Appears in the 39th International Symposium on Computer Architecture (ISCA 2012)

Revisiting Hardware-Assisted Page Walks for Virtualized Systems

Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh
Computer Science Department, KAIST
{jeongseob, swjin, and jhuh}@calab.kaist.ac.kr

Abstract

Recent improvements in architectural supports for virtualization have extended traditional hardware page walkers to traverse nested page tables. However, current two-dimensional (2D) page walkers have been designed under the assumption that the usage patterns of guest and nested page tables are similar. In this paper, we revisit the architectural supports for nested page table walks to incorporate the unique characteristics of memory management by hypervisors. Unlike page tables in native systems, nested page table sizes do not impose significant overheads on the overall memory usage. Based on this observation, we propose to use flat nested page tables to reduce unnecessary memory references for nested walks.

A competing mechanism to HW 2D page walkers is shadow paging, which duplicates guest page tables but provides direct translations from guest virtual to system physical addresses. However, shadow paging has been suffering from the overheads of synchronization between guest and shadow page tables. The second mechanism we propose is a speculative shadow paging mechanism, called speculative inverted shadow paging, which is backed by non-speculative flat nested page tables. The speculative mechanism provides a direct translation with a single memory reference for common cases, and eliminates the page table synchronization overheads. We evaluate the proposed schemes with the real Xen hypervisor running on a full system simulator. The flat page tables improve a state-of-the-art 2D page walker with a page walk cache and nested TLB by 7%. The speculative shadow paging improves the same 2D page walker by 14%.

1 Introduction

As system virtualization has been widely used for public cloud computing as well as corporate server consolidation, recent processor designs have been extending their architectural supports for virtualization [3, 19]. One of such architectural enhancements for virtualization is to support two-dimensional (2D) page table walks, which can traverse both guest and nested page tables with a hardware page table walker. In virtualized systems, a guest virtual address is translated into a guest physical address with a per-process guest page table, and the guest physical address must be translated to a system physical address with a per-VM nested page table. Translation Look-aside Buffers (TLBs) store direct translations from guest virtual to system physical page numbers, but for each TLB miss, both page tables must be traversed.

An alternative way to such hardware 2D page walks for memory virtualization is a software-based approach called shadow paging [4]. The hypervisor maintains a shadow page table for each guest process, and the shadow page table contains a direct mapping from guest virtual pages to system physical pages. Although shadow paging allows TLB misses to be served by walking a single page table, the most significant performance overhead of shadow paging is hypervisor interventions to reflect any changes of a guest page table to the corresponding shadow page table. Hardware 2D page table walkers eliminate the overhead of such costly hypervisor interventions [4].

However, the current hardware supports for nested page table walks extend traditional HW-based multi-level page table walkers. Both of the guest and nested page tables are organized as multi-level page tables to reduce memory overheads. With four-level page tables commonly used for 64-bit address spaces, a 2D page table walk takes six times more references than a native page walk [8]. Such multi-level page tables are designed for memory usage patterns for processes in native OS systems, to reduce memory footprints for page tables. However, multi-level nested page tables may not be able to reduce the memory requirement for nested page tables effectively. The number of virtual machines (VMs) is often very limited with only several or tens of VMs in a virtualized system. Furthermore, each virtual machine uses much of the initially declared guest physical address space, which is much smaller than the virtual address space of a process.

In this paper, we revisit the design space of hardware-assisted nested paging supports, by considering the unique characteristics of memory usages and allocation patterns for

virtual machines. As traditional multi-level page tables do not save memory space significantly for nested page tables in virtualized systems, we propose to use *flat nested page tables*. By eliminating unnecessary levels, memory references for TLB misses can be reduced significantly.

However, flat nested page tables do not reduce the overhead of multiple references to traverse guest page tables. Traditionally, hashed inverted page tables reduce the number of memory references for TLB misses. For nested address translations in virtualized systems, an inverted page table entry can contain a direct mapping from a guest virtual page to a system physical page. However, such an inverted page table for virtualization has the same performance overhead as shadow paging. Whenever a guest page table entry changes, the hypervisor must intervene and update the inverted page table. The second mechanism we propose is a speculative mechanism, called *speculative inverted shadow page table (SpecISP)* to eliminate the overhead of synchronizing the inverted page table with guest page tables. With the speculative mechanism, each TLB miss looks up the speculative inverted shadow page table, which should take only a single memory reference for common cases. At the same time, the flat page table is accessed to retrieve the correct mapping. A TLB miss is served speculatively, and must be verified by the correct flat page table.

We evaluate the proposed reorganization of nested page tables, and speculative shadow paging with a real hypervisor running on a full-system simulator. Flat page tables improve a state-of-the-art HW 2D page walker with a page walk cache and nested TLB by 7% on average for a set of applications. Speculative inverted shadow paging can potentially improve the same 2D page walker by 14%.

The rest of the paper is organized as follows. Section 2 describes hardware-assisted memory virtualization, and discusses the prior work to improve the performance of address translation for native and virtualized systems. Section 3 discusses the unique characteristics of VM memory allocation policies, and presents the organization of flat nested page tables. Section 4 presents the speculative mechanism to support the inverted shadow page table without synchronizations with guest page tables. Section 5 presents the experimental results, and Section 6 concludes the paper.

2 Background

2.1 Address Translation for VMs

To provide each VM with an isolated guest-physical address space, virtualization requires two-level address translations. For each process running in a VM, a guest virtual address (gVA) is translated into a guest physical address (gPA) by a per-process guest page table, which is maintained by the guest OS in the VM. As multiple VMs share the system memory, the guest physical address should be translated into a system physical address (sPA) with a per-VM nested page table. Figure 1 depicts the address translation procedure for virtualized systems. Translation look-aside buffers (TLBs) keep direct translations from guest-virtual addresses to system-physical addresses. For a TLB miss, either a SW or HW page table walker must traverse two page tables to fetch the final translation to the system physical page number. For HW-based page walkers used in many commercial processors including popular x86 architectures, there are two common ways to translate a guest-virtual page number to a system-physical page number for TLB misses without any modification in guest OSes.

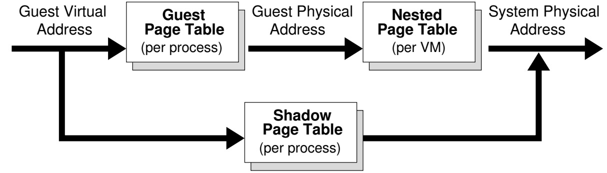

Figure 1: Address translation in virtualized systems

In a software-oriented method, for each process in guest virtual machines, the hypervisor maintains a shadow page table, which has a direct mapping between guest virtual to system physical addresses as shown in Figure 1. Shadow paging can be used with traditional page table walkers which can traverse only a guest page table for a TLB miss, and it does not require any extra architectural support for virtualization. Each core has a register pointing the top-level page table (`CR3` in the x86 architecture). Whenever a guest OS changes the page table register to a guest page table, the hypervisor must intercept the change, and update the register to the corresponding shadow page table. Per-process page tables must be duplicated in the guest OS and hypervisor, and furthermore, any change of a page table by the guest OS requires a costly hypervisor intervention to update the corresponding shadow page table. For applications with frequent memory mapping changes, such synchronization overheads between guest and shadow page tables cause significant performance degradations [22]. In Section 4.1, the overheads of shadow paging will be discussed in more details. Shadow paging can potentially reduce memory references for TLB misses, since it needs to walk a single page table for a TLB miss. However, such synchronization overheads often exceed the benefits of reduced memory references.

2.2 Hardware-Assisted Virtualization

Recent supports for two-dimensional walks eliminate the need for shadow paging. Each core has both the page table pointer to the current guest page table (gCR3 in x86), and the page table pointer to the nested page table (nCR3 in x86). For a TLB miss, the HW walker traverses both of the tables to get the final translation between guest-virtual

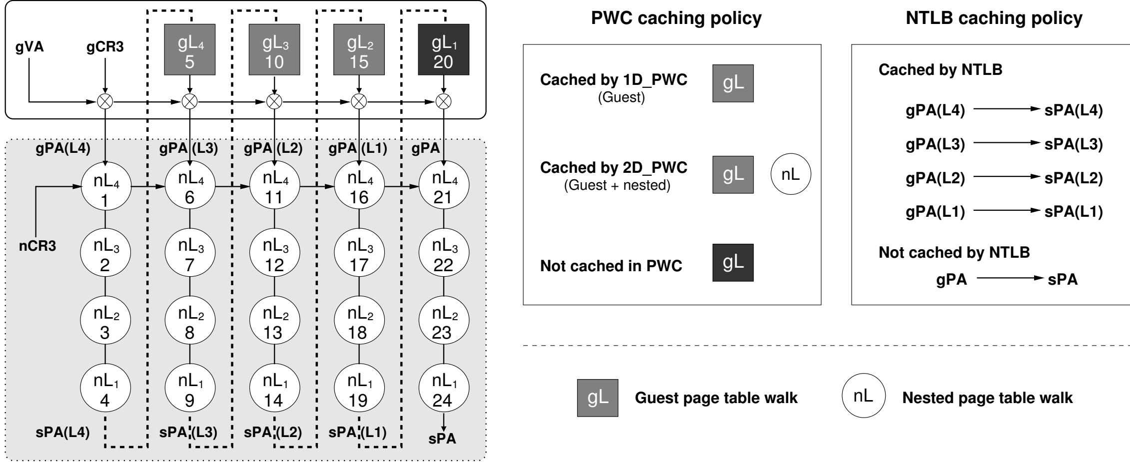

Figure 2: Two-dimensional (2D) page walks for virtualization in x86 architectures

and system-physical page numbers. Such page table walkers are designed to support the current multi-level page tables, whose formats are specified and fixed in the instruction set architecture. The hardware walker assumes the same organization of page tables for guest and nested page tables, although page granularity can differ modestly. Due to the restricted page table organization for multi-level tables, a TLB miss can cause many references to both page tables.

Figure 2 presents the address translation process for virtualized systems in the 64-bit long mode of the x86 architecture. In the figure, the rectangles are guest page table walks, and the circles are nested page table walks. Native systems require only 4 references to the page table for each TLB miss, but virtualized systems requires 24 references to the two tables by walking the tables in a two-dimensional way. More generally, if guest page tables and nested page tables are m and n levels, the 2D page walks take $mn+m+n$ references [8]. Due to the long latencies for handling TLB misses, the performance of applications with high TLB miss rates may be lower with the nested page table walker than that with shadow paging [21, 22]. On the other hand, the hardware-assisted 2D page walker performs better than shadow paging for applications with frequent updates on guest page tables. Furthermore, the hardware-assisted mechanism does not require extra memory for duplicating page tables for guest processes.

To reduce the overhead of many memory references by 2D page walkers, some architectures use a *page walk cache (PWC)*, which is an extra hardware table to hold intermediate translations [3, 8]. For a TLB miss, instead of accessing the memory, a small hardware PWC is first checked to find the necessary intermediate translations. As upper-level intermediate translations, which cover a large memory space in multi-level paging, exhibit high temporal localities, a small number of entries in PWC may capture the working set of the upper-level translations. In 1D PWC, intermediate translations only for guest page tables are cached in PWC, and in 2D PWC, intermediate translations both for guest and nested page tables can be cached in PWC. Figure 2 shows which translations can be cached in PWC. However, with 2D PWC, the limited PWC capacity must be shared by the intermediate translations for 1D translation from guest virtual to guest physical pages, and the 2D intermediate translations for nested page table walks.

Another hardware support to reduce the overhead of walking 2D page tables is *nested TLBs (NTLBs)* [3]. The nested TLBs hold the most recently used mappings from guest-physical to system-physical pages, as shown in Figure 2. These two techniques, PWC and NTLB, are used to reduce the overhead of multi-level nested page tables. The overhead of multi-level page walks stems from the assumption that nested page tables have the same requirements as the traditional page tables used in native OS systems.

2.3 Related Work

Reducing the overheads of address translation has been a critical aspect of optimizing memory hierarchy in native systems as well as virtualized systems. For native systems, several studies have investigated schemes to reduce TLB misses or reduce the overheads of handling TLB misses. Talluri et al. investigated several existing page table schemes in a 64-bit system and proposed the clustered page table which exploits contiguous memory allocation [18]. Jacob and Mudge evaluated various memory management units including software and hardware-based mechanisms [15]. Barr et al. compared several designs of caching intermediate page table entries in multi-level page tables, evaluating various page walk caching policies [6]. Their another work, SpecTLB, proposed to use speculation for address translation to reduce the page walk overhead [7]. It exploits the spatial locality available in memory allocation policies in commercial operating systems. Speculative

shadow paging proposed in this paper has been inspired by their speculative mechanism for handling TLB misses, although the purpose of using speculation greatly differs from that for SpecTLB.

Virtualization has made efficient address translation critical for the overall system performance, as it requires both guest and nested page table translations. Bhargava et al. discussed page walk overheads in virtualized systems [8]. They showed that a small number of entries for nested page tables are frequently reused due to spatial and temporal localities. To take advantage of this characteristic, page walk caches used in non-virtualized systems are extended to cover nested page walks. In addition, they also evaluated nested TLBs to further reduce memory references for nested translations. Wang et al. proposed a selective address translation mechanism between software-based shadow paging and hardware 2D page walk [22]. Considering the tradeoffs between shadow paging and HW 2D page walk, their study proposed to use both software and hardware schemes selectively, depending on application behaviors.

There are several studies to reduce TLB misses in chip multiprocessors. Bhattacharjee et al. investigated the TLB behaviors of parallel workloads [10]. Their study shows that many TLB misses for each core are redundant and predictable, if the cores run a shared-memory parallel application. To exploit these characteristics, they proposed prefetch mechanisms, which are based on the predicable behaviors [11]. In addition, recent studies proposed the shared last-level TLBs which are analogous to shared last-level caches in multi-cores [9, 17]. Shared TLBs maximize the caching efficiency of TLBs by sharing limited physical resources among cores.

3 Flattening Nested Page Tables

In this section, we discuss the differences of memory managements for virtual machines and for processes in native systems. Based on the observations, we propose to reorganize nested page tables to flat tables by exploiting the unique characteristics of memory management for virtual machines.

3.1 Memory Overheads of Nested Page Tables

Current nested page organizations assume that virtual machines are analogous to processes in native systems. However, there are several differences between two entities, which can affect the requirements for nested page tables significantly. In native systems, there are many processes, which may not always be running, but consume the memory to maintain the state. For example, there are many daemon processes, which wait for certain events. Many of such processes use only a tiny amount of the actual system memory. Figure 3 shows the distribution of memory usage by processes in a Linux system, after booting the system. There are 106 processes, and most of the processes use less than 1MB of memory. Since a page table must be created for each process even if the process uses a small amount of memory, the total memory used for all page tables must be reduced as much as possible.

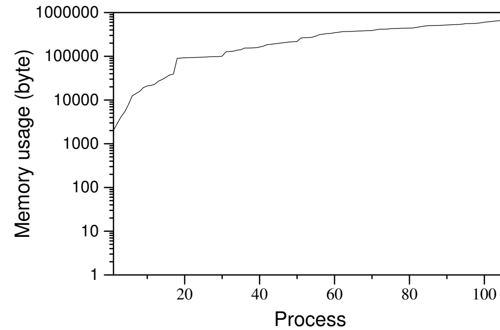

Figure 3: Cumulative distributions of memory usage after booting a Linux system

Furthermore, page tables for processes map the virtual address space to the physical address space. The virtual address space of a process is specified and fixed in the ISA, and it is much larger than the total system memory. For example, in the 64-bit x86 architecture, the long mode assumes a 48-bit virtual address space, which can be as large as 256 TB of memory. Page tables must be able to cover the entire virtual address space. Due to the aforementioned requirements for page tables of processes, page tables are organized into multi-level tables. In the 64-bit x86 architecture, a page table is organized as a four-level table, which can save the memory for each page table significantly.

However, virtual machines exhibit quite different behaviors than processes. Firstly, the number of virtual machines in a physical system is limited. For VMs running compute-intensive workloads, the number of virtual CPUs for all guest VMs often does not exceed the number of physical cores. Even for I/O intensive servers with relatively low CPU utilization, the number of VMs in a system is orders of magnitude smaller than the number of processes in native systems.

Secondly, nested page tables, which map the guest physical to system physical memory space, have much smaller mapping ranges than those of the page tables for processes. A per-process page table must cover the entire virtual address space, but a nested page table needs to cover only the guest physical memory space. When a VM is created, the guest physical memory is specified. However, the total guest physical memory from all VMs often does not exceed the available system memory. Even if the VM memory is over-committed, or the total VM memory exceeds the system memory, the overcommitment ratio is commonly two

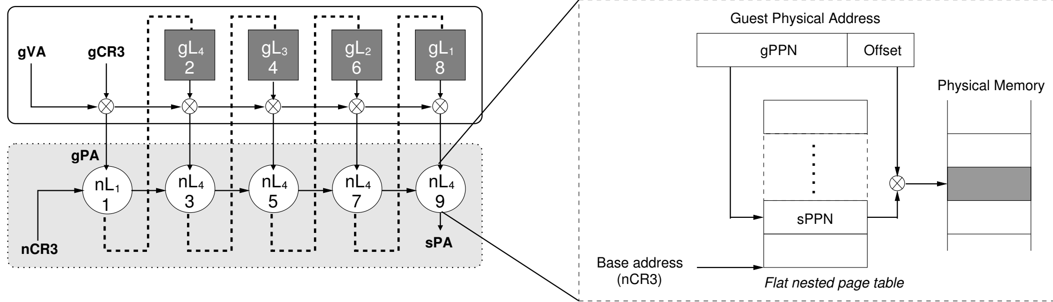

Figure 4: Page table walks with a flat nested page table

or three times of the system memory size. Furthermore, some hypervisor implementations do not even support the overcommitment of the total VM memory [1].

Thirdly, unlike processes in a native system, most of which use a tiny fraction of virtual memory space, virtual machines use much of the guest physical memory. When a VM is created, an administrator must consider the memory requirement for the VM, and attempt to assign an appropriate memory size. If the majority of the guest physical memory space is used, multi-leveling the nested page table for the VM does not reduce the memory overhead for the nested page table.

Due to the three major differences of virtual machines from processes, multi-level nested page tables do not have significant memory savings for virtual machines. The number of nested page tables is relatively small, as the number of VMs is limited. Each nested page table needs to cover a much smaller range of the guest physical address space than that of the virtual address space. Furthermore, VMs use much of the guest physical space, and thus require the nested page table entry anyway for the space. Therefore, for many of the common use cases of virtual machines, multi-level nested page tables are not necessary, and they only increase the number of references unnecessarily in two-dimensional page walks.

3.2 Flat Nested Page Tables

In this section, exploiting the unique property of VM memory usage, we propose a technique to reduce the overhead of nested page table walks. The technique, called *flat nested page table*, reduces the number of memory references for nested walks by eliminating intermediate page walks. Figure 4 depicts how the technique simplifies page walks for virtualized systems.

As discussed in Section 3.1, the benefit of multi-level page tables is marginal for nested page tables, and thus the hardware page walker can simply use a flat page table walk for nested page tables. Flat page table walks are much simpler than multi-level page table walks, and the support for them can be added to the current multi-level page table

	Process	VM(4GB)
# of pages	2^{48} / 4KB = 68,719,476,736	4 x 2^{30} / 4KB = 1,048,576
page table size	# of pages x 8B = 536870912MB	# of pages x 8B = 8MB

Table 1: Memory consumption for a flat page table: process vs. virtual machine (VM)

walker without any significant increase of complexity. The nested page table pointer (nCR3) has the starting address of the flat table. The address of the missed page table entry is calculated directly by adding the offset to the nCR3 value.

The technique does not necessarily replace the current multi-level nested page table walkers. Instead, the technique can co-exist with the current walkers. For certain virtualized systems, which require multi-level nested page tables, the conventional walker can be used. If such a fine-grained memory management is not necessary, the proposed technique can be used to improve the system performance. The flat nested page tables can be added on top of the currently available HW 2D walkers with a negligible increase of complexity.

Figure 4 depicts page walk procedures with flat nested page tables. When the level of a guest page table is *m*, the number of memory references is reduced to *2m+1* with a flat nested page table. In the four-level page table walks in the 64-bit x64 architecture, a TLB miss requires only 9 references with the flat nested page table, reducing 15 references from the current 24 references with a four-level nested page table.

As discussed in Section 3.1, even if a flat nested page table is used for each VM, it does not increase the memory overhead for nested page tables significantly. Only a limited number of VMs run in a system. The flat page table size for a nested page table for a VM is much less than that for a guest page table for a process, since a nested page table needs to cover only the guest physical memory space. For example, as shown in Table 1, for a VM with a 4GB guest-physical memory, if 4KB page size is used, the size of a flat nested page table is 8MB with 8B for each entry. However,

for a process with a 48-bit virtual address space, the flat page table size is 512GB, and thus supporting such a flat page table is impossible for processes.

Page Walk Cache (PWC) Policy: Using flat tables for nested paging can reduce the capacity requirement of PWC, as nested page table entries do not need to be in the PWC. Nested page table entries are still cached in NTLB, but without any intermediate entries with flat tables, the entire capacity of PWC can be used for guest page table walks (1D PWC).

Supporting Large Pages: Supporting large pages is also possible with flat page tables. A flat page table has the same number of entries for the smallest supported page size, regardless of actual page granularity. For a large page, all the entries corresponding the large page are marked as a large page entry. Only the first entry among the multiple entries for the large page, has an actual physical frame number. For a TLB miss, the page number at the smallest page granularity is used to access the flat page table. If the entry is not the first entry, an additional page table access occurs to fetch the physical frame number from the first entry of the large page. Such indirect accesses are necessary, as the page size is unknown when a TLB miss occurs.

Supporting flat page tables does not require any significant changes from current multi-level nested page table walks. Therefore, a system may support flat nested tables as well as multi-level nested tables, and the hypervisor may select different page table types for VMs considering their memory sizes and usage patterns.

4 Speculative Inverted Shadow Paging

Reorganizing nested page tables to flat tables reduces memory references to the nested page table for each TLB miss. However, it does not reduce the overhead of multi-level page walks for guest page tables. An alternative page table organization is an inverted page table, which can potentially fetch a page table entry by a single memory reference. In this section, we propose an inverted shadow page table, which can retrieve a direct mapping from a guest-virtual page to a system-physical page. However, maintaining direct translations from guest-virtual pages to system-physical pages has the same overhead of synchronization with guest page tables as the traditional shadow paging. We show how speculation can eliminate the hypervisor interventions for such synchronization.

4.1 Trade-offs of Shadow Paging

Shadow paging provides direct translations from guest virtual to system physical pages, which can potentially eliminate walks for nested page tables. However, maintaining shadow page tables incurs significant performance and memory overheads, often exceeding its benefits. In this section, we discuss three limitations of shadow paging, synchronization costs, extra memory overheads, and multi-level page walks.

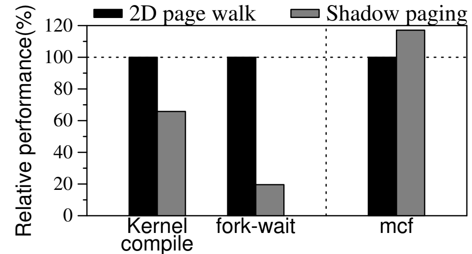

Figure 5: HW 2D page walk vs. shadow paging: relative performance with three example applications

The primary performance overhead of shadow paging is the cost of hypervisor interventions. Exiting from a guest VM context to the hypervisor context (*vm-exit*) incurs a significant performance overhead in virtualized systems. To maintain consistent shadow pages, there have been two types of mechanisms to detect guest page table changes. Firstly, the hypervisor restricts the memory area of guest page tables to be read-only. Any attempt by a guest OS to update guest page tables will be caught by the hypervisor. An alternative way, called virtual TLB, is to let a guest OS to modify a guest page table, and detect the change later if the corresponding shadow page table entry does not exist during an actual memory access to the page. In the mechanism, TLB invalidation instructions also invoke the hypervisor, as the guest OS executes them for TLB consistency, when it removes page table entries from guest page tables.

If a page fault occurs, the hypervisor must intervene and may have to traverse both the guest and shadow page tables, since it does not know whether the page fault must be handled by the guest OS or hypervisor. Furthermore during a process context switch within a guest VM, the hypervisor should replace the guest page table pointer with the corresponding shadow page table pointer, so that the HW page table walker uses the shadow page table, instead of the guest page table.

To compare the performance behaviors of shadow paging and hardware-based 2D page walk techniques, we examined three selected workloads on the Xen hypervisor running on an Intel Xeon E5530 system. The processor is equipped with EPT (Extend Page Table), which supports hardware-based 2D page walks. Two workloads, `kernel-compile` and `fork-wait`, represent a case when *vm-exit* operations occur frequently with shadow paging for synchronization. `Fork-wait` is a simple micro-benchmark which repeats process creation and destruction operations aggressively. Figure 5 shows the performance comparison of shadow paging with HW-based 2D page walks. The two benchmarks exhibit much worse performance with shadow paging than with HW-based 2D page

walks due to the excessive hypervisor interventions. In the figure, `mcf` represents a case which exhibits high TLB misses with low shadow page synchronization overheads. Since shadow paging can potentially reduce the memory references for retrieving a translation for each TLB miss, the performance of `mcf` is 17% better with shadow paging than with HW-based 2D page walks. Prior studies reported similar performance trade-offs between shadow paging and HW-based 2D page walks [20, 21, 22].

Another cost of shadow paging is the extra memory for shadow page tables for guest processes. To reduce such a memory overhead, some hypervisors use in-memory caches for shadow page tables, not to keep shadow page tables in memory for all processes in guest VMs. However, if a miss occurs for the in-memory shadow page cache, the missed shadow page table must be reconstructed with the guest page table and nested page table of the VM. Xen statically allocates the memory reserved for shadow page tables for each guest virtual machine. The amount of reserved memory for shadow pages increases, as the number of virtual CPUs and the configured memory for a guest VM increase. For example, a VM, configured with 1 vCPU and 4GB, consumes 33MB memory to maintain the shadow page tables on the Xen hypervisor.

The last limitation of traditional shadow paging is that it uses the same multi-level page walks as guest page tables. Shadow paging had been invented to support 2-level address translation with a legacy 1D HW page walker, and thus it assumes the same multi-level page table organization as traditional guest page tables. In this paper, we propose a new page table organization, which has the benefit of shadow paging providing direct translations from guest-virtual to system-physical pages, while reducing memory references to walk multi-level page tables.

4.2 Hashed Inverted Shadow Page Tables

One of the traditional alternatives to multi-level page tables is a hashed inverted page table [6, 15]. A hashed inverted page table organizes a page table as a hash table, and searches a page table entry by a hash of the virtual page number and process identifier. One inverted page table is necessary for the entire system, and its size is often proportional to the system memory size. PA-RISC supports such a hashed inverted page table [14]. With the hashed table, the best case requires only one or two memory references if there is no hash collision. However, a collision case may require to search the inverted page table with a chain of many references.

A slightly different approach from the inverted page table is to use a directly accessible in-memory cache of the complete page table. Translation Storage Buffers (TSB) used by the SPARC architecture is an in-memory direct-mapped cache of the complete address translation [2]. The

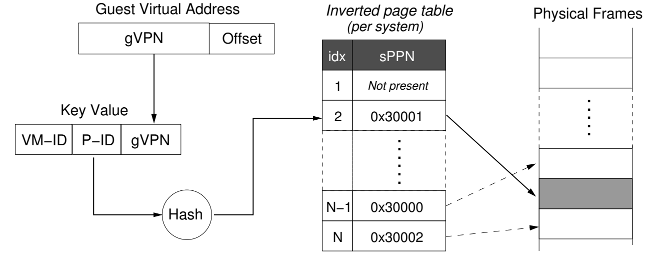

Figure 6: Page table walks with a hashed inverted shadow page table

main difference of TSB from the inverted page table is that TSB does not require a chain of references to the inverted table. If the missed translation entry is not found in the corresponding TSB entry, the operating system will search the complete translation table. TSB lookups can be done either by a software fault handler or by a hardware TSB walker.

A similar directly indexed translation caching can be used for virtualization. Traditional shadow paging uses a duplicate shadow page table for each guest process. Instead of such a per-process shadow page table, we propose a system-wide hashed inverted shadow page, indexed by a hash of the VM identifier, process ID, and guest virtual page number as shown Figure 6. If the corresponding hashed shadow page entry contains the mapping, a TLB miss can be handled by a single memory reference. If the entry does not contain the requested mapping, a page walk for the complete guest and nested page tables is initiated. As the hashed shadow page is a translation cache for nested translation, it does not replace nested page tables, which are necessary to keep a complete memory mapping for each virtual machine. Note that unlike traditional inverted page tables, the proposed inverted shadow page table does not include a tag in each entry to verify VM identifier, process identifier, and virtual page number. It can eliminate the tag information, since the translations in the inverted page table will be used only speculatively, as discussed in the next section.

The inverted shadow page table does not require the modification of the current page table organization specified in the ISA. However, the hardware page walker must be modified to additionally support a lookup mechanism for the inverted shadow page table. Since the inverted shadow page table is an in-memory cache of the complete translation, the hardware page walker must update the inverted page table if the translation is not in the inverted page table for a TLB miss. The hypervisor only needs to reserve the system memory for the inverted shadow page table. As a system-wide cache of translation, the inverted shadow page may reduce the memory overhead for duplicating all guest page tables in the traditional shadow paging. Even if the number of VMs or vCPUs increases, the size of the system-wide inverted shadow page table does not need to increase. With such a hardware-based management of the inverted

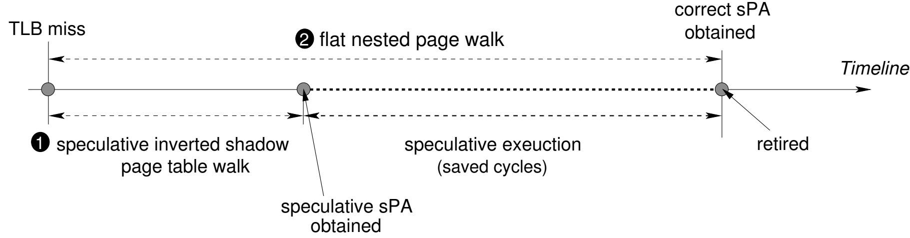

Figure 7: Speculative page table walk with SpecISP

shadow paging, it requires only one memory reference to fetch a direction translation from a guest virtual page to a system physical page, if the request hits on the inverted table.

However, such an inverted shadow page table has the same critical problem as the traditional shadow page table. If a guest OS modifies a guest page table, the hypervisor must intervene to update or invalidate the cached entry in the inverted shadow page table. In this paper, we propose a speculative mechanism to eliminate such synchronization by the hypervisor completely.

4.3 Speculatively Handling TLB misses

Using speculation to reduce the cost of TLB misses is not new. SpecTLB first proposed to use speculation to predict address translation [7]. For a TLB miss, the processor continues to execute with a predicted translation, while a traditional page walk will verify the prediction. The speculative mechanism uses the same recovery mechanism commonly found in the current processors supporting speculative executions with re-order buffers or checkpointing [13].

In this paper, we use speculation to eliminate the synchronization between guest page tables and the inverted shadow page table. *Speculative Inverted Shadow Paging (SpecISP)* eliminates the needs for hypervisor interventions, even if a guest page changes. Instead, the inverted shadow page table is allowed to have the obsolete mapping information. For a TLB miss, the processor can speculatively execute with a mapping found in the inverted shadow page table. Simultaneously, the nested page walks will fetch the correct translation to verify whether the speculation is correct.

Figure 7 depicts the simultaneous walks for the speculative inverted shadow page table (1), and for the non-speculative flat nested page table (2). The speculative walk will finish much earlier than the non-speculative walk for the majority of TLB misses, as it requires only a single memory reference. However, it is also possible that the inverted shadow page table lookup may take longer than the non-speculative flat table lookup. If the lookup of the inverted shadow page table is a cache miss, and all the memory references for the guest and nested page table lookups are cache hits, the speculative lookup may take longer than the non-speculative lookup. In Section 5.3, we will show differences in latencies between speculative and non-speculative page walks with our simulated system.

If a speculative execution is correct, the inverted table organization allows SpecISP to retrieve a translation entry with a single memory reference, in contrast to multiple memory references in 2D page table walks with HW-based nested paging, or 1D page walks with shadow paging. As misspeculation rates are relatively low for many applications, SpecISP excels both HW 2D walking and shadow paging for common cases.

Even if a misspeculation occurs, the latency for a TLB miss is bounded by the latency of the correct nested page walk and the recovery cost for speculative execution. SpecISP excels shadow paging when changes in guest page mappings occur frequently. Misspeculation costs with SpecISP are much smaller than those of hypervisor interventions in shadow paging. A page mapping change will cause SpecISP to misspeculate due to the obsolete information in the inverted table. However, unlike shadow paging which requires hypervisor interventions with long latencies, such a misspeculation can be resolved quickly in SpecISP, since a simultaneous correct nested page walk will finish quickly with the HW-based nested page walker.

SpecISP does not require significant extra supports from conventional speculative execution cores. The conventional speculative execution capability of out-of-order cores for branch and memory speculation is enough to support SpecISP. The added logic is to compare the speculative system physical address and permission bits with the correct translation when the non-speculative nested page table walk completes for a TLB miss. For a simple in-order processor, it is necessary to add a register checkpointing mechanism and a store queue to support speculation.

5 Evaluation

5.1 Methodology

To evaluate the proposed schemes, we use the Simics full-system simulator [16], running the Xen hypervisor (version 4.0.1) [5] on the simulator. Using the real hypervisor improves the accuracy of the evaluation, as it includes

Parameter	Value
Processors L1 I/D Cache L2 Cache	In-order x86 processor 1-cycle, 32KB, 4-way, 64B block 12-cycle, 512KB, 8-way, 64B block Average L2 miss latency: 100 cycles
Instruction TLB	1-cycle, 32-entry, fully assoc. L1 2-cycle, 512-entry, 4-way, L2
Data TLB	1-cycle, 64-entry, fully assoc. L1 2-cycle, 512-entry, 4-way, L2
Page Walk Cache	24-entry, fully assoc. 2-cycle PWC access Flushed on each TLB flush
Nested TLB	16-entry, fully assoc. 2-cycle NTLB access Never flushed during guest execution

Table 2: Simulated system configurations

SPECint Applications	
Application	gcc, mcf, sjeng, libquantum omnetpp, astar, xalancbmk
Dataset	Reference input
Commercial Applications	
SPECjbb 2005	4 warehouses
SPECweb 2005	100 simultaneous sessions
RUBiS (like ebay)	Default bidding workload Apache, PHP, MySQL
OLTP	MySQL
OrderEntry	Swingbench (Oracle)
StressTest	Swingbench (Oracle)
KernelCompile	linux kernel 2.6.38
Volano (chatting server)	50 rooms, 1000 connections Apache, Sun-JVM

Table 3: Application input data and parameters

the effect of hypervisors on TLB and cache behaviors. A custom memory hierarchy model has been augmented to the Simics simulator to model multi-level caches and TLBs. The simulated system has a single core to reduce simulation times, and the core has 32KB L1 instruction and data caches, and a 512KB L2 cache. The system uses separate two-level TLBs for instruction and data. The separate L2 TLB for each instruction and data has 512 entries. All page table entries, including intermediate translations, can be cached in the L2 cache.

We use a simple in-order processor model for the x86 ISA, due to the limitation of our simulation infrastructure. To the in-order model, we have added a checkpointing mechanism for speculative execution. However, if out-of-order execution cores supporting speculative execution, are available, it is not necessary to add such an extra checkpointing mechanism. For a TLB miss, a register checkpoint is created, and during a speculative execution period, up-to 24 stores can be buffered in the store buffer. If no more stores can be buffered in the store buffer, the core is stalled. The speculative execution capability limited by the store buffer size, is similar to the latest out-of-order execution cores with more than 100 outstanding instructions and 24 stores [12].

The baseline system uses four-level two-dimensional page walks for both guest and nested page tables as shown in Figure 2. To compare the proposed schemes to the latest advancements for reducing TLB miss latencies, the baseline system has a page walk cache (PWC), which can cache 2D intermediate page table entries, and a nested TLB (NTLB). We use the same PWC and NTLB management policies as discussed in Bhargava et al [8]. The details of system configurations are shown in Table 2.

On top of the Xen hypervisor, a guest virtual machine, which uses a Ubuntu distribution based on Linux kernel 2.6.18, and the domain 0 virtual machine are running. The domain0 virtual machine is a special virtual machine, which handles I/O devices. On the guest VM, we run our benchmark, seven applications selected from the SPECint 2006 benchmark (`SPECint`), and seven commercial applications (`commercial`). Table 3 describes the details of benchmark applications.

The execution times shown in the section are all normalized to the execution time with a state-of-the-art hardware 2D page walker with both PWC and NTLB (`baseline`). The flat nested walker (`flat`) uses 1D PWC and NTLB. We also show an ideal configuration with the perfect TLBs (`perfect-TLB`), which does not have any TLB misses. Speculative inverted shadow paging can use either flat page tables (`SpecISP w/ flat`) or 4-level nested page tables (`SpecISP w/4-level`), as non-speculative backing page tables.

5.2 Flat Nested Page Tables

In this section, we first evaluate how much performance improvement a flat nested page table can achieve, compared to the baseline state-of-the-art 2D page walker. Figure 8 shows the normalized execution times with a flat page table. The flat nested page table, requiring a very minor addition to the current page walker logic, improves the performance for all the applications from the baseline. On average, the flat nested page table reduces the execution times by 5% (`SPECint`) and 8% (`commercial`) from `4-level 2D PWC+NTLB`. The improvements are high in RUBiS and Volano, by 10% and 12% respectively. The performance improvements by the flat page table may be relatively modest for `SPECint`, but the improvements are significant for the `commercial` workloads. The ideal `perfect-TLB` has further potential performance improvements of 11% for `SPECint`, and 16% for `commercial` from the flat nested page table.

Table 4 presents the rates of TLB misses, and the number of L2 cache accesses and hit rates to handle TLB misses, comparing 4-level and flat nested page walks. For a TLB miss, if a nested page walk can be served by PWC or NTLB, the page walk does not need to access the L2 cache. However, if the corresponding entry is found neither in the PWC

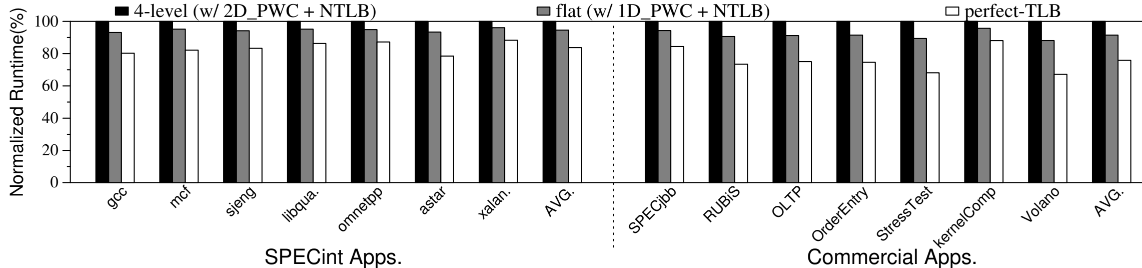

Figure 8: Execution times of flat vs. 4-level nested page table (normalized to the baseline)

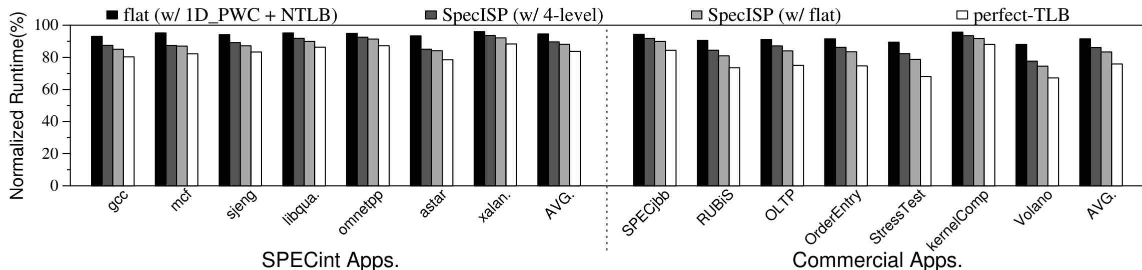

Figure 9: Execution times of SpecISP vs. flat nested paging (normalized to the baseline)

		L2 accesses for page walks			
	TLB	4-level (2D_PWC + NTLB)		flat (1D_PWC + NTLB)	
Workloads	Misses	Accesses	Hit rate	Accesses	Hit rate
gcc	11,454	44,364	99.4%	31,625	99.5%
mcf	36,461	86,193	98.3%	73,583	99.1%
sjeng	9,803	38,921	99.0%	28,114	99.1%
libquantum	6,949	26,800	99.3%	19,070	99.4%
omnetpp	5,489	20,216	82.3%	13,478	85.2%
astar	22,788	58,999	88.3%	50,738	92.1%
xalancbmk	6,212	21,233	98.1%	15,753	98.4%
SPECjbb	6,518	25,486	91.7%	18,220	93.2%
RUBiS	16,703	67,823	98.6%	47,539	98.7%
OLTP	14,176	56,661	97.4%	39,738	97.9%
OrderEntry	16,771	71,580	95.1%	50,425	95.1%
StressTest	26,952	110,326	96.4%	79,484	96.3%
KernelCom.	5,902	22,077	98.6%	14,893	98.6%
Volano	31,404	129,978	97.9%	88,316	97.9%

Table 4: TLB misses and L2 cache accesses for page table entry references (per 1M instructions)

nor NTLB, the walker accesses the L2 cache.

Using a flat nested page table reduces L2 accesses to serve TLB misses significantly. On average, the L2 accesses are reduced by 28% with the flat nested page table, as the flat table requires fewer memory references than the multi-level page walk in the baseline. However, the L2 cache hit rates for both of the four-level and flat nested page tables are very high for most of applications. Due to such high L2 hit rates for page table walks and relatively low TLB miss rates for some applications, the performance benefit of reducing page walk references with the flat nested page table is modest for them as shown in Figure 8.

5.3 Speculative Inverted Shadow Paging

In this section, we evaluate the performance benefit of speculative inverted shadow page tables (SpecISP). The inverted shadow page table reduces the number of references for guest page table lookups as well as nested page table lookups. Figure 9 presents the normalized execution times with SpecISP. The inverted shadow paging with flat nested paging reduces the average execution times by 6% and 8% from `flat`, for `SPECint` and `commercial` respectively. Compared to `baseline`, it reduces the execution times by 12% and 17%. Also, the execution times with SpecISP become close to those with the perfect TLBs, with 4% and 7% performance differences for `SPECint` and `commercial`, from the ideal configuration.

Using traditional 4-level nested page tables as the backing page tables for SpecISP reduces its effectiveness slightly. However, we expect that by increasing the depth of speculative execution, SpecISP can potentially use either flat page tables or 4-level page tables, achieving similar performance improvements.

Figure 11 shows the cumulative distributions of latencies serving a TLB miss. Depending on whether references hit on the PWC or L2 cache, the TLB miss latencies vary. The figure shows two curves. The dotted line is the distribution of latencies for walking through the speculative shadow page table, and the solid line is for the non-speculative flat page table walk. The latencies through the inverted shadow paging are less than 12 cycles for more than 95% of TLB misses, as it requires a single lookup of the inverted shadow page table entry. The lookups commonly hit on the

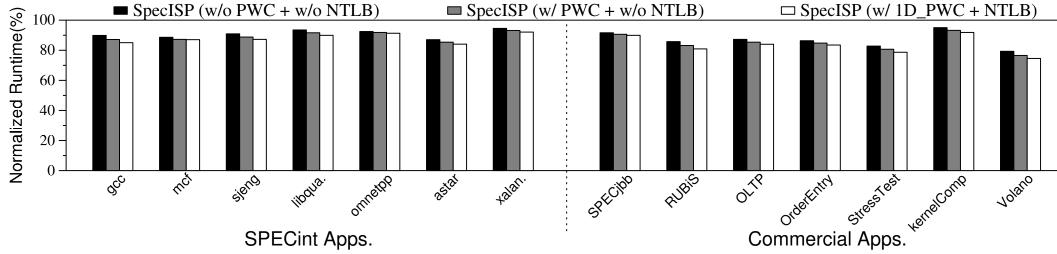

Figure 10: Sensitivity to PWC and NTLB with SpecISP (normalized to the baseline)

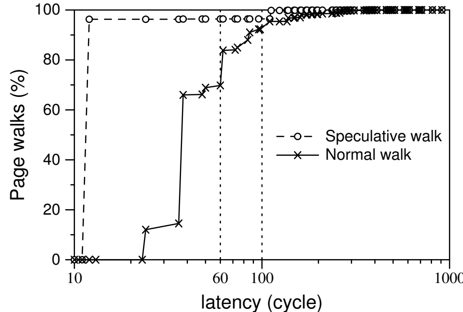

Figure 11: Cumulative distributions of TLB miss latencies for the Volano benchmark

Workloads	Mis-spec. rate	Workloads	Mis-spec. rate
gcc	2.072%	SPECjbb	0.057%
mcf	0.008%	RUBiS	0.051%
sjeng	0.150%	OLTP	0.000%
libquantum	2.400%	OrderEntry	0.008%
omnetpp	0.000%	StressTest	0.005%
astar	0.000%	KernelCompile	5.312%
xalancbmk	0.672%	Volano	0.000%

Table 5: Misspeculation rates with SpecISP

L2 cache. However, the latencies of non-speculative flat page table walks are within 60 cycles for 70% and 92 cycles for 90% of TLB misses. The latency differences are significant enough to justify accessing the inverted shadow paging speculatively, as speculative accesses return translations much faster than non-speculative accesses. Furthermore, the differences between two accesses are within 90 cycles for the majority of TLB misses. The differences are reasonably small enough to wait the verification by non-speculative accesses with the out-of-order execution capability in the current microprocessors.

Table 5 presents how often misspeculation occurs with the inverted shadow paging. For `gcc`, `libquantum`, and `kernelCompile`, there are relatively frequent changes of guest page tables, and thus misspeculation rates are high (2.1-5.3%). However, the rest of applications show very low rates of misspeculation. These low misspeculation rates indicate that the proposed speculative mechanism does not have significant overheads for pipeline flushes due to misspeculation. However, even if a misspeculation occurs, the cost of handling the misspeculation is much lower than the cost of hypervisor interventions in shadow paging.

5.4 Sensitivity Studies

Removing NTLB and PWC: Page Walk Cache (PWC) and Nested TLB (NTLB) reduce memory references to fetch translations for guest page tables and nested page tables in 2D nested page walks. With the proposed SpecISP, their benefits will decrease, as speculative execution can hide long latencies of accessing the correct mapping. If the performance gain with PWC and NTLB is low, the structures can be removed to reduce the area and power costs. In this section, we evaluate the impact of removing PWC and NTLB from SpecISP.

Figure 10 presents two additional design configurations with SpecISP. The first bar represents a SpecISP configuration with neither PWC nor NTLB for the non-specultive 2D walk, and the second bar represents a configuration with only PWC. The last bar includes both of them. The results indicate that there are no significant performance differences among the configurations. Since the accuracy of speculative execution is high, and the speculative execution hides extra latencies increased due to the lack of PWC or NTLB, the increased latencies for getting the correct mapping do not influence the overall performance significantly.

Sensitivity to Speculation Depth: In this section, we discuss how sensitive the overall performance is to speculative execution capability. In our checkpoint-based simulation, reducing the store buffer size limits the capability of speculative execution. To evaluate the effect of reduced speculation depth, we run experiments with a 12-entry store buffer, comparing them to the results with a 24-entry store buffer. Even with the store buffer size decreased by half, there is no noticeable effect on the performance of SpecISP (0.3% decrease on average). Even if we assume an infinite store buffer, the execution times do not improve significantly with only 1% improvement on average. The main reason for the insensitivity to speculation depth is that stalls due to the limited speculation depth occur rarely. The time period of speculative execution is short, with less than 92 cycles for more than 90% of all the speculation cases.

6 Conclusions

This paper explored how the nested address translation mechanism for virtualization may evolve, if the distinct characteristics of memory management for virtual machines are considered for the architectural supports. We proposed and evaluated two schemes to reduce the overheads of nested address translation in virtualized systems. Firstly, flat nested page tables reduce memory references required for 2D page walks. Supporting flat nested page tables in current nested page table walkers should require a minor change with little extra hardware. Secondly, speculative inverted shadow paging can reduce the cost of a nested page walk to a single memory reference in common cases, without hypervisor interventions for guest page table changes. With minor changes from the current HW 2D walkers, flat nested page tables can reduce the average execution time by 7% over a state-of-the-art 2D page walker with the PWC and NTLB. With more extensive changes than flat nested paging, SpecISP improves the overall performance by 14% from the state-of-the-art 2D page walker.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Satish Narayanasamy, for their invaluable comments. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0026465).

References

[1] Microsoft Hyper-V Server. `http://www.microsoft.com/hyper-v-server/en/us/default.aspx`.

[2] UltraSPARC III Cu Users' Manual.

[3] AMD-V Nested Paging, 2008.

[4] 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.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, 2010.

[7] T. W. Barr, A. L. Cox, and S. Rixner. SpecTLB: a mechanism for speculative address translation. In *Proceeding of the 38th annual international symposium on Computer architecture (ISCA)*, 2011.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. 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)*, 2008.

[9] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level TLBs for chip multiprocessors. In *proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[10] A. Bhattacharjee and M. Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[11] A. Bhattacharjee and M. Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the 15th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2010.

[12] M. Butler. AMD "Bulldozer" Core - a new approach to multithreaded compute performance for maximum efficiency and throughput. HotChips, 2010.

[13] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, 2009.

[14] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA)*, 1993.

[15] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the 8th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 1998.

[16] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, feb 2002.

[17] S. Srikantaiah and M. Kandemir. Synergistic TLBs for high performance address translation in chip multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[18] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the 15th ACM symposium on Operating systems principles (SOSP)*, 1995.

[19] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. K?gi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38:48–56, 2005.

[20] VMware. Performance Evaluation of AMD RVI Hardware Assist. Technical report, 2009.

[21] VMware. Performance Evaluation of Intel EPT Hardware Assist. Technical report, 2009.

[22] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. In *Proceedings of the 7th international conference on Virtual execution environments (VEE)*, 2011.