SpecTLB: A Mechanism for Speculative Address Translation

Thomas W. Barr, Alan L. Cox, Scott Rixner

Rice University
Houston, TX
{twb, alc, rixner}@rice.edu

ABSTRACT

Data-intensive computing applications are using more and more memory and are placing an increasing load on the virtual memory system. While the use of large pages can help alleviate the overhead of address translation, they limit the control the operating system has over memory allocation and protection. We present a novel device, the SpecTLB, that exploits the predictable behavior of reservation-based physical memory allocators to interpolate address translations.

Our device provides speculative translations for many TLB misses on small pages without referencing the page table. While these interpolations must be confirmed, doing so can be done in parallel with speculative execution. This effectively hides the execution latency of these TLB misses. In simulation, the SpecTLB is able to overlap an average of 57% of page table walks with successful speculative execution over a wide variety of applications. We also show that the SpecTLB outperforms a state-of-the-art TLB prefetching scheme for virtually all tested applications with significant TLB miss rates. Moreover, we show that the SpecTLB is efficient since mispredictions are extremely rare, occurring in less than 1% of TLB misses. In essense, the SpecTLB effectively enables the use of small pages to achieve fine-grained allocation and protection, while avoiding the associated latency penalties of small pages.

Categories and Subject Descriptors

C.0 [**General**]: Modelling of computer architecture; C.4 [**Performance of systems**]: Design studies; D.4.2 [**Operating Systems**]: Virtual Memory

General Terms

Performance, Design

Keywords

TLB, Memory Management

ISCA'11, June 4–8, 2011, San Jose, California, USA.

1. INTRODUCTION

The memory capacity of modern computers is growing at a much faster rate than the size of translation-lookaside buffers (TLBs). Therefore, TLB misses are becoming more frequent and address translation is an increasingly significant performance bottleneck. Recent work has shown that the impact of address translation on overall system performance ranges from 5-14% even for nominally sized applications in a non-virtualized environment [5]. Larger applications have even higher overhead, approaching 50% in some cases [17].

Virtualization compounds address translation overhead. Nested paging, a popular mechanism for hardware supported memory virtualization, increases virtual memory overhead dramatically. Translating a single guest virtual address to a machine physical address can take as many as twenty-four memory accesses on x86-64. Consequently, the overhead of virtual memory increases to up to 89% for real-world workloads when using nested paging [5].

The use of large pages reduces the performance overhead of virtual memory by increasing TLB coverage. Each entry in the TLB that holds a large page covers a larger region of virtual memory, so the entire TLB is able to translate a larger region of the address space. However, this increased coverage does not come for free.

The page is the unit of a program's address space that the operating system can allocate and protect. The operating system must allocate an entire page of physical memory at a time, regardless of how much of that space will be used by the application. Always allocating large pages leads to excessive physical memory use if a program fragments its virtual memory use. More critically, application permissions (read/write/execute) must be consistent across an entire page, large or small. Finally, the operating system can tell if memory has been modified or accessed on a page granularity. It uses this facility to know when to write back portions of memory mapped files. This means that if large pages are used for memory mapped files, the operating system must write back the entire large page even if only a single byte was modified.

On x86-64, pages are only available in three sizes: 4KB, 2MB and 1GB. The large gap between these sizes makes page size selection critical. Using too small of a page overburdens the TLB, while using too large of a page can waste physical memory and have high I/O overhead. The proper page size for a region of memory may change from application to application or even from execution to execution. Therefore, some modern operating systems take an automatic approach to selecting page size.

FreeBSD's *reservation-based physical memory allocator* uses small pages for all memory by default. After a program uses every 4KB page within an entire 2MB region of virtual memory, that contiguous region is *promoted* to a large page. To prepare for this, the operating system places small pages that might be promoted in

the future into large page reservations. In a reservation, 4KB pages are aligned within a 2MB region of physical memory corresponding to their alignment within their 2MB region of virtual memory. This means that within a reservation, consecutive virtual pages will also be consecutive physical pages [19].

Processor architectures must evolve to meet the demands of modern operating systems [18]. The challenges involved in address translation require innovative architectural solutions that provide support for such reservation-based physical memory allocators. In this paper, we present the *SpecTLB*, a novel TLB-like structure that exploits the contiguity and alignment generated by a reservation-based physical memory allocator to interpolate address translations based on the physical address of nearby pages. While the underlying page table still must be walked to verify these speculative translations, this walk can be done concurrently with speculative memory access and execution. This new capability allows the operating system to maintain fine-grained protection and allocation over memory while hiding the latency of the resulting TLB misses.

We show that the SpecTLB is able to eliminate the latency penalty from a majority of TLB misses with an unmodified version of FreeBSD. However, one of the key contributions of the SpecTLB is its ability to achieve large-page like performance when large pages are impractical to use, such as in virtualization. Traditional hypervisors implement I/O by marking pages of the guest physical address space that contain memory mapped I/O as unavailable. When the guest system accesses them, the hypervisor is invoked which emulates the I/O device. Ideally, the physical memory space of a virtualized guest would be stored as a small number of 1GB pages. However, the guest physical pages that the guest operating system will use for I/O are fixed, defined by the x86 architecture itself. Therefore, the hypervisor must have fine-grained protection control over those pages. With a speculative TLB, this space can be stored in a 1GB reservation with both data and emulated I/O pages mixed together. All accesses are made speculatively, as if they were to a data page. Therefore, all guest physical memory accesses can proceed without blocking for a TLB miss. If the address turns out to be part of a data region, the speculative work is committed. However, if an address is part of an emulated I/O region, the speculative execution will not be committed, and the hypervisor will be invoked as before.

We evaluate the SpecTLB using a microarchitecture-independent simulation. We quantify how often TLB misses are predictable and how many MMU-related memory accesses can be parallelized with our system. Our results show that the SpecTLB can remove 40% of the high-latency DRAM accesses related to memory management from the critical path of execution. Moreover, prediction accuracy is very high, exceeding 99% for most benchmarks. Finally, a SpecTLB only needs to be of moderate size, approaching maximum hit-rate with tens of entries on our benchmarks.

Additionally, we show that the SpecTLB compares well against TLB prefetching. We simulate a previously described TLB prefetching scheme [12] and show that our system adapts more readily to applications with random access patterns and performs unused speculative work less frequently.

The rest of this paper is organized as follows. Section 2 describes the format of the x86-64 page table and the operation of a reservation-based physical memory manager. Section 3 discusses the design of the SpecTLB. Section 4 describes our simulator. Sections 5 and 6 discuss our simulation results for applications running natively and under nested paging, respectively. Section 7 discusses related work. Finally, we conclude in Section 8.

63:48	47:39	38:30	29:21	20:12	11:0
se	L4 idx	L3 idx	L2 idx	L1 idx	*page offset*

Figure 1: Decomposition of the x86-64 virtual address.

2. BACKGROUND

The SpecTLB is intricately tied to the operation of address translation and pysical memory allocation. This section provides background information that explains how traditional x86 address translation works, how address translation is extended to handle virtualization, and how a reservation-based physical memory allocator operates. This background provides context to help explain how the SpecTLB is able to improve performance by hiding the latency of TLB misses.

2.1 x86 Address Translation

All x86 processors since the Intel 80386 have used a radix tree to record the mapping from virtual to physical addresses. Although the depth of this tree has increased, to accommodate larger physical and virtual address spaces, the procedure for translating virtual addresses to physical addresses using this tree is essentially unchanged. A virtual address is split into a page number and a page offset. The page number is further split into a sequence of indices. The first index is used to select an entry from the root of the tree, which may contain a pointer to a node at the next lower level of the tree. If the entry does contain a pointer, the next index is used to select an entry from this node, which may again contain a pointer to a node at the next lower level of the tree. These steps repeat until the selected entry is either invalid (in essence, a NULL pointer indicating there is no valid translation for that portion of the address space) or the entry instead points to a data page using its physical address. In the latter case, the page offset from the virtual address is added to the physical address of this data page to obtain the full physical address. In a simple memory management unit (MMU) design, this procedure requires one memory access per level in the tree.

Figure 1 shows the precise decomposition of a virtual address by x86-64 processors [1]. Standard x86-64 pages are 4KB, so there is a single 12-bit page offset. The remainder of the 48-bit virtual address is divided into four 9-bit indices, which are used to select entries from the four levels of the page table. The four levels of the x86-64 page table are named PML4 (Page Map Level 4), PDP (Page Directory Pointer), PD (Page Directory) and PT (Page Table). In this paper, however, for clarity, we will refer to these levels as L4 (PML4), L3 (PDP), L2 (PD) and L1 (PT). Finally, the 48-bit virtual address is sign extended to 64 bits (the *se* field). As the virtual address space grows, additional index fields (*e.g.*, L5) may be added, reducing the size of the sign extension.

An entry in the page table is 8 bytes in size regardless of its level within the tree. Since a 9-bit index is used to select an entry at every level of the tree, the overall size of a node is always 4KB, the same as the page size. Hence, nodes are commonly called page table pages. The tree can be sparsely populated with nodes—if at any level, there are no valid virtual addresses with a particular 9-bit index, the sub-tree beneath that index is not instantiated. For example, if there are no valid virtual addresses with L4 index `0x03a`, that entry in the top level of the page table will indicate so, and the 262,657 page table pages (1 L3 page, 512 L2 pages, and 262,144 L1 pages) beneath that entry in the radix tree page table will not exist. This yields significant memory savings, as large portions of

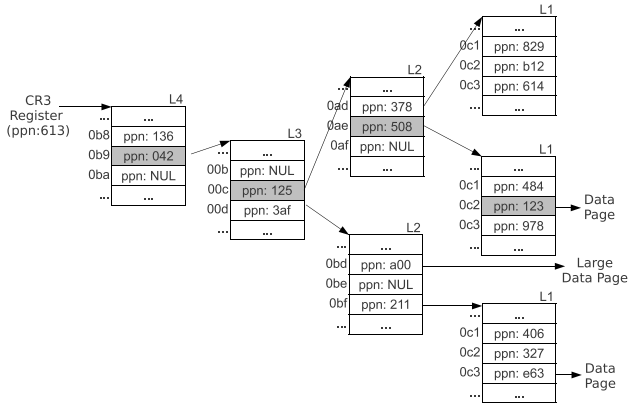

Figure 2: An example page walk for virtual address (`0b9, 00c, 0ae, 0c2, 016`). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures for simplicity.

the 256 TB virtual address space are never allocated for typical applications.

Figure 2 illustrates the radix tree page table walk for the virtual address `0x0000 5c83 15cc 2016`. For the remainder of the paper, such 64-bit virtual addresses will be denoted as (*L4 index*, *L3 index*, *L2 index*, *L1 index*, *page offset*) for clarity. In this case, the virtual address being translated is (`0b9, 00c, 0ae, 0c2, 016`). Furthermore, for simplicity of the examples, only 3 hexadecimal digits (12 bits) will be used to indicate the physical page number, which is actually 40 bits in x86-64 processors.

As shown in the figure, the translation process for this address proceeds as follows. First, the page walk hardware must locate the top-level page table page, which stores L4 entries. The physical address of this page is stored in the processor's CR3 register. In order to translate the address, the L4 index field (9 bits) is extracted from the virtual address and appended to the physical page number (40 bits) from the CR3 register. This yields a 49-bit physical word address that is used to obtain the appropriate 8-byte L4 entry (offset `0b9` in the L4 page table page in the figure). The L4 entry may contain the physical page number of an L3 page table page (in this case `042`). In which case, the process is repeated by extracting the L3 index field from the virtual address and appending it to this physical page number to obtain the appropriate L3 entry. This process repeats until the selected entry is invalid or specifies the physical page number of the actual data in memory, as shown in the figure. Each page table entry along this path is highlighted in grey in the figure. The page offset from the virtual address is then appended to this physical page number to yield the data's physical address. Note that since page table pages are always aligned on page boundaries, the low order bits of the physical address of the page table pages are not stored in the entries of the page table.

Given this structure, the current 48-bit x86-64 virtual address space requires four memory references to "walk" the page table from top to bottom to translate a virtual address (one for each level of the radix tree page table). As the address space continues to grow, more levels will be added to the page table, further increasing the cost of address translation. A full 64-bit virtual address space will require six levels, leading to six memory accesses per translation.

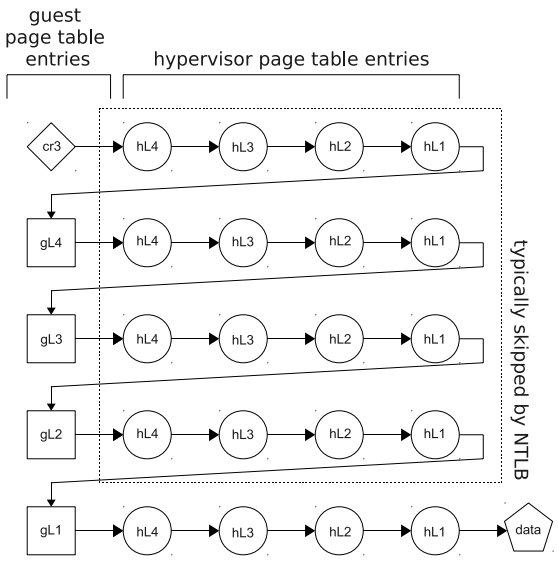

Figure 3: The two-dimensional page walk used in nested paging. Each intermediate guest physical address must be translated through the hypervisor page table.

Alternatively, an L2 entry can directly point to a contiguous and aligned 2MB data page instead of pointing to an L1 page table page. In Figure 2, virtual address (`0b9, 00d, 0bd, 123f5d7`) is within a large page. This large-page support greatly increases maximum TLB coverage. In addition, it lowers the number of memory accesses to locate one of these pages from four to three. Finally, it greatly reduces the number of total page table entries required since each entry maps a much larger region of memory. Similarly, an L3 entry can directly point to a 1GB page.

2.2 Nested Paging

To reduce virtualization overhead, AMD and Intel have implemented *nested paging* on recent processors [5, 26]. Nested paging uses two sets of page tables to translate the virtual addresses used by programs in guest virtual machines to host physical addresses. A process running in the guest accesses memory through a guest virtual address. This is first translated into an intermediate "guest physical address" through the guest's page table. Then, the guest physical address is itself translated into a host physical address using the hypervisor's page table.

On a TLB miss, the guest virtual address must first be translated to a guest physical address, then to a host physical address. This process is complicated by the fact that the guest page table uses guest physical addresses to point to each lower level of the page table. These references must themselves be translated. An example of this is shown in Figure 3. The guest page table walk is shown vertically, starting with the guest CR3 register in the upper left and proceeding downwards. However, each pointer in the guest page table must be translated through the hypervisor page table. These translations are shown as the five horizontal page table walks (hL4 through hL1). Overall, twenty-four accesses are required to various page tables per guest TLB miss.

To reduce the number of page table references required, AMD has introduced a *Nested TLB* (NTLB). This caches the results of the hypervisor translations from guest to host physical addresses for the first four translations that are for the guest page table, al-

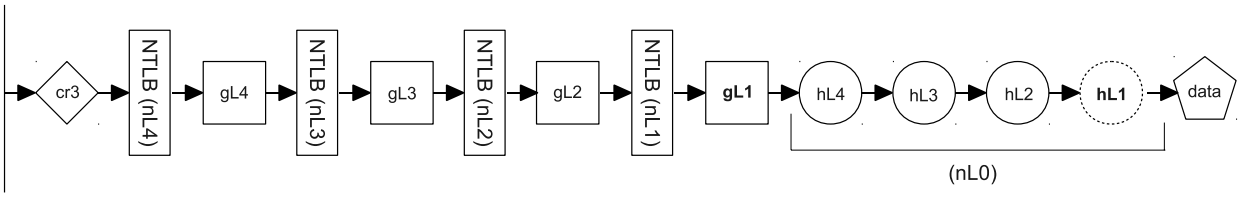

Figure 4: An example of a typical nested page walk.

lowing a nested page walk to be skipped on a NTLB hit, as shown in Figure 4. The NTLB does not cache the fifth translation, since those translations are already cached by the normal system TLB, which is far larger than the NTLB.

The cost of the fifth hypervisor translation can be reduced by using large pages in the hypervisor table. This reduces the number of memory or cache references from four to three. Ideally, a hypervisor would use a page size that is close in magnitude to the size of the entire guest physical memory space, such as the 1GB page available in x86-64. However, the hypervisor emulates guest I/O devices using permissions in the hypervisor page table. This precludes the use of very large pages because memory mapped I/O, including devices using fixed addresses within the "ISA hole", must be emulated by the hypervisor using permission bits, which can only be set on a page granularity.

2.3 FreeBSD Reservation System

The ability to dynamically select the optimal page size is important because incorrect page size selection (either too big or too small) can have significant overhead in execution time. For example, in 2009, Google identified "dynamic huge pages, which can be assembled and broken down on demand" as a key objective for their Linux kernel work [8]. Allocating a large page when only a small part of it will be used has the obvious effect of wasting physical memory. However, a more important side-effect is increased I/O traffic. Since the OS can only track application memory use on a page granularity, it must write back an entire page to disk when a memory mapped file is modified. The overhead of writing back two megabytes to disk when only a single byte is changed can overwhelm the performance gained by using large pages [19].

One approach to dynamically selecting page size is a reservation-based physical memory allocator, as first suggested by Talluri and Hill [24]. Navarro, *et al.* ([19]) extended this idea to a practical memory allocation system and implemented it under FreeBSD. For example, this extended design reclaims partially filled reservations, allowing the empty pages to be used by other processes.

When a process allocates virtual memory, through `mmap()` or indirectly through `malloc()`, the operating system does not immediately allocate any physical memory. Instead, it creates a bookkeeping entry that describes the allocation and what virtual addresses are being used. Physical memory is only allocated and the page table updated when the program first tries to access that virtual memory. Since the page table does not have mappings for those virtual addresses until then, that first access causes a page fault. Then, the fault handler locates the bookkeeping entry associated with the faulting virtual address and actually allocates memory.

The fault handler uses that bookkeeping entry to predict if the virtual memory space used is likely to be contiguous and larger than a large page. For example, a memory mapped 5KB file will not use an entire 2MB page, so it will not be placed in a reservation. If the handler decides that a superpage is appropriate, it will reserve an entire large physical page, and allocate the small page within it that is virtually and physically aligned to the faulting virtual address. When the program faults again on the same region of virtual memory, the fault handler will recognize that the address is part of a reservation, and it will again return the virtually and physically aligned small page within that reservation. When all small pages in the reservation are allocated, the mapping is promoted, that is, the small page mappings are replaced by a single large page mapping. However, under memory pressure, this reservation may be broken down if it is never fully utilized. The virtual memory system can then return the unused pages back to the free pool of small pages.

3. PAGE TABLE SPECULATION

The SpecTLB is a translation-lookaside buffer that tracks partially filled large-page *reservations* instead of large pages themselves. On a TLB miss, the SpecTLB is consulted to see if the virtual page that missed in the TLB may be part of a large page reservation. If so, the physical address of the missing page can be interpolated between the starting and ending physical addresses of the reservation using the small page's position within the large reservation.

The primary difference between the SpecTLB and a traditional TLB is that mappings generated by the SpecTLB are predictions; they are not guaranteed to be correct. An entry in the SpecTLB indicates that the operating system may have placed small pages within a particular physical reservation in the past, but it does not guarantee that any particular virtual page was placed in the reservation or even that the page is valid. This distinction means that a TLB miss that does hit in the SpecTLB must still be validated against the underlying page table. However, the interpolated translation can be used for speculative execution while the page walk itself continues in parallel. If the interpolated translation matches the result from the page walk, the speculative work can be committed and execution continues. If the results differ, the speculative work is not committed and execution restarts from the first incorrect prediction. While the SpecTLB does not reduce the overall memory bandwidth required by the MMU, it does reduce the latency penalty from TLB misses by removing the page walk from the critical path of execution.

This relaxed requirement of correctness allows two different variants of the SpecTLB to be built: one that requires software support and one that does not. We describe both in this section.

3.1 Explicit page table marking

The SpecTLB maintains a set of large page reservations it believes the operating system is assigning to the current process. This set is maintained by monitoring which small page page table entries are marked by the operating system as being part of a large page reservation. These marks are implemented using one of the currently unused bits in bottom level (L1) page table entries in x64-64. They do not effect how these entries are translated; they are still standard small pages and they are only used to maintain the contents of the SpecTLB.

For purposes of describing the operation of the SpecTLB, we will use a similar simplified address representation as described in Section 2.1. Virtual addresses are split up into four indices and an offset: *e.g.*, (0b9, 00c, 0ae, 0c2, a2e). Physical addresses are similarly divided into the nine-bit parts of a physical page number and a page offset: *e.g.*, {0ac, 0c2, a2e}. For simplicity, only eighteen bits (two parts) of the 40-bit physical page number are used in the following description.

The SpecTLB associates the bits of the virtual address corresponding to a 2MB virtual page number with the physical address of the reservation Note that only the upper bits of the translation, those that select a particular large physical or virtual page are stored. A SpecTLB entry for a particular address is effectively whatever the standard TLB entry would be if that address were part of a large page.

On a subsequent TLB miss, the SpecTLB is searched like a normal TLB. If a subsequent TLB miss is for virtual address (0b9, 00c, 0ae, 0c3, 001), the newly added SpecTLB entry will match. This means that the new virtual address is part of that same virtual large page and that the operating system likely placed the corresponding physical page within the physical reservation. The speculative translation concatenates the stored physical page number of the matching reservation with the large page offset from the virtual address, yielding {0ac, 0c3, 001}. This speculative translation is provided to the core, which uses it to speculatively execute subsequent instructions.

Of course, this translation may not be valid. The underlying reservation may have been broken down by the operating system or the virtual address may not even be valid. Therefore, while the processor can use this translation, it must do so speculatively. While program execution continues, the standard x86 page walk happens concurrently. If the predicted physical address matches the actual address, the speculative work may be committed. Otherwise, execution must roll back to the point of the misprediction. This rollback capability is important because it is useful to be able to make predictions even on reservations that were previously broken down. In this case, the pages that were allocated when the reservation still existed will still be stored in predictable physical page numbers.

Since SpecTLB translations are always confirmed against the underlying architectural page table, implementations can be more lazy about SpecTLB invalidation than they can be with a TLB. Stale entries do not lead to incorrect operation, as they do in a TLB. Invalid translations generated by stale entries will be corrected automatically. Our implementation only invalidates the SpecTLB on a context switch, though even this is not strictly necessary.

The SpecTLB can support multiple reservation sizes by maintaining a structure for both 2MB and 1GB reservations. The contents of the 1GB structure is maintained by monitoring the reservation bit of 2MB pages while the contents of the 2MB structure is maintained by monitoring the bit in 4KB pages.

3.2 Heuristic reservation detection

The above description of the SpecTLB requires explicit marking of the page table. This requires both a modification of the x86 page table architecture and the operating system itself. However, it is possible to build a variant of the SpecTLB that detects reservations and requires no modification to existing system software that supports large page reservations.

To allow a region of memory to be promoted to a large page, small pages must be aligned within their large page reservation. In the example above, virtual address (0b9, 00c, 0ae, 0c2, a2e) is mapped to physical address {0ac, 0c2, a2e}. The virtual page's offset within a 2MB virtual page is 0c2, equal to the physical page's offset within a 2MB physical page. Specifically, `(virtual address)[20:12] == (physical address) [20:12]`. This equality can be used as a heuristic to signal that the operating system has placed this page within a reservation. While it has a false positive rate of 1:512 (assuming 4K pages that are not part of a reservation are placed randomly), it has a zero false negative rate.

The heuristic based SpecTLB uses this detection scheme to maintain its contents. Translations are inserted when `(virtual address)[20:12] == (physical address) [20:12]` and entries are removed when they lead to false predictions.

3.3 Memory side-effects

TLB entries in modern processors store the effective memory type of the memory region, enabling the processor to know that the memory region has special properties, such as being uncacheable and/or being used for I/O. Care must be taken when memory accesses are made to such regions, as they can have irreversible side effects.

With the use of a SpecTLB, it is important to prevent memory references made using speculative translations from causing irreversible side effects before the translation has been validated. This may occur when a reservation is broken down and pages are reclaimed from that reservation. The reclaimed pages could be retasked for any purpose, including for I/O. With heuristic reservation detection, it is possible to improperly speculate on these reclaimed regions. Therefore, hardware mechanisms may be necessary to prevent speculative I/O accesses.

Tagging requests throughout the memory system as being speculative should be sufficient to ensure safety. Current systems do not need such a tag because speculative accesses are always made to a page that is already known to be safe. While the exact architectural implementation is beyond the scope of this paper, this tag could be used to prevent speculative accesses from reaching an I/O controller or writing to memory. Additionally, this will allow new speculative execution from other systems such as the prefetcher or branch predictor to make similar memory accesses without going through the TLB.

Alternatively, the explicit marking SpecTLB can avoid these problems entirely by ensuring that uncachable and I/O memory is never mapped into 2MB regions of memory that also contain data pages with the reservation bit set. In effect, the reservation bit becomes a reservation-safe bit, indicating that speculative accesses to nearby pages are safe.

3.4 Virtualization

The extra overhead of nested paging (see Section 2.1) magnifies the possible benefit of the SpecTLB. While I/O emulation requirements preclude the use of 1GB pages in the hypervisor page table (see Section 2.2), it is still possible to use 2MB pages within a 1GB *reservation*. This means that even a small SpecTLB will be able to keep track of all the reservations used by the hypervisor. Thus, it can speculatively translate any guest physical address into a host physical address.

In the nested page walk, if there is a NTLB miss, the SpecTLB provides a speculative translation that allows the next guest page table entry to be read immediately. Similarly, the SpecTLB can provide a speculative translation for the fifth hypervisor translation (guest physical to host physical translation for the underlying data) that is never provided by the NTLB. We call this process "horizontal speculation" (Figure 5). In our design, the nested page walk can defer all accesses to the hypervisor page table until after the spec-

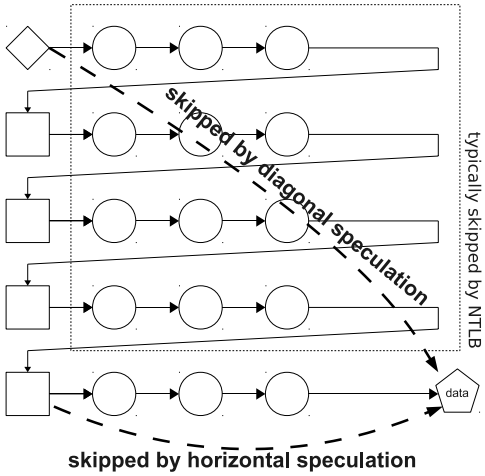

Figure 5: Horizontal and diagonal speculation.

ulative page walk. These accesses are only needed to verify the SpecTLB translations and can occur in parallel with other speculative work. If any of the speculative translations end up being incorrect (primarily due to a region of memory being an emulated I/O device), the speculative work is canceled and the page walk starts over. In the common case, however, all the additional memory and cache accesses to the hypervisor page table are removed from the critical path of execution. This leaves five SpecTLB or NTLB accesses as the only additional unhidden latency of the nested page walk above the native page walk.

The largest possible performance benefit from the SpecTLB is seen when large page reservations are used in both the hypervisor and the guest. In this case, the SpecTLB is modified to associate the guest virtual address of a guest reservation directly to a host physical address. This is analogous to how the normal TLB functions. On a TLB miss, the SpecTLB can immediately provide a speculative host physical address while the entire nested page walk occurs in parallel. We call this process *diagonal speculation* because both the guest and hypervisor page table entries are skipped.

If a speculative diagonal translation is not available for a particular address, the CPU can perform horizontal speculation to hide the latency from the nested portions of the walk.

4. METHODOLOGY

The actual execution time benefit of speculative execution depends greatly on the parallelism of the underlying microarchitecture. Instead of evaluating the SpecTLB against any particular current-generation machine, we used a microarchitecture-independent simulation that shows the memory-level parallelism that we expose. As memory systems continue to become more and more parallel, the SpecTLB will have an increasingly significant impact on the overhead of address translation.

4.1 Full-system simulator

The AMD SimNow [4] full-system simulator was used to run various benchmarks under an unmodified version of FreeBSD 8.0-Release for x86-64. A custom analyzer plugin to SimNow records each virtual memory access made by the simulated system along with the associated physical address and page size. This trace includes all memory loads and stores made by the guest operating system and processes, but it does not include instruction or page table loads. Instruction loads are not modeled by SimNow, so they are not included in this study. Page table loads are simulated by the SpecTLB simulator. Context switches are monitored by tracking the value of the CR3 register. These trigger TLB invalidations in our simulator. Finally, our custom analyzer plugin to SimNow counts the total number of instructions executed during the trace.

4.2 Benchmarks

Traces were collected for several popular benchmarks, including the SPEC CPU2006 suite [11], SPECjbb2005 [23], the NASA Advanced Computing Parallel Benchmarks (NAS) suite [2], benchw [7] and an ad-hoc Python microbenchmark. However, not all of the benchmarks in the SPEC CFP2006 suite could be compiled with the standard tool chain in FreeBSD 8.0, so soplex, calculix and wrf are not included in this study. SPECjbb2005 was run on one warehouse, and Sweep3d was run on a 150x150x150 grid. The NAS benchmarks are configured to use a class-C problem size for all benchmarks but dc, which was too large to run within our simulator. The class-B size was used for this benchmark. The benchw benchmark functions similarly to TPC-H. Specifically, this benchmark executes a `JOIN` between two tables of approximately one gigabyte in size under PostgreSQL 8.4 using default tunings. The Python benchmark is a custom microbenchmark that initializes a large array of long integers.

4.3 SpecTLB Simulation

A custom memory system simulator was built to simulate the SpecTLB design with heuristic reservation detection, both in nested and native execution. This simulator models an MMU that closely resembles that of the Family 10h AMD Opteron [5]. It includes a 64-entry, fully-associative L1 TLB with random replacement, a 512-entry, 4-way set associative L2 TLB with LRU replacement for small pages, a 128-entry L2 TLB for large pages, a 16-entry Nested TLB, and a 24-entry Page Walk Cache.

A 1MB L2 cache was included in the model, simulated using the Dinero IV cache simulator [9]. Both application data accesses and MMU page table accesses are simulated using a shared L2 cache model. Instruction loads are not instrumented and are not included in this study. The L1 cache was not simulated, since the page walk hardware does not use it on the Opteron.

The simulator reads the memory trace, running each virtual address through the TLB and data cache model. On a TLB miss, the SpecTLB is searched to see if a possible matching reservation can be found. If so, a speculative physical page number is generated. If this page number matches the actual page number stored in the trace, the speculation is correct. The MMU simulator then performs the native or nested page walk and keeps track of memory accesses made to the page table.

Since explicit reservation marking is not yet included in any operating system, the heuristic based SpecTLB is implemented. As a baseline, the number of reservations tracked by the SpecTLB is set at 24, the number of entries in the AMD Page Walk Cache. This size is varied from one to forty-eight.

We included a model of the distance-based TLB prefetcher described by Kandiraju and Sivasubramaniam [12] in our simulator for comparison. Our model was configured to use 32 entries for both the distance and prefetch buffers, which is the size explored by Kandiraju and Sivasubramaniam that most closely matched the size of our simulated SpecTLB. We had to modify the original design of the TLB prefetcher to support multiple page sizes, which was not addressed by the original paper. We chose a simple approach, replicating the entire prefetcher for each page size. After each TLB miss which is filled by a page of a particular size, that size TLB prefetcher attempts to fetch the next page of that size that

will be used. This is an optimistic design since it greatly increases the amount of history tracked by the overall system. A more practical implementation would combine prefetch history from different page sizes which could lead to conflict between entries.

For nested operation, we assume that the 1GB SpecTLB is large enough to hold all reservations in use by the hypervisor and is never flushed.

5. SIMULATION RESULTS

This section evaluates the SpecTLB using the aforementioned applications. The ability to accurately predict translations and hide MMU-related cache and DRAM accesses are examined. These results show that even a small SpecTLB is effective at accurately predicting the majority of translations for many benchmarks and enabling parallelization of much of the overhead of address translation.

5.1 TLB miss parallelization

When a translation is able to be accurately predicted, all memory and page walk cache accesses required to serve the TLB miss are removed from the critical path of execution. Instead, the predicted translation can be used immediately to enable the memory reference to proceed in parallel with the page walk.

Table 2 shows the potential TLB miss parallelization benefits of the SpecTLB. The first columns of the table show the average number of instructions between TLB misses as well as the average number of memory accesses and DRAM accesses made by the program between TLB misses. Some applications, like povray, incur a TLB miss for every 1.3 DRAM accesses. Since TLB miss handling is memory intensive, we report the frequency of TLB misses in relation to memory system activity.

A speculative translation is only attempted when a matching reservation is found. The rate at which this happens varies significantly by workload. As shown in Table 2 in the attempts per walk column, some benchmarks, such as `mcf` find a reservation and attempt a speculative translation over 99% of the time. Others, such as `libquantum` never attempt a speculative translation. Over half the benchmarks have speculation rates greater than 62%. The ability to speculatively translate addresses depends on the locality in which a program uses its address space and if it is used in a fragmented manner, leading to partially filled reservations that were never able to be promoted to a large page.

Even using heuristic-based reservation detection, most benchmarks have a prediction accuracy above 99%. While some benchmarks have particularly low prediction accuracies (less than 35% in the case of dc.B), speculation happens comparatively rarely in these workloads. In this case, a speculative prediction is only attempted for 8.3% of TLB misses. The low rate of speculation here is likely due to its small working set, which would lead to comparatively few reservations being made. An explicit reservation marking based SpecTLB would eliminate these mispredictions.

When addresses are accurately predicted, the page table walk can be overlapped with speculative execution using the predicted address. A high prediction rate combined with high accuracy allows useful work to be performed in parallel with much of the overhead of virtual memory. This is quantified by counting the number of total L2 data cache misses made by the MMU as well as the fraction that are overlapped through successful prediction. Since DRAM accesses are so slow, this fraction should translate into a proportional speedup of overall TLB miss handling. Tested benchmarks have an average of 40% of their MMU-related DRAM accesses overlapped with speculative execution by successful prediction. The benchmark which sees a particularly large number of TLB misses per program DRAM access, `mcf`, has 96% of its MMU-related DRAM accesses overlapped.

5.2 Overlap opportunity

Superscalar microprocessors can speculatively execute only a limited number of instructions as constrained by the size of their reorder buffer. If the instructions after a TLB miss complete quickly, the reorder buffer may fill before the parallelized page walk completes. The processor then has to stall until the page walk is complete, as it does without the SpecTLB. However, the load or store which triggered the TLB miss is by definition to an address not recently accessed, or else it would have hit in the TLB. Therefore, it is less likely to be cached and may be a long-latency operation itself. The high-latency TLB miss can be overlapped with the high-latency load or store before the reorder buffer starts to fill.

To examine this, our simulator was instrumented to determine the fraction of memory operations that cause TLB misses *and* a read miss in the L2 cache for the data itself. The per-benchmark average of this fraction is 40% and is as high as 93% (`sphinx3`). These results are presented for all workloads in Table 2. These long-latency read operations present significant latency in a single instruction that can be executed while the parallelized page walk completes.

5.3 Misprediction Cost

On a misprediction, unnecessary work will be performed, which may lead to increased resource contention and power consumption. However, since the SpecTLB does not provide a speculative translation when a request is not part of a tracked reservation, misprediction rates are extremely low (Table 2). Therefore, even when the SpecTLB is unable to provide predicted translations, the penalty from its use should be low. Misprediction rates are below 1% for over half the benchmarks tested.

The mispredictions that are present are either for pages that are not present, that were allocated after a reservation was broken down or were made using a incorrect SpecTLB entry. Incorrect SpecTLB entries are generated when a page is aligned by chance, not because it was part of a reservation. Using explicit marking avoids these incorrect entries.

5.4 Sizing and replacement considerations

Even a relatively small SpecTLB is effective. Each entry covers 512 small pages, or 2MB of virtual address space, so even a small device is effective. For SPECjbb2005, the benchmark that requires the most entries, reducing device size from 24 to 12 entries only reduces successful speculation rate by 3%. Other benchmarks are impacted even less. Additionally, we simulated both LRU and random replacement policies and found both to be equally effective.

5.5 Comparison to TLB prefetching

Both the SpecTLB and TLB prefetching are speculative techniques for hiding the latency of TLB misses. The SpecTLB uses large-page reservations to predict the translation for the current TLB miss, and the TLB prefetcher uses access patterns to predict future TLB misses. The SpecTLB does speculative work in the form of the instructions that are executed while a speculative translation is confirmed. The TLB prefetcher does speculative work in the form of page table walks to prefetch page table entries.

When simulated with the parameters described in Section 4, the SpecTLB handles a larger fraction of TLB misses than the TLB prefetcher does in all but seven of the benchmarks tested (dc.B, ep.C, bzip2, GemsFDTD, lbm, libquantum and milc). Of these, the TLB prefetcher handles less than 20% of TLB misses for all

benchmarks other than ep.C, bzip2 and milc. These three benchmarks are well suited for a TLB prefetcher because they have a very regular access pattern. However, this regular access pattern also means that TLB miss rates are comparatively low, and thus accelerating address translation will have a proportional small impact on execution time.

Finally, simulation shows that while misprediction rates are very low for the SpecTLB (less than 1% for over half the benchmarks), the TLB prefetcher must perform a large number of unnecessary page walks to achieve a reasonable hit rate. For example, with a TLB prefetcher, the MMU performs 1.118 page walks per TLB miss for PostgreSQL, an application which has a relatively high TLB miss rate. Over half the benchmarks tested require 1.3 page walks per TLB miss.

While TLB prefetching does not rely on any specific operating system behavior, it does require that applications access memory in predictable patterns. As a result, the SpecTLB handles a higher fraction of TLB misses for nearly all applications tested that have significant TLB miss rates.

6. NESTED PAGING SUPPORT

Virtualization compounds the overhead of virtual memory by forcing every guest physical address to be translated into a host physical address. While the use of large pages in the guest reduces this overhead, recent work has shown that the overall execution time penalty as compared to native execution is still between 7-14% [5]. In this section, we argue that *diagonal* and *horizontal speculation* (see Section 3.4) can significantly reduce this overhead.

Figure 4 shows a timeline of a typical nested page walk. In the common case, the Nested TLB translates the guest physical addresses of guest page table entries to host physical addresses. Since the Nested TLB does not translate guest physical addresses of data (non-page table) pages, the bottom level nested translation (nL0) must be performed by walking the hypervisor page table.

If both the guest and hypervisor use large page reservations, diagonal speculation can allow this entire process to occur during speculative execution. The rate at which diagonal speculation is possible in nested paging is identical to the rate at which speculation is possible in native execution. These results are shown in detail in Table 2. However, when diagonal speculation fails or is not possible, horizontal speculation can allow the nested phases of the nested page walk to occur in parallel with speculative execution.

Table 1 shows the number of memory and DRAM accesses made in each phase of the nested page walk as shown in Figure 4. These results are split into those TLB misses where diagonal speculation was successful and those where it was not. When diagonal speculation is successful, all cache and DRAM accesses are overlapped with speculative execution. When diagonal speculation is not successful, horizontal speculation can overlap the memory accesses in the nested (nL4, nL3, nL2, nL1, nL0) phases with speculative work.

When the SpecTLB performs horizontal speculation, the hypervisor page walks (and the memory accesses contained in the nested phases) do not contribute to the critical-path latency of the TLB miss. The SpecTLB can immediately translate any guest physical address into a speculative host physical address. This means that all four guest page table entries can be loaded in succession, just as in the native translation, without access to the hypervisor page table. The five nested phases must be performed at some point to verify that the speculative translations were correct, but these can be done later in execution. As long as sufficient memory bandwidth is available, this hides all of the additional latency generated by using nested paging.

For example, on a TLB miss, the processor will use the SpecTLB to translate the guest physical address of the gL4 page table entry into a host physical entry. This entry is loaded, which provides the guest physical address of the gL3 page table entry. Again, the SpecTLB translates this into a host physical entry. This process repeats until the gL1 entry provides the guest physical address of the underlying data. This is translated with a fifth and final SpecTLB access, and the data is loaded. At this point, the nL4, nL3, nL2, nL1 and nL0 phases are performed in parallel with speculative execution and in any order.

7. RELATED WORK

The SpecTLB implements a very limited form of value prediction. Martin, *et al.* show that simply verifying that the value predicted and the value actually fetched match is not sufficient to guarantee correctness on shared-memory multiprocessors [14]. One process could incorrectly predict a value while another process modifies the underlying address. This could lead to a false positive verification which could be a correctness violation under some strict memory consistency models. Fortunately, the page table uses a weak ordering consistency model since the TLB must be explicitly be flushed when the page table is changed. When a TLB flush occurs for a given address space, any speculative translations not yet verified by the SpecTLB should be cancelled. This should be sufficient to guarantee correctness. Romanescu, *et al.* discuss memory consistency issues in virtual memory in more detail [20]. They show that in multithreaded applications, global TLB synchronization may be necessary to guarantee correctness. This will also be needed for the SpecTLB.

Talluri and Hill first proposed a reservation based memory allocator for using a *partial-subblock TLB* [24]. This device, like the SpecTLB, provides direct support for translating small pages within a large page reservation. However, unlike the SpecTLB, the partial-subblock TLB explicitly tracks which small pages within the reservation are valid. While this also increases TLB coverage compared to a standard TLB, the partial-sublock TLB needs a much larger structure than the SpecTLB does. Each entry must have a valid and modified bit for each small page within a large page reservation. With the 4KB and 2MB page sizes explored in this paper, 1024 bits per entry are required in a partial-subblock TLB.

This explicit tracking approach also means that the partial-subblock TLB is unable to hide access latencies that the SpecTLB can. Assuming you do not preload each subblock TLB entry (which would require accessing 512 page table entries for each TLB miss), the subblock TLB must suffer a TLB miss and access to the page table for each 4KB page upon first access to that page. In contrast, the SpecTLB also hides the TLB miss latency of the first access to each 4KB page in the region, as the mapping can be predicted immediately.

Navarro, *et al.* extended this idea into a completely automatic system for superpage management and implemented it under FreeBSD [19]. Most critically, this extension developed the reclamation of partially filled superpage reservations. Additionally, their work extended the system to support multiple page sizes and defragmentation of physical memory. While these systems are trying to create promotable reservations, as a by-product they also create the contiguity in the page table that is exploited by the SpecTLB.

Earlier approaches to supporting superpages in general-purpose operating systems, such as IRIX, Solaris, and HP-UX, relied upon the program to explicitly specify a static page size for a region of the virtual address space [10, 16, 15]. These operating systems would then allocate and map the entire superpage at once. Conse-

		nL4	gL4	nL3	gL3	nL2	gL2	nL1	gL1	nL0
No diagonal speculation	L2 accesses	0.005	0.109	0.066	0.149	0.105	0.359	0.279	1.000	1.062
	DRAM accesses	0.000	0.048	0.033	0.075	0.049	0.161	0.142	0.372	0.199
After diagonal speculation	L2 accesses	0.000	0.029	0.028	0.033	0.031	0.044	0.057	0.946	0.946
	DRAM accesses	0.000	0.028	0.028	0.028	0.029	0.031	0.033	0.197	0.046

Table 1: Memory accesses to the page table in different phases of the nested page walk for an average of all benchmarks examined. The table is split into those TLB misses where diagonal speculation was not attempted or failed and those where diagonal speculation was successful.

quently, there would never be any partially filled reservations to be exploited by the SpecTLB. However, in such systems, a program had to be conservative in its use of superpages to avoid wasted physical memory and increased I/O. In contrast, with FreeBSD's promotion-based approach, it is possible for the system to be very aggressive in speculatively allocating reservations, most of which may never be promoted, because of the low impact of breaking reservations. Thus, the SpecTLB may have many opportunities to exploit partially filled reservations in situations where the same program running under these earlier operating systems would not have used superpages.

Romer, *et al.* propose the creation of superpages by moving existing pages, previously scattered throughout physical memory, into contiguous blocks [21]. While this process may be prohibitively expensive for very large superpages, it may have more success with the architecture described here that does not require a full reservation.

Saulsbury, *et al.* propose a prefetching scheme for TLBs that preload pages based recently accessed pages [22]. Unlike the system presented in this paper, their techniques require page table modification. More recent work has proposed architecturally independent prefetching techniques based on access patterns and intercore cooperation [12, 6]. We simulated the work of Kandiraju and Sivasubramaniam [12] in Section 5.5.

Bhargava, *et al.* first described the AMD page walk cache and their extensions to it to support virtualization [5]. Barr, *et al.* explored the design space of these dedicated MMU caches and showed that they are often successful at eliminating memory accesses for upper level page table entries [3].

Alternatively, page table accesses can be accelerated by replacing the page table format with one that requires fewer accesses. In terms of space, a radix tree-based page table can be an inefficient representation for a large, sparsely-populated virtual address space. Liedtke introduced *Guarded Page Tables* to address this problem [13]. In particular, Guarded Page Tables allow for path compression. If there is only one valid path through multiple levels of the tree, then the entry prior to this path can be configured such that the page walk will skip these levels. Talluri and Hill presented an alternate form of the inverted page table, the *clustered page table*, that derives from their subblock-TLB [25]. This technique can dramatically reduce the size of an inverted page table.

Aside from the prefetchers simulated in Section 5.5, these techniques focus on reducing either the frequency of TLB misses or their cost in terms of main memory accesses. In contrast, the SpecTLB allows whatever cost a TLB miss has to be parallelized with other work. It does not prevent the page table walk, it hides its latency. Therefore, all of the hardware techniques discussed above could be combined with the SpecTLB to reduce the memory bandwidth required by the paralellized page table walks. They are complementary to, not competitive with, the SpecTLB.

8. CONCLUSIONS

With memory capacity growing faster than TLB sizes, the increasing footprint of modern applications will lead to higher and higher virtual memory overhead. While much progress has been made to accelerate TLB miss handling with various techniques such as caching and large pages, the footprint and access pattern of popular workloads is making address translation more and more difficult. We have presented a device, the SpecTLB, that effectively decouples the address translation from the allocation and permission setting of large pages. Our device exploits a reservation based physical memory allocator to deliver much of the benefit of large pages without incurring their limitations.

Operating systems that use reservation based memory allocation are currently designed to use reservations in the hope that those reservations will fill and memory will be promoted to a large page. FreeBSD is tuned to reserve memory only when there is a possibility that it will be able to promote a region to a large page. Our results show that even with these tunings, there are enough partially filled reservations that an accurate prediction can be made for over 62% of TLB misses for the majority of benchmarks tested. Moreover, we show that these predictions can enable 40% of the MMU related DRAM accesses to be performed in parallel with data access.

When compared to the TLB prefetcher, a different architectural approach to hiding the latency of TLB miss handling, we show that the SpecTLB is able to handle a higher percentage of TLB misses in virtually all applications tested that have significant TLB miss rates. Moreover, the SpecTLB is far more sensitive in when it performs speculative work. It mispredicts translations less than 1% of the time for most benchmarks while the TLB prefetcher causes an 30% increase in the number of page walks which must be done.

Finally, we show that our device can hide the increased latency from nested paging. Recent work has shown that nested paging comes with significant performance penalty. While the use of very large pages (1GB on x86-64) would greatly reduce the frequency of TLB misses, they cannot typically be used because the hypervisor needs to maintain fine-grained control over guest physical address space permissions. Speculative address translation allows the performance of large pages while maintaining fine-grained control.

Although neither Windows nor Linux currently implement a reservation-based physical memory allocator, the limitations of the current large page support in Linux are widely recognized [8]. Therefore, rather than restricting the scope of our work to developing hardware features that benefit the current versions of these operating systems, this paper has stepped outside that box. It has, instead, explored innovative hardware that complements a new approach to physical memory management that is now being used by at least one operating system, FreeBSD. Arguably, this paper's results strengthen the case for other operating systems to adopt a reservation-based approach to physical memory management, because of the potential gains from synergistic hardware. More

broadly, this paper provides a compelling example of the benefits of coordinated innovation in hardware and software.

9. REFERENCES

[1] Advanced Micro Devices. *AMD x86-64 Architecture Programmer's Manual, Volume 2*, 2002.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[3] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: Skip, don't walk (the page table). In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, New York, NY, USA, 2010. ACM.

[4] R. Bedicheck. SimNow: Fast platform simulation purely in software. In *Hot Chips 16*, 2004.

[5] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, New York, NY, USA, 2008. ACM.

[6] A. Bhattacharjee and M. Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2010. ACM.

[7] J. Corbet. benchw. http://benchw.sourceforge.net/.

[8] J. Corbet. KS2009: How Google uses Linux. http://lwn.net/Articles/357658/.

[9] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator, 1998.

[10] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *In Proceedings of the USENIX Conference. USENIX*, pages 91–104, 1998.

[11] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[12] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[13] J. Liedtke. Address space sparsity and fine granularity. In *EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop*, pages 78–81, New York, NY, USA, 1994. ACM.

[14] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 328–337, Washington, DC, USA, 2001. IEEE Computer Society.

[15] C. Mather, I. Subramanian, I. Subramanian, C. Mather, K. Peterson, K. Peterson, B. Raghunath, and B. Raghunath. Implementation of Multiple Pagesize Support in HP-UX, 1998.

[16] J. Mauro and R. McDougall. *Solaris Internals*. Sun Microsystems Press, 2000.

[17] C. McCurdy, A. L. Cox, and J. Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *ISPASS '08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 95–104, Washington, DC, USA, 2008. IEEE Computer Society.

[18] J. C. Mogul. SIGOPS to SIGARCH: Now it's our turn to push you around. In *OSDI 2010*, Berkeley, CA, USA, 2010. USENIX Association.

[19] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, 2002.

[20] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 323–334, New York, NY, USA, 2010. ACM.

[21] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 176–187, New York, NY, USA, 1995. ACM.

[22] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based tlb preloading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 117–127, New York, NY, USA, 2000. ACM.

[23] Standard Performance Evaluation Corporation. The SPEC JBB2005 Benchmark, 2005.

[24] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[25] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 184–200, New York, NY, USA, 1995. ACM.

[26] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48 – 56, May 2005.

name	TLB miss rate			Speculations		Prediction	DRAM Accesses Overlapped		data cache misses	TLB Prefetcher	
	ins/miss	mem/miss	dram/miss	total attempts	attempts/walk	accuracy	per walk	fraction of total	after TLB misses	accuracy	overhead
bzip2	950	2342	250.0	5393	0.293	0.998	0.040	0.235	0.700	0.978	1.890
gcc	8020	3578	235.4	10516	0.852	0.988	0.288	0.664	0.831	0.330	1.568
mcf	210	87	12.6	502933	0.992	1.000	0.143	0.956	0.781	0.051	1.065
gobmk	8722	3326	22.4	3648	0.279	0.994	0.091	0.338	0.423	0.261	1.320
hmmer	16145	8908	58.1	4713	0.903	0.996	0.169	0.703	0.208	0.712	1.366
sjeng	18433	6522	10.4	2561	0.383	0.985	0.121	0.393	0.556	0.096	1.091
libquantum	1275748	224895	40091.7	0	0.000	0.000	0.000	0.000	0.806	0.024	1.076
h264ref	18605	9171	19.3	3878	0.785	0.999	0.096	0.475	0.295	0.506	1.634
omnetpp	5410	1407	33.3	22875	0.754	0.994	0.100	0.602	0.676	0.325	1.465
astar	51888	21769	52.5	1287	0.625	0.981	0.092	0.317	0.067	0.350	1.437
bwaves	8763	5260	78.6	7918	0.963	0.999	0.141	0.720	0.084	0.850	1.391
gamess	40429	18410	1.5	1450	0.590	1.000	0.013	0.304	0.057	0.032	1.178
milc	18490	8048	192.4	4525	0.828	0.999	0.076	0.280	0.818	0.862	1.723
zeusmp	42484	8018	824.2	4570	0.952	1.000	0.131	0.521	0.746	0.924	1.335
gromacs	25142	10828	27.7	3434	0.857	0.992	0.142	0.603	0.314	0.148	1.318
cactus	16674	12433	105.9	3010	0.846	1.000	0.258	0.609	0.508	0.628	1.504
leslie3d	17028	6727	254.1	5924	0.931	1.000	0.209	0.706	0.630	0.707	1.589
namd	44411	15728	16.1	1437	0.518	0.990	0.051	0.311	0.261	0.067	1.229
deal	32357	19022	4.4	1134	0.472	0.984	0.016	0.260	0.121	0.168	1.240
sphinx3	3740	1149	55.7	31090	0.833	0.995	0.179	0.700	0.934	0.393	1.339
calculix	72338	24671	4.6	420	0.234	0.933	0.046	0.336	0.249	0.022	1.169
Gems	51135	19572	1.5	50	0.023	0.000	0.000	0.000	0.047	0.012	1.046
tonto	33002	16266	2.1	1123	0.412	0.978	0.008	0.196	0.060	0.065	1.209
lbm	666336	324572	20150.3	2	0.015	1.000	0.061	0.027	0.439	0.068	1.114
povray	61836	31687	1.3	528	0.375	1.000	0.016	0.267	0.071	0.001	1.050
bt.C	109448	45809	54.4	815	0.815	0.996	0.108	0.434	0.084	0.715	1.409
cg.C	33512	4776	26.4	152	0.019	0.928	0.013	0.013	0.892	0.013	1.044
dc.B	43508	21323	23.6	170	0.083	0.353	0.011	0.073	0.067	0.190	1.107
ep.C	23927	7633	77.3	79	0.014	0.962	0.004	0.023	0.299	0.897	1.899
is.C	140525	35884	53.1	860	0.721	0.995	0.103	0.346	0.078	0.617	1.565
lu.C	103736	41975	52.4	796	0.743	0.996	0.104	0.335	0.083	0.649	1.562
sp.C	18346	7709	20.4	5664	0.965	0.999	0.102	0.720	0.500	0.883	1.789
ua.C	117350	38486	48.5	470	0.415	0.994	0.068	0.193	0.236	0.117	1.199
PostgreSQL	2450	1354	13.4	29996	0.762	0.989	0.060	0.448	0.331	0.106	1.118
Python 2.6	29200	13028	65.1	3323	0.760	0.998	0.112	0.419	0.535	0.633	1.302
SPECjbb2005	5432	2506	49.9	8848	0.418	0.971	0.142	0.310	0.535	0.151	1.176

Table 2: SpecTLB simulation results for SPEC CPU2006 (integer and floating point), NAS, PostgreSQL, Python and SPECjbb2005 benchmarks. TLB miss rates are shown as the number of program instructions, memory references and DRAM accesses between TLB misses. TLB prefetcher accuracy is the fraction of TLB misses that are served from the TLB prefetch buffer and overhead is the number of page walks performed per TLB miss.