not always even across all page entry reference locations. For instance, the *PWC Nested gL1* accesses are reduced by the NTLB in 2D_PWC+NT as expected from Figure 5(c). Even though the $\{nL_1,gL_1\}$ reference has lower reuse, it is valid for multiple $\{G,gL_1\}$ entries and exhibits short-term spatial locality similar to that observed in Figure 4. Therefore, one NTLB entry can eliminate a nested page walk for all page entries on the same page in memory. This same NTLB property allows 2D_PWC+NT to eliminate even more of the *PWC Nested gL2*, *PWC Nested gL3*, and *PWC Nested gL4* accesses.

The *PWC Guest* component of the stack counts guest page entry accesses. Since these G column references are not skipped, the number of guest page entry references does not change between 2D_PWC and 2D_PWC+NT. The *PWC Nested gPA* portion of the stack represents the four nested accesses in the gPA row of Figure 1(b) and is also constant across both caching schemes. As shown in Figure 5(c), the gPA level nested translation is uncacheable in the 2D_PWC+NT scheme, so it will perform as many gPA-level nested accesses as 2D_PWC.

Page entry translation caching misses, which result in memory hierarchy accesses, are presented on the right half of Figure 7. The

2D_PWC+NT design has 23% fewer PWC misses than 2D_PWC compared to a 67% reduction in PWC accesses and 40% reduction in overall accesses. This disparity occurs because 2D_PWC+NT obviates the need to store entries that can be skipped but does not solve the reuse issues inherent in the large working set steps shown in Figure 3. So, although the NTLB eliminates many of the PWC accesses for well-behaved page entry references, it does not eliminate a significant portion of the accesses that have the highest penalty.

Figure 8 continues the in-depth look at *MiscServer* by focusing on the per-reference characteristics of 2D_PWC+NT caching. Each data pair in the table displays the access frequency on the top (in accesses per thousand retired instructions) and the PWC hit percentage on the bottom. Together these metrics portray the utility of caching each page entry reference, where darker shading indicates that a reference is more cacheable and lighter shading indicates that a reference is less cacheable.

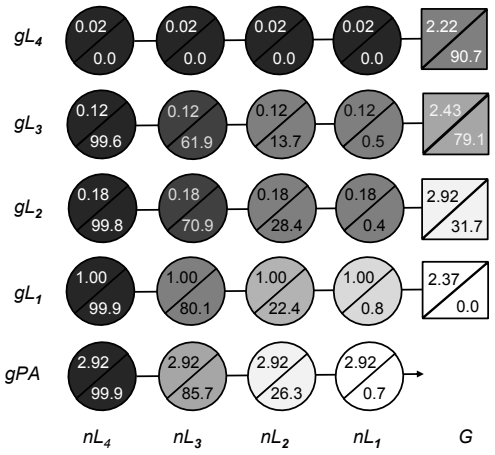


Figure 8. Access rate and hit percentage per page entry reference

The data pairs consist of an activity metric (top) and a success metric (bottom). Activity is measured in PWC accesses per thousand retired instructions. Success is measured in PWC hit percentage.

The page entry access rates show the relative activity of each reference. The access rates for nested page entries within a guest row are equal since the presented schemes do not allow skipping a portion of a nested page walk. However, between guest rows, the magnitude of page entry accesses varies based on the ability of the NTLB to eliminate nested page walks. The guest page entry references (G column) and gPA row nested page entry references occur more often because they are never skipped, as shown earlier in Figure 5(c). The access rates in the G column are not exactly the same due to the guest's choice of paging modes, e.g., non-long mode paging may not require the guest L_3 or L_4 levels while 2MB pages will not require the guest L_1 level.

The PWC hit percentages correlate closely with the working set analysis of Figure 3. The $\{G,gL_1\}$ entries are not written into the PWC, thus the 0% hit percentage. The $\{nL_1,gPA\}$ entry is written into the PWC, but is available only for roughly seven out of 1,000 accesses. These results indicate that this entry should not be cached by default and additional simulations (not shown) report a 0.4% performance increase if $\{nL_1,gPA\}$ is not cached. It is not as clear whether the frequently accessed $\{G,gL_2\}$ and $\{nL_2,gPA\}$ entries should be cached by default since they hit in the PWC for less than half of the accesses. Selective page walk caching schemes are not pursued in this work.

6.3 Page Entry Memory Hierarchy Accesses

Figure 6 shows that 2D_PWC+NT improves guest performance by 18%-45% compared to no page walk caching. This is possible because the page walk caching hardware hits in the PWC or skips the references for page entries with high reuse. However, the page entry references at the lower levels of the guest and nested page tables prove difficult to contain within the PWC. This section reveals that these PWC misses often miss in the L2 cache as well. This has a significant impact on page walk performance and explains much of the remaining 2D page walk overhead because an L2 cache miss is approximately nine times more costly, on average, than an L2 cache hit.

Recall that all page entries are initially brought into the L2 cache and are subject to the policies of the memory hierarchy. Therefore, on a PWC miss, the memory hierarchy is queried with the system physical address for the required page entry. Table 4 presents L2 cache statistics for just these page entry references. The first data column states that L2 accesses incurred during a 2D page walk using the 2D_PWC+NT configuration generate 2.7-5.5 times more L2 misses than the native page walk. This increase is primarily because the native page walk has fewer entries that are difficult to cache (L_1 and sometimes L_2) compared to the 2D page walk ($\{G,gL_1\}$, $\{nL_1,gPA\}$ and sometimes $\{G,gL_2\}$, $\{nL_2,gPA\}$, $\{nL_1,gL_1\}$, and $\{nL_2,gL_1\}$). Guests that use large pages will also see an increase in page entry misses in the L2 cache because the smaller nested page size leads to page splintering and results in more TLB misses (see Section 6.5 for performance with a large nested page size).

Table 4. L2 cache misses for page entry references (2D_PWC+NT)

	Page Entry L2 Cache Misses (2D/Native)	Page Entry L2 Cache Miss %
MiscServer	3.06X	25.07
WebServer	5.52X	19.58
JavaServer	3.13X	24.55
IntCpu	2.74X	14.67
FpCpu	2.95X	16.35

The second data column shows the L2 cache miss percentage when isolating the accesses due only to page entries from the 2D page walk. As an example, references in the *MiscServer* workloads are not available in the L2 cache approximately once out of every four accesses. The miss percentages are relatively high because the PWC and NTLB have filtered the easy-to-cache accesses and the remaining accesses are difficult to cache.

The results from this section imply that the reuse analysis and cacheability conclusions of Section 3.2 apply to both the PWC and to the L2 cache. Increasing the sizes of the PWC and NTLB can further improve performance to an extent, as examined in Section 6.4.

6.4 Hardware: Impact of Increased Resources

This section examines the sensitivity of guest performance to the sizes of the PWC and NTLB. This analysis demonstrates the modest extent to which the increasing page walk caching entries can reduce the 2D page walk overhead.

Figure 9 presents performance improvements for the *MiscServer* suite due to increases in the PWC and NTLB sizes. These improvements are relative to the baseline 24-entry PWC and 16-entry NTLB 2D_PWC+NT configuration. For the five PWC caching configurations on the X-axis, there are three corresponding NTLB sizes. The *8096 w/(G, gL1)* configuration is unique in that it writes the gL_1 guest page entry to the PWC. At 8096 entries, the PWC is large enough to surpass the translation working set of the TLBs.

Under these circumstances, this change in policy improves performance. All PWC and NTLB configurations are fully-associative and accessed with the same latency regardless of size.

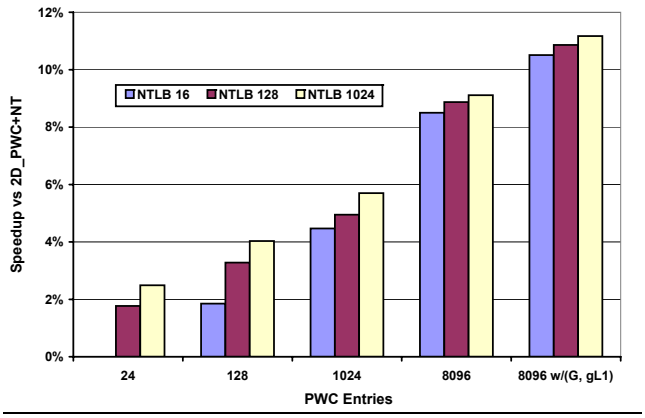


Figure 9. Impact of PWC and NTLB configuration on *MiscServer* guest performance

The performance improvement for three NTLB sizes are presented for each PWC configuration. The *NTLB 16* bar for the 24-entry PWC configuration is 0% since it is the same configuration as the baseline. The *w/(G, gL1)* configuration allows caching of the $\{G,gL_1\}$ entry.

Several important conclusions can be drawn from Figure 9. First, a substantial increase in PWC and NTLB entries brings some extra performance but does not come close to bridging the gap between guest and native performance. Second, while the first 16 NTLB entries resulted in a 6% performance improvement for *MiscServer*, investing in additional NTLB entries has little impact, especially as the number of PWC entries are increased. Finally, for our base configuration, increasing PWC entries is more productive than increasing NTLB entries, but requires substantial hardware for little performance improvement. Note that increasing page walk caching resources for *JavaServer* had very similar results to *MiscServer* while it delivered much lower performance improvements for *WebServer* (no greater than 4% for the same configurations).

These hardware resource results emphasize that the utility of page walk caching is confined to efficiently taking advantage of inter-walk and intra-walk page entry reuse to improve page walk latencies and will not remove the 2D page walk overhead entirely. If additional resources are dedicated to translation speedup, it is preferable to target full translation caching (i.e., the TLBs). For example, increasing the L2 data TLB to 8096 4KB page entries increases overall performance for *MiscServer* by 11.3% (not shown) compared to 8.5% with 8096 PWC entries.

6.5 Software: Impact of Large Nested Pages

Adding large nested page sizes to the 2D_PWC+NT page walk caching brings guest performance within 7%-14% of unvirtualized performance, as shown earlier in Figure 6. Large nested page sizes are beneficial because they remove one nested page entry reference per guest level, including the hard-to-cache nL_1 references. This section compares the default 4KB nested page size to a 2D page walk that uses 2MB nested pages to map all of guest physical memory.

The hypervisor is responsible for generating the nested page tables and therefore determines the nested page size. Since the nested page size in turn dictates the number of nested page levels, this choice has important implications for 2D page walk performance. Section 2.4 explains that the TLB will store a translation at the smaller of the guest and nested page sizes. Therefore, hypervisors that use large nested pages allow guests that use large pages to receive the full benefits of large pages. Even when a large nested page backs a small guest page, the elimination of nL_1 references provides a significant reduction of memory references. System virtualization requires techniques like these, in addition to better TLBs and page entry caching, to significantly reduce the memory management overhead.

Table 5 presents relative performance and page translation activity for the 2MB nested page size configuration compared to the default 4KB-only nested page size configuration. The three data columns are presented as a percent reduction versus the default 4KB nested page configuration.

Table 5. Page walk data for 2D_PWC+NT with 2MB nested page size

	Reduction vs 2D_PWC+NT with 4KB Nested Page Size		
	TLB Misses	PWC Accesses	Page Entry L2 Cache Misses
MiscServer	21%	38.0%	60.0%
WebServer	43%	55.5%	61.0%
JavaServer	32%	44.4%	64.7%

Large pages allow the TLB to cover a larger data region with fewer translations. For guest applications that regularly use 2MB pages, mapping these pages to 4KB nested pages will put more pressure on the TLBs. Here, the use of 2MB nested pages results in 21%-43% fewer TLB misses than in the 4KB nested page size configuration. In addition, if all guest physical addresss are mapped to 2MB pages by the nested page tables, the nL_1 references for the gPA, gL_1, gL_2, gL_3, and gL_4 levels are all eliminated. So, even without guest large pages, 2MB nested pages will improve TLB performance due to page walk access and miss count reductions. The PWC access counts for these server workloads are reduced by 38%-55%. The biggest improvement comes from having fewer L2 cache misses. The ability to eliminate poor-locality references, like $\{nL_1,gL_1\}$ and $\{nL_1,gPA\}$, reduces the number of L2 cache misses by 60%-64%.

7. Related Work

There are a number of alternatives to the x86 hierarchical page walk that, on average, require fewer levels of translation and fewer accesses to memory. For example, PA-RISC's *hashed page table* [10] works well for large address spaces with sparse utilization. In the worst case, the entire link must be traversed to determine the physical address. However, the average traversal time is fairly short because the average chain length is 1.5 entries.

Clustered page tables have been proposed as a way to improve hashed page tables when there are clusters of contiguous pages being allocated [13]. They minimize the size of the page table and/or reduce the size of the hash lists by clustering contiguous pages together into sub-blocks. Minimizing the size of the page table results in better locality for the page table entries in the caches, while reducing the size of the hash table lists means fewer traversals are required to find the correct page.

Some of these hashed paging techniques could have been used to minimize the number of translations in nested paging. However, allowing the nested page table format to differ from the native format would incur undesired complexity.

Other work has focused on caching or eliminating some of the translation steps similar to the work presented here. The Intel application note on memory management [9] discusses paging caches that use various bits of the linear address to index into the cache and skip some of the translation steps. However this document does not provide a detailed analysis of their mechanism. Leidtke proposed *guarded page tables* [12] to eliminate page tables and translation steps when there is only one valid entry in a page table. Similarly, the AMD I/O virtualization architecture (IOMMU) allows the

grouping of virtual address bits to be used for the next step of the translation [3]. This allows the IOMMU to skip page translation steps for sparsely allocated device virtual memory. *Region Lookaside Buffers* are used to skip some translation steps by caching multiple level page tables in a special TLB [5].

Finally, a set of papers and implementations have focused on reducing the impact of TLB page walks by prefetching entries into the TLB. The PA-SEMI PWRficient family prefetches the next sequential page into the TLB on a TLB miss [19]. Also, a doubly-linked list can be used to keep track of the TLB entries for prefetching [1]. The work by Kandiraju examines different methods for predicting the next TLB entry to prefetch based on mechanisms used for cache prefetching [11].

8. Conclusion

Nested paging is a hardware technique to reduce the complexity of software memory management during system virtualization. Nested page tables combine with the guest page tables to map the guest physical address space to the system physical address space, resulting in a two-dimensional (2D) page walk. A hypervisor is no longer required to trap on all guest page table updates and significant virtualization overhead is eliminated. However, nested paging can introduce new overhead for guest applications due to the increase in page entry references. Unvirtualized server and SPEC applications are shown to outperform the virtualized versions with no page walk caching by 20%-70%. This work studies the sources of this overhead, the properties of a 2D page walk, and solutions for minimizing the 2D page walk latency.

This paper first presents the benefits and limitations of page walk caching along with an analysis of page entry reuse and spatial locality. The spatial locality analysis confirms that encountered page entries are frequently co-located at a cache line granularity. The reuse analysis shows that the page entry working set is not distributed equally across all the references in a 2D page walk. While there is heavy page entry reuse at the upper levels of both the nested and guest page walks, the references at the lower levels account for nearly 90% of the page entry working set and will not be cached effectively.

The AMD Opteron page walk cache (PWC) for native page walks and PWC extensions for 2D page walks are described and evaluated using a hypervisor-independent simulation methodology. Extending the PWC and adding a Nested TLB for 2D page walks improves guest performance by as much as 46% and eliminates over half the overhead of 2D page walks compared to no page walk caching. However, page entries at the lower page table levels limit the improvements because of poor cacheability in both the PWC and the memory hierarchy. This paper then demonstrates that hypervisor software can eliminate long-latency page entry references and further improve guest performance as much as 22% by using large nested page sizes.

Acknowledgments

The authors acknowledge the following individuals for their technical contribution to the definition and implementation of nested paging in AMD processors: Kevin McGrath, Mike Clark, Alex Klaiber, Mike Haertel, Doug Hunt, Steve Thompson, Mike Tuuk, and Richard Klass. In addition, the authors thank our colleagues who provided important feedback along the way: Ben Sander, David Christie, Rich Witek, Andy Kegel, Chris Svec, Matt Crum, Kevin Lepak, Leendert van Doorn, Ole Agesen, Alex Garthwaite, and the ASPLOS reviewers.

References

[1] A. Saulsbury et al. Recency based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

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

[3] *AMD I/O Virtualization Technology (IOMMU) Specification*, February 2007.

[4] *AMD Programmer's Manual, Volume 2*, September 2007.

[5] D. Chang et al. Microarchitecture of HAL's memory management unit. In *COMPCON '95: Technologies for the Information Superhighway, Digest of Papers*, pages 272–279, 1995.

[6] E. Perelman et al. Using SimPoint for accurate and efficient simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[7] G. Neiger et al. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–782, 2006.

[8] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, Nov. 1983.

[9] TLBs, paging structure caches, and their invalidation. *Intel Application Note*, 317080-001, April 2007.

[10] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB refill mechanisms, and page table organizations. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[11] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 195–206, 2002.

[12] J. Liedtke. Address space sparsity and fine granularity. *6th ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, pages 78–81, 1994.

[13] M. Talluri et al. A new page table for 64-bit address spaces. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SIGOPS)*, 1996.

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

[15] B. Sander. Processor optimizations for system-level performance. Presentation at Microprocessor Fall Forum, 2006.

[16] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[17] *Software Optimization Guide for AMD Family 10h Processors*, September 2007.

[18] V. Makhija et al. VMmark: A scalable benchmark for virtualized systems. Technical report, VMWare, 2006.

[19] T.-Y. Yeh. Low-power, high-performance architecture of the PWRficient processor family. *Hot Chips 18*, 2006.