Hash, Don't Cache (the Page Table)

Idan Yaniv
Technion – Israel Institute of Technology
idanyani@cs.technion.ac.il

Dan Tsafrir
Technion – Israel Institute of Technology
dan@cs.technion.ac.il

ABSTRACT

Radix page tables as implemented in the x86-64 architecture incur a penalty of four memory references for address translation upon each TLB miss. These 4 references become 24 in virtualized setups, accounting for 5%–90% of the runtime and thus motivating chip vendors to incorporate page walk caches (PWCs). Counterintuitively, an ISCA 2010 paper found that radix page tables with PWCs are superior to hashed page tables, yielding up to 5x fewer DRAM accesses per page walk. We challenge this finding and show that it is the result of comparing against a suboptimal hashed implementation—that of the Itanium architecture. We show that, when carefully optimized, hashed page tables in fact outperform existing PWC-aided x86-64 hardware, shortening benchmark runtimes by 1%–27% and 6%–32% in bare-metal and virtualized setups, without resorting to PWCs. We further show that hashed page tables are inherently more scalable than radix designs and are better suited to accommodate the ever increasing memory size; their downside is that they make it more challenging to support such features as superpages.

> *"In all affairs it's a healthy thing now and then to hang a question mark on the things you have long taken for granted." (B. Russell)*

> *"The backbone of the scientific method involves independent validation of existing work. Validation is not a sign of mistrust—it is simply how science is done." (D. Feitelson)*

1. INTRODUCTION

Computer systems typically utilize translation lookaside buffers (TLBs) to accelerate the conversion of virtual addresses to physical addresses. The TLB is consulted upon each memory reference, and if it misses, the hardware retrieves the absent translation using the corresponding page tables and places it in the TLB. In the x86-64 architecture, the page tables are hierarchical, organizing the translations in a 4-level radix tree. Finding a missing translation in this hierarchy—a.k.a. "walking" the page tables—thus incurs an overhead of four memory references. Although the TLB often avoids this overhead (when it hits), TLB misses might still degrade performance substantially and might account for up to 50% of the application runtime [9, 10, 14, 34, 38].

Attempting to mitigate the cost of TLB misses, hardware vendors introduced special page walk caches (PWCs) to accelerate the page table walks [5, 27]. PWCs store partial translations—of prefixes of virtual addresses—thus allowing the hardware to quickly skip over upper levels in the radix tree hierarchy instead of traversing them. In the best-case scenario, when the table walker always hits the PWC, a walk requires only one memory access instead of four. SPEC CPU2006 benchmarks, for example, require 1.13 memory references per walk, on average [9].

Utilizing PWCs to shorten table walks is specifically tailored for the radix tree structure of x86 page tables. Shortening table walks can, in principle, be achieved without resorting to PWCs by replacing the radix page tables with *hashed* page tables, which yield short page walks by design [22, 29, 39]. Assuming no hash collisions, a hashed page table walk consists of only one memory reference rather than four, similarly to the best-case scenario of radix page tables with PWCs.

Arguably, as hashed page tables obviate the need for PWCs, they may constitute an appealing alternative to the commonly used radix page tables design. Counterintuitively, however, an ISCA 2010 study by Barr et al. [9] proclaimed that radix page tables are more efficient than hashed page tables:

> *"... this paper contributes to the age-old discourse concerning the relative effectiveness of different page table organizations. Generally speaking, earlier studies concluded that organizations based on hashing [...] outperformed organizations based upon radix trees for supporting large virtual address spaces [29, 39]. However, these studies did not take into account the possibility of [PWCs] caching page table entries from the higher levels of the radix tree. This paper shows that [PWC-aided] radix tables cause up to 20% fewer total memory accesses and up to 400% [=5x] fewer DRAM accesses than hash-based tables".*

Barr et al. explained that this poor performance is the result of: (1) hashed page tables being "unable to take advantage of the great locality seen in virtual address space usage" since hashing scatters the page table entries (PTEs) associated with virtually-adjacent pages, as opposed to radix tables, which tightly pack such PTEs within the same cache lines; (2) radix page tables having "a smaller page table entry size,

SIGMETRICS '16, June 14 - 18, 2016, Antibes Juan-Les-Pins, France

ISBN 978-1-4503-4266-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2896377.2901456

because [hashed page tables] must include a tag in the page table entry", making hashed PTEs consume significantly more cache lines; and (3) the presence of hash collisions, inevitably increasing the number of memory accesses required per walk to "more than one reference to follow a collision chain."

In this study, we find that, in fact, hashed page tables *can* be more performant than radix page tables. We contend that the above findings by Barr et al. do not apply to hashed page tables in general; rather, they are specific to the hashed page table implementation that the authors used for their evaluation—the Itanium architecture [19, 25]. We present three optimizations that correspondingly tackle the aforementioned three flaws in this hashed page table design: (1) utilizing a hashing scheme that maps groups of PTEs of virtually-adjacent pages into the same cache line [39]; (2) leveraging properties of hashed paging (inapplicable to radix page tables) that allow us to pack PTEs more densely and thereby fit more of them per cache line; and, consequently, (3) substantially reducing the load factor of the hash table, thus decreasing the number of collisions and lowering the page walk latency.

Our **first contribution**, therefore, is to show that the ISCA 2010 findings are not applicable to hashed page tables in general. We do so by experimentally demonstrating (1) why the Itanium design is suboptimal and (2) how to optimize it to be more performant than the radix design. Our proposed hashed page table reduces benchmark runtimes by 1%–27% compared to the 4-level radix page tables baseline with PWCs as implemented in current x86-64 hardware.

The ISCA 2010 work pertains to bare-metal (non-virtual) setups only. Here we also consider virtualized setups, because they substantially increase the page walk length and amplify the overhead caused by TLB misses. Hardware-assisted virtualization utilizes two layers of page tables, one for the host system and one for the guest virtual machine. The translation process of simultaneously walking these page tables is done in a "two-dimensional" (2D) manner, requiring 24 instead of 4 memory references [5, 6, 26]. Such 2D page walks make up 5%–90% of the total runtime when TLB miss rates are high, prompting hardware vendors to extend PWC support to virtualized setups [4, 28]. Previous work showed a 15%–38% improvement in guest performance by extending the PWCs to store entries from both the guest and the host page tables [11].

Our **second contribution** is the design, optimization, and experimental evaluation of a 2D hashed page table hierarchy suitable for virtualized setups. Our insight is that the number of memory accesses required to satisfy one TLB miss when using a d-dimensional page table is $2^d - 1$ for hashed page tables and $5^d - 1$ for (4-level) radix page tables, making hashed page tables exponentially more efficient. Thus, for a 2D paging system suitable for current virtualized setups ($d = 2$), our hashed page table allows for walks consisting of only 3 (rather than 24) memory references, assuming no hash collisions.

Interestingly, to achieve optimal performance, we find that the hashed page tables of the host and guest should be configured to impose different locality granularities on the PTEs that they house. The host should group PTEs of virtually-adjacent pages at the granularity of cache lines, whereas the guest should group such PTEs at the granularity of pages. With such granularities, a host PTE that maps (part of) the guest's hashed page table can be utilized to access a page-worth of guest PTEs associated with virtually-adjacent guest pages. This design is thus compatible with the locality of reference often observed in real workloads.

Our newly proposed 2D hashed page table design reduces benchmark runtimes by 6%–32% as compared to the 2D radix page table design of existing x86-64 hardware (which employs PWCs that support 2D table walks).

Our **third contribution** is a qualitative assessment of the scalability of the two competing page table designs. Radix page tables rely on PWCs, whose size is physically limited—equivalently to other level-one caches whose speed must be on par with that of the processor (L1, TLB-L1). As emerging workloads access larger memory regions with lower locality, their radix page tables grow larger and harder to cache. We demonstrate the performance consequences of the PWC size limit by simulating unrealistically perfect PWCs (infinite, always hit) and showing that the resulting performance improvement can approach 19% and 35% in bare-metal and virtualized setups, for some workloads. Hashed page tables, in contrast, are insensitive to the application's memory footprint or access pattern, approaching their optimal performance whenever their load factor is low enough to curb hash collisions. They are also practical and viable, whereas optimal, unlimited-size PWCs are not. Although hashed page tables are not without their drawbacks—these will be enumerated and discussed—we nonetheless conclude that the hashed page table design is more scalable than the radix page table design.

2. RADIX PAGE TABLES

2.1 Bare-Metal Address Translation

Modern computer systems typically utilize virtual memory to make application programming easier. Each process exists in its own contiguous virtual address space, so applications do not need to be aware of each other. Other benefits of virtual memory include improved security due to memory isolation, and the ability to use more memory than physically available by swapping data to/from secondary storage. Virtual memory is supported by both software and hardware. The operating system (OS) is responsible for assigning physical memory to virtual memory, and the memory management unit (MMU) carries out the address translation upon each memory access. The commonly used page size in x86 and other architectures is (2^{12} =) 4KB. Therefore, the 12 least significant bits of each address (virtual or physical) serve as the offset within the page, and the remaining bits serve as the page number. The virtual memory subsystem maps virtual page numbers (VPNs) to physical page numbers (PPNs). Per-process VPN to PPN mappings are stored in the page table of the process and are cached by the TLB.

The x86 architecture uses radix page tables, which are also called multilevel, hierarchical, and forward-mapped. As their name suggests, radix page tables store VPN to PPN mappings in a radix tree data structure. On current 64-bit architectures, the tree consists of a four-level hierarchy when standard 4KB pages are used [5, 26]. Accordingly, walking the page table in order to perform a VPN to PPN translation requires four memory references. Page sizes larger than 4KB are also supported so as to increase the memory coverage of the TLB [36]. When larger pages are used, the table walk is shortened to three or two steps, depending on the page size.

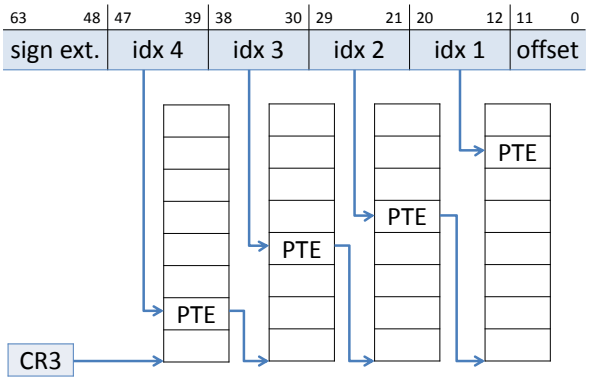

Figure 1: Bare-metal radix page table walk.

63 52	51 12	11 0
0's padding	PPN	attributes

Table 1: Radix PTE structure.

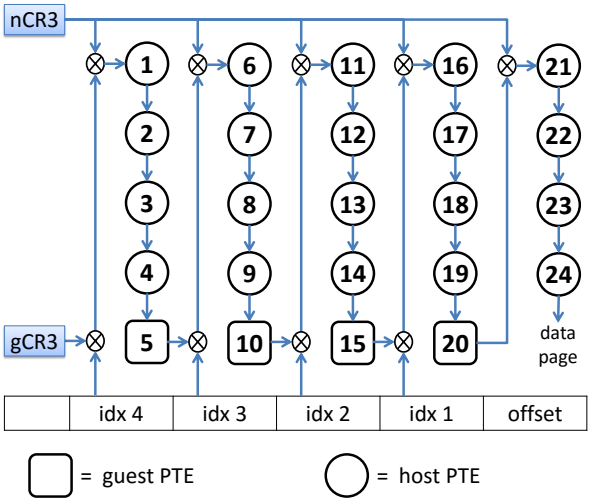

Figure 2: 2D radix page table walk.

Figure 1 depicts the radix page table structure and page walk process. Current x86-64 processors utilize 48-bit virtual addresses and no more than 52 bits for the physical addresses. With 4KB pages, the 48-bit virtual address decomposes into a 36-bit VPN and a 12-bit page offset. The 36-bit VPN further decomposes into four 9-bit indexes, such that each index selects a PTE from its corresponding level in the tree. Each PTE contains the physical address of a table in the next level. The topmost, root level table is pointed to by the CR3 register. PTEs in the lowest level contain the PPNs of the actual program pages. Since the page indexes consist of 9 bits, there are $2^9 = 512$ PTEs in each tree node, and since the tree nodes reside in 4KB pages ($= 2^{12}$ bytes), each PTE consists of $2^{12}/2^9 = 8$ bytes. PTEs encode more information than just the next PPN, in the format shown in Table 1.

Notably, PTEs are stored in the regular L1, L2 and L3 caches to accelerate the page walks [7]. In modern x86-64 processors, these caches consist of 64-byte cache lines. Radix page tables arrange the PTEs contiguously, one after the other, so each cache line encapsulates exactly eight PTEs. Lowest level PTEs that are co-located within a cache line correspond to eight consecutive pages that are contiguous in the virtual memory space. Thus, whenever the MMU accesses a PTE, its seven same-line neighboring PTEs are implicitly prefetched to the data cache hierarchy, and there is a non-negligible chance that these seven will be accessed soon due to spatial locality.

2.2 Two-Dimensional Address Translation

Machine virtualization technology allows multiple guest OSes to run on the same physical host system encapsulated within virtual machines (VMs). Guests do not control the physical host resources, and what they consider to be physical memory addresses are in fact guest physical addresses. The host creates this abstraction by introducing another dimension of address translation, called the nested dimension. In the past, this abstraction was created using "shadow page tables," a software-only technique that involves write-protecting the memory pages that the guest is using as page tables [2]. Each update of these pages triggers an exit to the host, which is thus made aware of the guest's view of its memory. The host in fact maintains another set of pages—the shadow pages—that serve as the real page table hierarchy of the guest.

Shadow page tables are difficult to implement and induce substantial overheads caused by the repeated guest-host switches upon modifications of write-protected pages [12, 13]. These drawbacks motivated chip vendors to provide hardware constructs that directly support the host in its efforts to efficiently maintain the illusion that guest OSes control the physical memory. Both AMD and Intel have implemented *nested paging* [4, 28], which supports two levels of address translation. The guest maintains a set of page tables that map between guest virtual addresses (GVAs) to guest physical addresses (GPAs). The host maintains a different set of page tables that map GPAs to host physical addresses (HPAs). The hardware is responsible for seamlessly "concatenating" the two layers of translation, by performing a *two-dimensional page walk* (as opposed to the one-dimensional page walk in bare-metal setups). Nested paging lets the guest manage its own page tables and eliminates the need for host intervention. The downside is that nested paging squares the number of memory references required to obtain a translation.

Figure 2 outlines a 2D page walk in the x86-64 architecture. The MMU references the memory hierarchy 24 times in the order given by the numbered shapes. The squares (numbered 5, 10, 15, and 20) denote the guest PTEs, and the circles (numbered 1–4, 6–9, 11–14, 16–19, and 21–24) denote the host PTEs. The four references to guest PTEs are analogous to the four references in the 1D page walk. Bare-metal page walks access PTEs by their physical addresses, whereas guests access PTEs by their GPAs. Each of these four GPAs is translated with a nested page walk to produce the HPA of the corresponding PTE. The guest page walk outcome is a fifth GPA that points to the requested data page. It requires an additional nested page walk before the page walk process is complete (numbered 21–24). Note that 2D page walks make use of two separate CR3 registers for the guest and the nested dimensions. The guest CR3 register and the virtual page number combine to start the guest page table walk. The nested CR3 register is used five times to initialize each of the five nested page table walks.

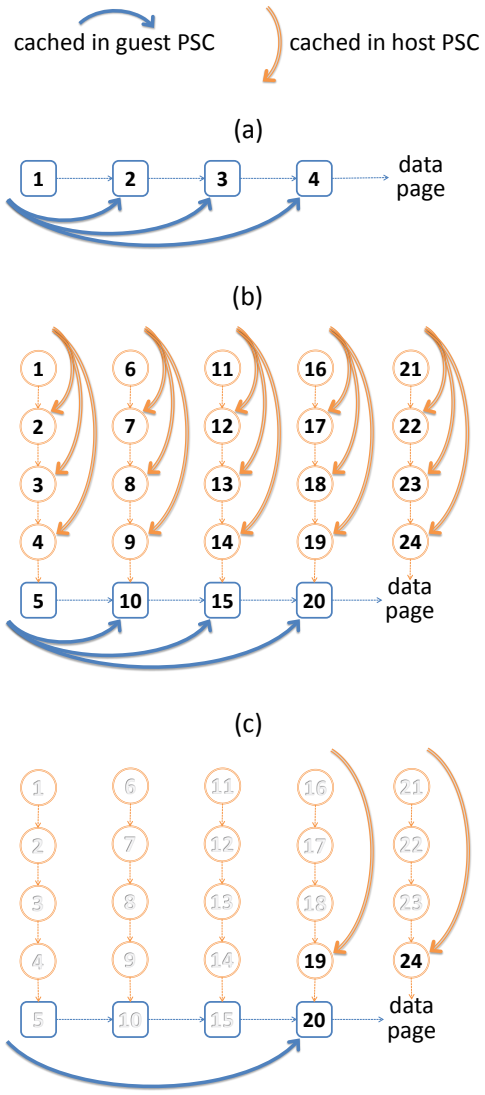

Figure 3: Page walk caches.

3. PAGE WALK CACHES

Radix page table walks require expensive memory references, and 2D page walks all the more. This motivated hardware vendors to present MMU caches, which accelerate address translation by caching parts of the page walk process. However, as with every cache, their performance degrades when the working set of partial translations is too large. In this section, we survey the existing implementations of MMU caches in bare-metal and virtualized hardware. To the best of our knowledge, Intel has not published detailed information on the x86-64 hardware that speeds up 2D page walks. We therefore introduce a simple yet not previously reported MMU cache design, which we set as the Intel baseline design for our evaluation in section 5.

3.1 Caching One-Dimensional Page Walks

MMU caches accelerate bare-metal page walks [9] by storing PTEs from the higher levels of the radix tree in small low-latency caches. Higher level PTEs are good cache candidates because they map large regions of virtual memory, so many workloads use only a small set of them with considerable reuse. PTEs from the lowest level are harder to cache; indeed, TLBs utilize hundreds of entries to effectively cache the lowest level PTEs, whereas MMU caches typically contain dozens of entries.

AMD and Intel provide different implementations of MMU caches. AMD's *page walk caches* (PWCs) tag the PTEs with their physical addresses in memory, so the PWC serves as a dedicated "L1D cache" for page walks.[1] When the MMU finds a PTE in the page walk cache, it immediately reads it and saves a reference to the memory hierarchy. Intel's *paging-structure caches* (PSCs) tag the PTEs with prefixes of the virtual page number corresponding to their location in the radix tree. That is, PTEs from the topmost level are tagged by the uppermost 9-bit index from the VPN, PTEs from the next level are tagged by two 9-bit indexes, and PTEs from the third level are tagged by three 9-bit indexes. If the MMU finds a PTE in the paging-structure cache, the page walk process can begin from that PTE. Therefore, the MMU searches for an entry with the longest matching tag, to shorten the page walk as much as possible. Figure 3(a) illustrates a radix page walk process that utilizes a paging structure cache. In the best-case scenario, when a third level PTE is found in the MMU cache, the page walk process skips over three levels in the tree and directly accesses the lowest level PTE.

3.2 Caching Two-Dimensional Page Walks

As stated earlier, 2D radix page tables lengthen the page walk to 24 memory references. Thus, extending MMU caches to speed up 2D page walks is crucial. AMD's page walk caches are physically tagged, so they can be extended to hold guest and host PTEs in the same cache according to their host physical addresses. Indeed, AMD researchers showed that caching PTEs from the guest and host dimension in the PWC brings a 15%–38% improvement in guest performance [11].

Our performance measurements on an Intel platform indicate that Intel's microarchitectures also implement hardware that accelerates 2D page walks. Alas, we do not know the exact implementation details of that hardware. We therefore offer a new, readily-implementable extension of Intel's PSCs, which boosts 2D page walks. Because Intel's paging structure caches tag PTEs with their guest virtual addresses, they can only keep guest PTEs. We propose to add another, separate PSC that stores host PTEs tagged by their host virtual addresses. Figure 3(b) depicts a 2D page walk process in a processor that utilizes two PSCs for the guest and host. The two PSCs operate in the guest and host dimensions independently and decrease the overall page walk latency. In the best-case scenario, depicted in Figure 3(c), we hit both PSCs, and the 2D page walk shortens to only three memory references. Since our proposed design is a straightforward generalization of Intel's microarchitecture, we will further assume it is the nominal Intel design.

Previous works [9,14] explored the different types of MMU caches and showed that they achieve similar performance gains. Our simulation results corroborate this finding; thus we examine only PSCs in our experiments and assume the same analysis can be applied to other kinds of MMU caches. For simplicity, we will use the terms "MMU caches," "page walk caches," and "paging structure caches" interchangeably.

[1]We remark that AMD microarchitectures caches PTEs in the L2 and L3 caches only [11].

4. HASHED PAGE TABLES

Hashed page tables [22], as their name suggests, store mappings from virtual to physical pages in a hash table. When assuming no hash collisions, only one memory access is required for address translation, regardless of the workload's memory consumption or access pattern. As radix page walks involve four memory references, hashed page tables should perform better than radix page tables, without requiring PWCs. But recent results by Barr et al. [9] surprisingly found that PWC-aided radix page tables are superior to hashed page tables, "increas[ing] the number of DRAM accesses per walk by over 400%."[2] The authors identified three drawbacks of hashed page tables underlying this poor performance: (1) hashing scatters the mappings of consecutive pages in different cache lines, so hashed page tables are unable to leverage the spatial locality seen in many workloads; (2) hashed PTEs are bigger than radix PTEs, leading to higher memory usage; and (3) hash collisions occur frequently, resulting in more memory references per page walk.

We contend that these drawbacks are specific to the Itanium design that Barr et al. used for their evaluation [19, 25]. We demonstrate that these drawbacks do not apply to hashed paging in general, and that hashed page tables can be optimized through a series of improvements that make them better performing than PWC-aided radix page tables. In this section, we first describe the basic Itanium design assessed by Barr et al. [9] (§4.1). We then point out the problems in this design (§4.2) and propose optimizations that address them (§4.3). Last, we describe how hashed page tables can be applied in virtual systems, and how they can shrink the 2D page walk cost (§4.4).

4.1 The Basic Design

The Itanium architecture utilizes a hashed page table when its virtual hash page table (VHPT) is configured to use "long format" PTEs [19, 25]. The VHPT resides in the virtual memory space, which the OS pins to the physical memory. The long format PTEs consist of 32 bytes that house full translation information, including protection keys, page sizes, and reserved bits for software use. The latter may store any value, notably an address that points to a collision chain. The Itanium architecture explicitly requires the OS's involvement on hash collisions. The hardware page walker hashes into a single table slot, raising a TLB miss fault that invokes an OS handler upon a hash collision. The OS can then resolve the collision as it pleases, e.g., by searching for the translation somewhere else in the hash table, or by using some auxiliary OS-defined data structure.

Barr et al. chose the latter alternative. They utilized "chain tables", which hold PTEs whose hash slots were already taken, resolving hash collisions with the *closed addressing* method [15]. With this design, entries that hash to the same table slot form a collision chain, which is a simple linked list where each entry points to the next. The chain's head is stored directly in the table, and the other nodes are stored in the aforementioned chain table data structure. Both the hash table and the chain table are allocated upon startup. To avoid dynamic resizing, the maximal number of PTEs should be known. When assuming no sharing of physical pages between processes, this number is bounded by the number of physical pages [29]. Thus, if the chain table size is set to this maximal number of PTEs, no dynamic resizing is needed.

[2]The primary focus of the paper by Barr et al. was, in fact, on optimizing PWCs.

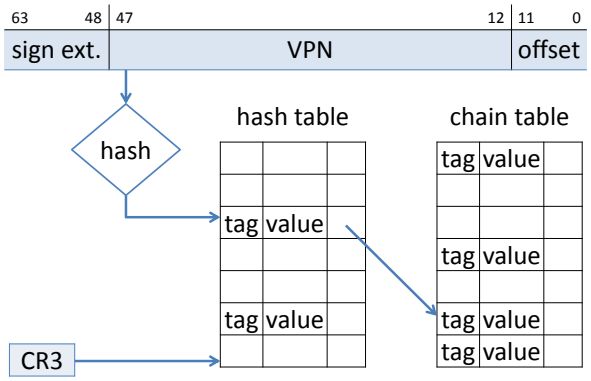

Figure 4: Hashed page table utilizing closed addressing.

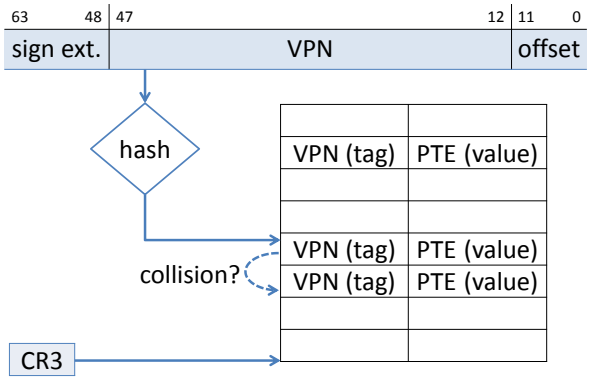

Figure 5: Hashed page table utilizing open addressing.

Figure 4 depicts the Itanium hashed page table variant that was used by Barr et al. Unlike the per-process radix page tables, there is a single, big hashed page table shared among all processes, pointed to by the CR3 register. Each slot in the hash table consists of three fields: tag (VPN), value (PTE), and a pointer to the next node in the collision chain. When translating an address, the 36-bit VPN hashes to some slot, and it is then compared to the tag stored in that slot to detect if a hash collision has occurred. If the tag mismatches, the next node in the collision chain is examined. The search continues until a match is found or the chain ends, indicating that the element is not in the table. Our simulation of the basic Itanium design reproduces the methodology of Barr et al., which does not account for the additional overheads incurred due to chain table lookups being handled by software. (Subsequent simulated versions implement the search entirely in hardware, as will be later explained.)

4.2 Flaws of the Basic Design

The basic hashed design as implemented in the Itanium architecture has several weaknesses. First, the hashed page table does not cache the PTEs effectively since it scatters the PTEs of virtually adjacent pages to non-adjacent table slots that reside in distinct cache lines. Radix page tables, on the other hand, keep the mappings of virtually adjacent pages contiguously in the physical memory, tightly packing

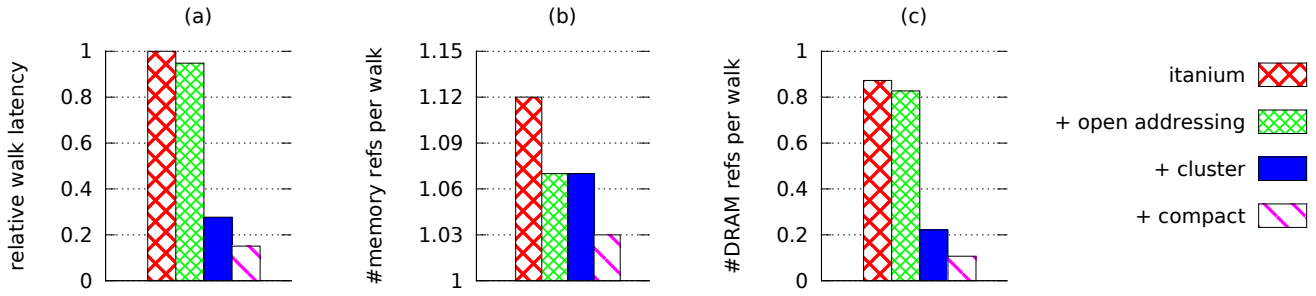

Figure 6: Optimizing the hashed page table gradually cuts down the walk latency and the number of references to the memory and DRAM. (Showing results for the 433.milc benchmark in bare-metal setup.)

their PTEs in cache lines. When one page is referenced, it is often the case that its virtually neighboring pages will soon be accessed as well, because many workloads exhibit locality of reference. Correspondingly, when one PTE is referenced, there is a greater chance that the PTEs of virtually adjacent pages will be referenced soon. Consequently, when the page walker references a radix PTE and inserts it into the cache, it also inserts the PTEs of neighboring pages, which reside in the same cache line, thereby eliminating cache misses in future page walks. Hashed designs, on the other hand, are less likely to prefetch PTEs of neighboring pages into the cache. They thus increase the number of costly DRAM references due to cache misses.

Hashed page tables also require additional memory since they must keep the tags and chain pointers to resolve potential hash collisions. In the Itanium design, the 32-bytes long PTE houses the "short" 8-byte PTE version along with its matching 36-bit VPN and an 8-byte chain pointer. A 64-byte cache line thus contains exactly two long PTE slots, in contrast to the radix page table design whereby each cache line contains eight PTEs. Hashed performance therefore degrades because the caches can hold four times fewer hashed PTEs than radix PTEs. Hashed page tables reference a larger working set of cache lines than do radix designs, so they may cause more cache misses during page walks.

The rate of hash table collisions is mainly determined by the ratio of occupied slots to the total number of slots, also called the *load factor* [15]. The hash table performance degrades as the load factor grows, since more occupied slots increase the likelihood that different elements will hash to the same slot. Importantly, for a fixed load factor, the rate of hash collisions is not affected by the workload memory footprint or access pattern, if uniform hashing is assumed [15]. The basic design sets the maximum load factor to 1/2 by allocating a hash table with twice the number of slots required to map the resident physical pages. That is, two 32-byte slots are allocated for each 4KB page in the physical memory. The hash table thus consumes 1.6% of the physical memory.

Another drawback of the Itanium hashed page table is that it requires an additional data structure—the chain table. Ideally, without hash collisions, the chain table is unnecessary since the hash table has enough space to contain all the PTEs. But hash collisions cannot be completely eliminated, so some memory must be allocated to the chain table. Radix designs are thus more space efficient, because they do not require additional data structures.

4.3 Optimizations

We now describe three optimizations aimed at remedying the flaws of the basic Itanium design. Figure 6 shows that the improvements reduce the page walk latency, as well as the number of references to the memory and the DRAM, for a specific SPEC CPU2006 benchmark (433.milc). We present the results for a single benchmark since the other benchmarks exhibit the same qualitative behavior.

Open Addressing

Our first proposal is to get rid of the wasteful chain table by switching to the *open addressing* method [15], which stores all PTEs in the hash table slots. Figure 5 illustrates the hashed page table structure and hashed page walk process in the open addressing scheme. Each slot in the hash table consists of three fields: tag (VPN), value (PTE), and an "empty" bit to mark a free slot. The chain pointers are no longer used to resolve hash collisions; when a tag mismatch occurs, the next slot in the hash table is examined. The search ends when a match is found or an empty slot is reached, indicating that the element is not found in the table.

Besides saving the memory required by the chain table, the open addressing scheme has another, more important advantage: it discards the chain pointers that lie inside PTEs. We can therefore shrink the table slots of the open addressing design to 16 bytes, as each slot contains only a 36-bit VPN and its matching 8-byte PTE. Consequently, the hashed page table size is cut by half. We can turn this space we saved to good use by lowering the load factor and shortening the hash table lookup. The open addressing design in Figure 6 sets the load factor to 1/4, without using additional memory over the basic Itanium design. Decreasing the load factor indeed lowers the average number of memory references per walk, as can be seen in Figure 6(b).

PTE Clustering

Our second proposal is to utilize *clustered page tables*—a hashed page tables variant that pack the mappings of consecutive pages in the same hash table slot [39], thereby making the page table easier to cache for workloads with a high degree of locality. The number of PTEs that occupy the same slot is called the *clustering factor*. It is normally chosen to be the largest possible such that each table slot fits into a single cache line. If the table slots are smaller than a cache line, the clustered page table does not exploit the spatial locality to its maximum. And if the table slots are bigger than a

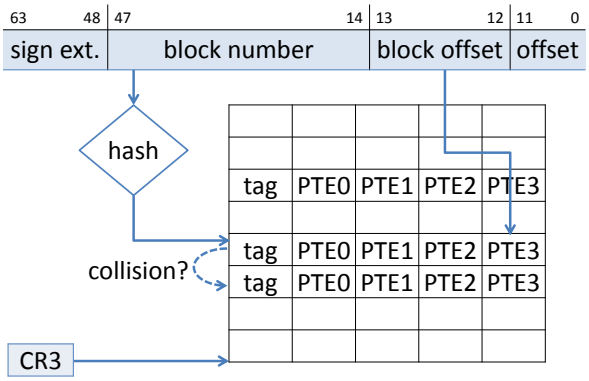

Figure 7: Clustered page table walk.

cache line, the tag (block number) and the value (PTE) may reside in different lines, so probing a single slot may require two memory references. Because hashed PTEs are 16 bytes long, and since x86-64 processors use 64-byte cache lines, the optimal clustering factor is four; radix designs still pack twice more, that is, eight PTEs per cache line.

Figure 7 depicts the clustered page table structure and the clustered page walk process. The hash table lookup follows that of hashed page tables with minor changes. The 36-bit VPN decomposes to a page block number (bits 14:47) and page block offset (bits 12:13). The block number hashes to a table slot, and the block offset indexes into the array of PTEs stored in the slot. When a mismatch occurs between the block number and the tag in the slot, the next slot in the table is examined. Each hash table slot is 64 bytes long, to hold a single tag (34-bit block number) and four values (four PTEs, 8 bytes each).

Figure 6(c) provides the average number of DRAM references per walk. We see that clustering the PTEs as was just described takes advantage of the locality seen in this workload and generates fewer DRAM references than the basic design.

PTE Compaction

We propose a new technique that further improves hashed page tables by compacting eight adjacent PTEs in a single 64-byte cache line, resulting in the spatial locality of hashed page tables similar to that of the x86-64 radix page tables. The clustered page tables, as were previously defined, cannot pack eight PTEs and a tag in a single cache line, since PTEs are 8 bytes long. But we can exploit the unused topmost bits of each PTE and store the tag in this unused space. Specifically, the x86-64 architecture reserves bits 52:63 of the PTE as available for use by the system software in long addressing mode [5]. If we group all the available bits from the eight PTEs, we have more than enough space to store the 33-bit page block number, which serves as the tag. Table 2 shows the possible layout of a hash table slot containing eight PTEs. We use 8 bytes for the tag and 7 bytes for each of the eight PTEs. This layout is valid even for architectures with 56-bit physical addresses, i.e., 64 petabytes of physical memory, which is more than enough for the foreseeable future. Note that the compaction technique can only be applied in tandem with clustered page tables, since they amortize the space overhead of the tag over many PTEs.

8 B	7 B	7 B		7 B	7 B
tag	PTE0	PTE1	...	PTE6	PTE7

Table 2: Compact cluster of PTEs.

Compacting the PTEs cuts the number of hash table slots by half, as each slot now holds eight PTEs instead of four. If we fix the load factor, compaction thus saves half the memory used by the hashed page table. Alternatively, we can decrease the load factor and reduce the hash table lookup complexity. The compact design in Figure 6 implements the compaction of eight PTEs in a cache line and sets the load factor to 1/8, bringing the average page walk cost close to a single memory reference per walk, without using additional memory over the previous designs. We see that it also exploits the spatial locality and further reduces the number of DRAM references.

Note that radix paging hardware cannot easily implement a compaction scheme that derives benefit from the unused bits in the PTEs. Current radix hardware packs eight PTEs in a cache line, and it can pack two additional PTEs if the unused bits were utilized. That is, the radix hardware can pack a maximum of 10 PTEs in a cache line and hence no more than 640 PTEs in a 4KB page. But trying to pack 640 PTEs in a page will demand a more complex calculation of the radix tree indexes, via a sequence of three divide and modulo operations, which cannot be parallelized. Current radix paging hardware exploits the fact that each page contains a power-of-two number of PTEs to immediately extract the indexes to the radix tables without having to calculate them. Specifically, each 4KB page contains exactly 512 PTEs, and the 36-bit virtual page number directly decomposes to the four 9-bit radix tree indexes, as explained in section 2.

To summarize: Itanium hashed page tables can be improved via three optimizations. First, utilizing an open, rather than closed addressing scheme shrinks the table slots and reduces the load factor. Second, clustering adjacent PTEs leverages the spatial locality seen in many workloads. Third, compacting the PTEs makes it possible to pack eight PTEs per cache line, similarly to the x86-64 radix design, thereby (i) equalizing the number of cache lines occupied by the two virtual memory implementations, as well as (ii) further decreasing the load factor and eliminating almost all hash collisions.

In section 5, we show that our optimized hashed page tables indeed reduce benchmark runtimes by 1%–27% compared to x86-64 PWC-aided radix page tables.

4.4 Two-Dimensional Hashed Page Tables

Hashed page tables reduce the page walk cost in bare-metal setups, so they may be able to shorten 2D page walks in virtualized setups as well. We now design and optimize 2D hashed page tables, which implement the guest and the host page tables with hashed schemes. Figure 8 outlines a 2D hashed page walk, which references the memory hierarchy three times per TLB miss, assuming no hash collisions. The 36-bit VPN hashes to a slot in the guest page table. This index combines with the guest CR3 register to generate the GPA of the guest PTE. We perform a nested page walk (step 1 in the figure) to translate the GPA to a HPA. Only then can we access the guest PTE (step 2 in the figure) and discover

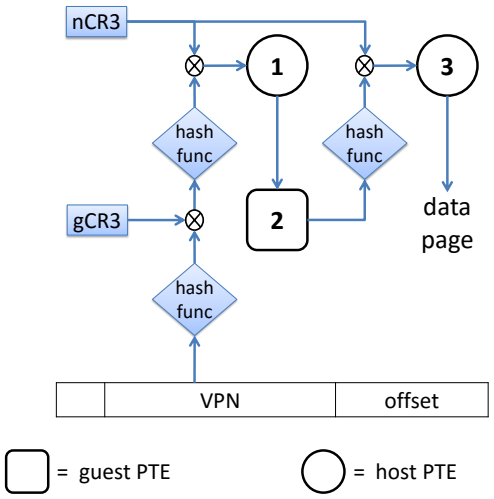

Figure 8: 2D hashed page table walk.

the GPA of the data page. We finish the page walk process with a second nested page walk (step 3 in the figure), which finds the HPA of the data page. The nested CR3 register is used two times during the page walk to initialize each of the two nested page table walks.

A 2D hashed page table is no more than two separate 1D hashed page tables for the guest and the host. We applied the three optimizations that we proposed for 1D hashed paging to both the guest and the host page tables. We initially believed that these two page tables should be tuned the same, i.e., adopting the open addressing scheme, clustering and compacting eight PTEs in a cache line, and thereby reducing the load factor to 1/8. To validate our belief, we scanned the design space of 2D hashed page tables. Surprisingly, we discovered that the guest and host page tables should be configured differently to realize the full potential of 2D hashed page tables.

Our analysis helped us to spot a subtle difference between the guest and the host page tables: the former maps the application pages, whereas the latter also maps the pages assigned to the guest page table. The host page table is able to leverage the locality of reference to the guest page table in the same way it benefits from locality in the application data. When the guest page table is scattered over many pages (in the host address space), the host page table keeps that many translations to map these pages. To minimize the number of required host translations, we should attempt to minimize the number of pages used by the guest page table. Therefore, we increase the clustering factor of the guest page table from 8 to 512, such that the mappings of each group of 512 consecutive pages will condense into a single page instead of lying in multiple pages. Figure 9 indicates that increasing the clustering factor of the guest page table from 8 to 512 reduces the page walk latency (for a specific benchmark).

Furthermore, when the host page table clusters contiguous translations in the same cache line, it can leverage locality at granularity of several pages. Assuming the host page table uses a clustering factor of 8, when the guest page table is spread over many blocks of 8 pages, the host translations consume more cache lines. To minimize the working set of cache lines that hold host page table mappings, we should

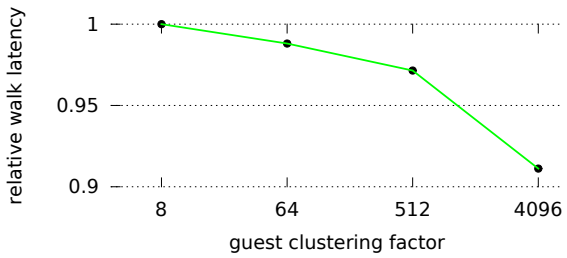

Figure 9: The walk latency of 2D hashed page table decreases as the guest clustering increases.

attempt to minimize the number of 8-page blocks used by the guest page table. Therefore, we increase the clustering factor of the guest page table even further, from 512 to 4096. Figure 9 shows that increasing the guest clustering factor even further to 4096 reduces the page walk latency.

> To summarize: In virtualized setups, the hashed paging scheme is more efficient than radix paging, because it incurs a TLB miss penalty of only 3 (rather than 24) memory references, assuming no hash collisions. To further improve their performance, we find that 2D hashed page tables should also cluster guest PTEs in granularity of several pages.

In section 5, we show that our newly proposed 2D hashed page tables reduce benchmark runtimes by 6%–32% as compared to the current x86-64 hardware (which employs PWCs that support 2D table walks).

5. HASHED VERSUS RADIX PAGING

In this section, we first describe the methodology we use to evaluate page table designs. Next, we measure the performance gap between present x86-64 hardware and perfect, unrealistic PWCs. We then compare hashed paging to radix paging; our results show that hashed page tables can bring notable performance benefits in bare-metal and virtualized setups. Finally, we enumerate the practical drawbacks and problems of hashed paging.

5.1 Methodology

Our goal is to estimate the performance of nonexistent hardware, such as the proposed hashed page tables or radix paging with perfect PWCs. To this end, we apply the de facto methodology of recent microarchitectural research concerning virtual memory [10,14,18]. First, we develop an approximate performance model, which assumes, for simplicity, that the application runtime (measured in cycles) is a linear function of the *walk cycles*, i.e., the cycles spent during page walks:

$$runtime = A * walk\_cycles + B.$$

The model parameters A, B are benchmark specific, and we find them by measuring a real system running each benchmark separately. Second, we calculate the walk cycles of new page table designs from our memory simulator, which we implemented and thoroughly tested. Last, we plug the simulation results into the linear performance model to predict the runtime of the benchmarks on these new designs.

The performance model is gauged for each benchmark individually with experiments conducted on a Linux server equipped with an Intel processor, as outlined in Table 3. Our Intel processor provides monitoring hardware that is able to measure the cycles spent during page walks (events 0x0408, x0449 in the Sandy Bridge microarchitecture [27]). The connection between the walk cycles and the application runtime is not immediate, as the walk cycles overlap with other processor activities in modern, pipelined, out-of-order CPUs. But the walk cycles certainly hint at the application performance; thus our assumption that the relationship is linear. Finding the model parameters A, B is simple given two points on the line. We obtain the first point from measurement with the default 4KB pages used, and the second point after configuring the system to use 2MB pages via the Transparent Huge Pages feature [1].

Having the performance curve, we can now apply it to evaluate several hardware designs. We develop a trace-based simulator, and generate traces dynamically with Intel's Pin binary instrumentation tool [31, 32, 34]. For each memory reference in the trace, the simulator initially consults the TLB for address translation, walks the page table hierarchy upon a TLB miss, and then performs the memory access. The simulator outputs the number of references to all the relevant hardware structures (L1/L2 TLB, L1 data cache, L2/L3 caches, DRAM, and PWCs when available). The walk cycles are then calculated via summing the access latencies to each hardware component, weighted by the access frequency. The microarchitecture parameters were taken from the closely related studies by Bhattacharjee et al. [14] and Gandhi et al. [18] as outlined in Table 3.

We note that memory simulators are able to produce accurate estimations of the walk cycles, because the page walk process is sequential, where each reference to the memory hierarchy depends on the previous one. We sample a small, representative subset of the full memory address trace following the statistical sampling technique by Wunderlich et al. [40]; we verified that our sampling simulator is accurate, yielding cycle estimates within ±5% of the full run outcome with 95% confidence. The trace-based approach, together with sampling, enables us to evaluate new architecture designs relatively quickly. Average programs run 5x–50x slower under the simulator, so we can inspect realistic workloads with big memory footprints. As Pin is limited to instrumentation in user mode only, we measured the TLB misses and the walk cycles invoked by the kernel via performance counters and verified that they are negligible.

5.2 Radix Paging with PWCs Does Not Scale

Perfect PWCs, which theoretically eliminate all misses, cut down bare-metal and virtualized page walks considerably, to one and three memory references per walk, respectively. But realistic PWCs have finite sizes that limit their performance. Our analysis shows that the gap between the performance of actual (x86-64) and ideal (infinite and always hit) PWCs can be significant in memory intensive workloads, as depicted in Figure 10 for three benchmark suites. The first suite is the canonical **SPEC CPU2006** [20], whose benchmarks typically exhibit negligible TLB miss rates and hence are largely unaffected by PWCs. We therefore limit our analysis to only those benchmarks that are sensitive to PWC misses, enjoying a performance improvement of at least 1% when PWCs are perfect—these are mcf, cactusADM, and xalancbmk. The performance of the remaining 28 SPEC CPU2006 benchmarks neither improves nor degrades when employing perfect PWCs (or our optimized hashed design). The second benchmark suite is **Graph500** [8, 35], which ships with several implementations and a scalable input generator; we use the shared memory (OpenMP) version with eight threads, and we test three different input sizes: 4GB, 8GB, and 16GB. The third suite is **GUPS** [30, 33], which likely approximates an upper bound on the performance improvement that perfect PWCs offer, randomly accessing a large in-memory table; we use three table sizes: 2GB, 8GB, and 32GB.

bare-metal system		
processor	dual-socket Intel Xeon E5-2420 (SandyBridge), 6 cores/socket, 2 threads/core, 1.90 GHz	
memory hierarchy	component	latency [cycles]
	64 KB L1 data cache (per thread)	4
	64 KB L1 inst. cache (per thread)	not simulated
	512 KB L2 cache (per core)	12
	15 MB L3 cache (per chip)	30
	96 GB DDR2 SDRAM	100
TLB	64 entries L1 data TLB 128 entries L1 instruction TLB (not simulated) 512 entries L2 TLB all TLBs are 4-way associative	
PSC	2 entries PML4 cache, 4 entries PDP cache 32 entries PDE cache, 4-way associative all caches have 2 cycles access latency	
operating system	Ubuntu 14.04.2 LTS, kernel version 3.13.0-55	
fully virtualized system		
host	QEMU emulator version 2.0.0 with KVM support	
PSCs	separate PSCs for the guest and the host, each of the same size as the bare-metal PSC	
operating system	Ubuntu 14.04.2 LTS, kernel version 3.13.0-55 for both the guest and the host	

Table 3: Details of the experimental platform; the microarchitectural parameters were also used in the simulator.

Figure 10 shows that PWC misses degrade the performance by up to 19% and 35% in bare-metal and virtualized setups. When focusing on Graph500 and GUPS, we see that the effectiveness of the PWCs monotonically drops due to (1) bigger memory footprints (compare benchmarks in the same suite) and (2) lower locality of reference (compare GUPS benchmarks to Graph500 benchmarks with similar/smaller size).

Figure 10 further demonstrates that the gap between real and ideal PWCs is larger in the virtualized case, because 2D radix page walks are longer and harder to cache. We generalize this observation and qualitatively show that radix paging scales poorly as more layers of virtualization are added. Specifically, we prove that the *page walk length* (number of memory references required to obtain a translation) grows exponentially with the number of virtualization layers. For a nested setup with d virtualization layers and a page walk length of L in each layer, we show that the *overall* page walk length is $(L+1)^d - 1$. Consequently, the overall page walk length in a d-dimensional page table is $5^d - 1$ for radix paging, whereas it is $2^d - 1$ for hashed paging, assuming no hash collisions. Thus, 2D radix page walks are exponentially longer, and the effectiveness of the PWC drops as it struggles to cache these longer walks.

The formula for the d-dimensional overall page walk length is proved by induction. The base case ($d = 1$) is trivial. The inductive step assumes (1) a d-dimensional nested virtualiza-

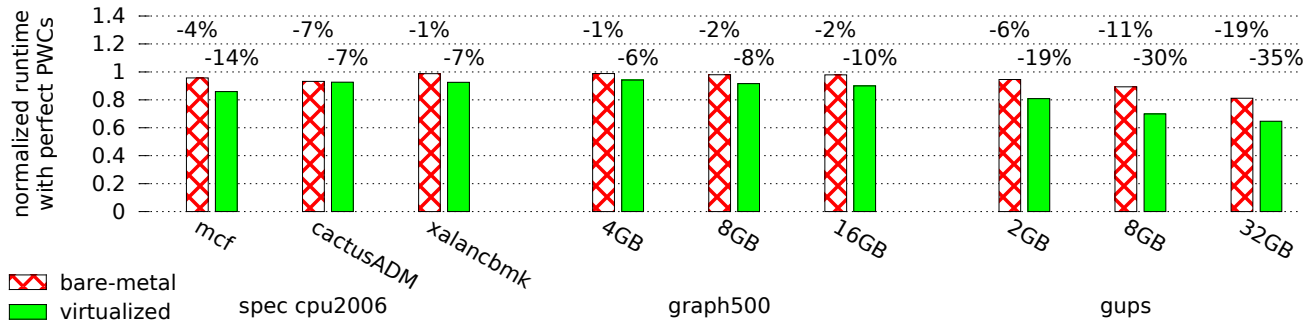

Figure 10: Simulating perfect PWCs (infinite and always hit) as compared to actual PWCs. Clearly, memory intensive workloads could achieve notable improvement gains if all PWC misses were eliminated.

tion hierarchy, denoted g, with an overall page walk length of $L_g = (L+1)^d - 1$, and (2) a host h with page walk length of $L_h = L$. With this notation, our formula trivially stems from the following lemma.

LEMMA 1. *If we host g on top of h, then the overall page walk length will become $(L_g + 1) \cdot (L_h + 1) - 1$.*

PROOF. Before hosting g on top of h, our induction hypothesis implies that a virtual memory reference by g induces $L_g + 1$ physical memory references: L_g for the table walk, plus one for the application data pointed to by the PTE at the end of the walk. Likewise, a virtual memory reference by h induces $L_h + 1$ physical memory references. When hosting g's nested hierarchy on top of h, each of the $L_g + 1$ guest physical references must now be translated to host physical addresses, so we end up with $(L_g + 1) \cdot (L_h + 1)$ references to the physical memory, which includes the access to the application data at the end of the table walk. We thus subtract one memory reference from this result. □

5.3 Hashed Paging Performs Better

Our two competing designs are hashed paging and radix paging with PWCs. The comparison between them is not "fair," since the latter employs additional cache structures—the PWCs—skewing the odds in its favor. Still, we find that hashed page tables perform better than radix tables for the workloads tested. (Recall that we focus on workloads that stress the TLB and PWCs; for workloads that experience few TLB and PWC misses, as do most SPEC CPU2006 benchmarks, hashed paging and radix paging perform similarly.) Figure 11 depicts the runtime improvement achieved by using hashed paging rather than radix paging with PWCs. The Graph500 and GUPS benchmarks show larger improvement gains as the memory footprint grows, because the PWC efficiency degrades, while hashed page tables require the same number of memory references per walk. The figure also confirms that the improvement opportunity from using hashed page tables is greater in the virtualized case, since hashed paging is exponentially more efficient, as was discussed above.

We now provide an in-depth analysis of hashed paging performance by comparing Figure 11 to Figure 10—the differences are summarized in Table 4. We see that, for the Graph500 and GUPS benchmarks, perfect PWCs reduce the runtime more than hashed page tables. The reason for this performance gap is the difference in the page walk length. Radix paging with perfect PWCs require 1 and 3 memory references per walk in bare-metal and virtualized setups, while hashed page tables exhibit hash collisions and thus require a few more, that is, 1.08 and 3.33 references on average. Our results confirm that reducing the hash table load factor and, hence, decreasing the hash collision rate closes the gap between the two designs.

Interestingly, Table 4 shows that hashed paging outperforms perfect PWCs for the three SPEC CPU2006 benchmarks that we examined. Radix paging with PWCs incurs additional cycles to search the PWCs, whereas hashed paging does not. Our analysis indicates that this PWC overhead, albeit small, is noticeable in the mcf and xalancbmk benchmarks. Table 4 also reveals that hashed paging cuts the cactusADM bare-metal runtime by 27%, whereas the perfect PWCs design reduces it by 7%. This significant performance gap merits further discussion, as it cannot be attributed to the PWC latency alone.

CactusADM solves a set of partial differential equations. The main loop references, on each iteration, multiple array elements whose virtual addresses are separated by a constant stride [21]. Since radix page tables map virtual addresses contiguously, the PTEs matching these virtual addresses also reside in physical addresses separated by a constant stride. And so, if the PTEs map to the same set in the data caches on the first iteration, they will conflict again on each subsequent iteration. Our analysis indicates that the lowest level PTEs, which are left out of the PWCs, indeed conflict in the L1 data cache due to this access pattern. We experimentally substantiated this finding by verifying that an (impractical) fully associative L1 data cache almost completely eliminates the L1 misses during page walks. Hashed page tables, on the other hand, are less sensitive to such cache pathologies because they randomly scatter the PTEs in the physical memory allocated to the hash table, lowering the probability for successive misses due to cache conflicts. The hashed paging randomization reduces the likelihood of encountering a worst-case input, similarly to the memory layout randomization that is used in STABILIZER to get predictable performance [16].

TO SUMMARIZE: The performance of radix page tables depends on PWCs, so their effectiveness degrades when workloads access larger and larger memory regions with lower locality. In contrast, our optimized hashed page tables cut the page walk overhead without resorting to PWCs, as their performance is unaffected by the memory footprint or the access pattern of the application.

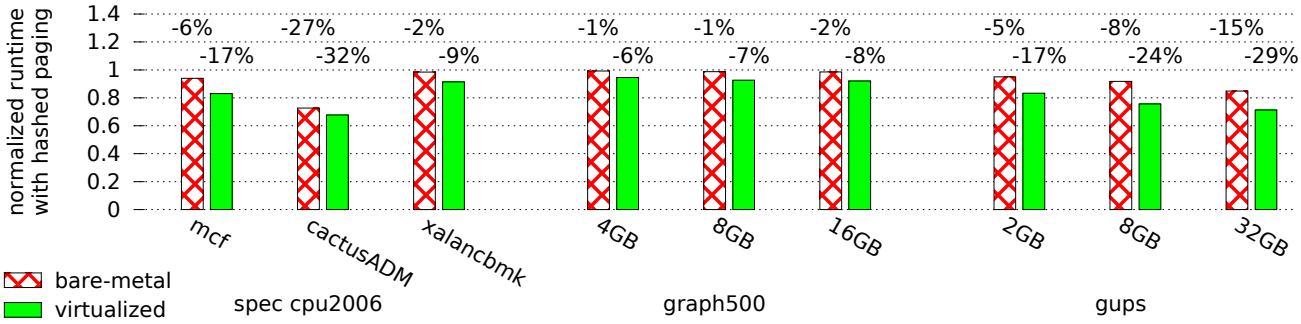

Figure 11: Normalized benchmark runtimes of hashed paging relative to radix paging with (real) PWCs.

benchmark	*bare-metal*		*virtualized*	
	perfect PWCs	*hashed paging*	*perfect PWCs*	*hashed paging*
SPEC mcf	-4%	-6%	-14%	-17%
SPEC cactusADM	-7%	-27%	-7%	-32%
SPEC xalancbmk	-1%	-2%	-7%	-9%
graph500 4GB	-1%	-1%	-6%	-6%
graph500 8GB	-2%	-1%	-8%	-7%
graph500 16GB	-2%	-2%	-10%	-8%
GUPS 2GB	-6%	-5%	-19%	-17%
GUPS 8GB	-11%	-8%	-30%	-24%
GUPS 32GB	-19%	-15%	-35%	-29%

Table 4: The runtime improvement achieved when utilizing (1) ideal PWCs and (2) hashed paging, as compared to radix paging with real PWCs. The improvement provided by hashed paging is comparable to that of ideal PWCs. The exception is cactusADM, for which hashed paging offers a much greater improvement due to radix caching pathologies.

5.4 Drawbacks of Hashed Paging

Practical implementation of hashed page tables is hindered by several obstacles [17]. First, hashed page tables are designed to avoid dynamic resizing, so they are allocated once and spread over a large memory area. This necessitates sharing the page table between all processes, contrary to the per-process page table employed with radix paging. Using a single page table for all processes has some advantages, e.g., saving the overhead of allocating and deallocating page tables as processes are created and terminated. One of the drawbacks is that killing a process requires a linear scan of the hash table to find and delete the associated PTEs, though this can be carried out lazily. Deleting a slot from an open addressed hash table is another challenge. Just marking the deleted slot as "empty" might truncate search chains that probed this slot, found it occupied and moved on to the next probe. Therefore, deleted slots should be distinguished with a special "deleted" value. But marking deleted slots has the undesirable side effect of a longer hash table lookup, because it no longer depends solely on the load factor.

Page tables should also support several features besides mapping virtual to physical addresses. Such features include sharing memory between processes and multiple page sizes. A feasible solution for both requirements is to add another level of translation, as implemented in the Power architecture [23, 37]. The Power architecture defines three types of addresses: effective, virtual, and physical. Each process owns a separate effective memory space, but the virtual address space is shared by all processes. This way, applications can share data by translating different effective segments to the same virtual segment. The effective and virtual address spaces are divided to 256MB segments, whereas the virtual and physical address spaces are divided to 4KB pages by default. Recent versions of Power allow different segments to be configured with different virtual memory page sizes: 4KB, 64KB, 16MB, or 16GB [24]. Multiple page sizes may boost performance with fewer virtual to physical translations, at the cost of potential waste of memory that was allocated but never used.

The Power architecture requires a two-level translation procedure for each memory reference: the first stage translates effective to virtual addresses, and the second stage translates virtual to physical addresses via hashed page tables. The Power architecture employs a closed addressing scheme, where each hash table slot contains a group of 8 PTEs, which form a constant size list. The hardware page walker uses two different hash functions to obtain two such groups of PTEs. If the VPN is not found in one of the 16 PTEs, the hardware triggers a page-fault interrupt that the OS must resolve.

This study investigates page table design with respect to their performance. In real environments that employ hashed page tables, we acknowledge that the aforementioned difficulties are likely to have an adverse affect on performance. Quantifying the precise implications is a nontrivial task; we leave this study for future work.

6. RELATED WORK

6.1 Optimizing MMU Caches

Several studies proposed optimizations to make MMU caches more efficient. Barr et al. [9] offered three new designs that, along with the existing designs by Intel and AMD, encompass the whole design space of MMU caches. They found that the optimal design is the "unified translation cache" with a modified insertion policy. This design caches entries from the different levels of the radix tree in the same structure, as AMD's page walk cache does. This makes it a flexible design, unlike Intel's Paging Structure Cache, which allocates a fixed portion of the cache to each tree level. But unified caches are not able to adapt well to workloads of varying sizes, because lower-levels PTEs, which have low reuse, may evict higher-levels PTEs, which show greater reuse. Therefore, the authors proposed the novel "variable insertion-point LRU replacement policy," and showed that it performs better than

the LRU algorithm. Their scheme favors higher-level PTEs by inserting entries from the second level into a recency position below the most recently used position. Employing this scheme raises the hit rates for the third and fourth level PTEs, but the hit rates for the second level remain very low, so page walks require two memory references on average. In other words, the proposed design still suffers from misses caused by the limited capacity of PWCs and is not able to close the performance gap to obtain the results of perfect PWCs or hashed page tables.

Bhattacharjee et al. [14] suggested two techniques to improve the performance and coverage of MMU caches. The first modifies the operating system to allocate pages for the radix page table in a way that encourages coalescing, i.e., in a way that is likely to place the page table pages in consecutive physical addresses. Coalescing can save valuable PWC entries with modest hardware changes, by detecting coalescing pages and unifying the corresponding PTEs into a single PWC entry. The authors focused on coalescing page table data from the lowest level in the radix tree, and they showed that on average dozens of pages can be coalesced.

The second technique proposed by Bhattacharjee et al. replaces the per-core PWCs with a single PWC with the same total capacity, which is shared between the cores. The shared PWC has higher access latency because it is a large structure that resides outside the core. But the authors proved that it yields higher hit rates that compensate for the longer access time and boost the performance of parallel and multiprogrammed workloads. The two techniques are orthogonal and, when applied together, achieve performance close to perfect PWCs. However, the study examined small to medium-sized applications, with memory footprints smaller than 8GB, so future studies will have to check that these results hold for big memory workloads as well. Assessing the performance of the proposed methods for virtualized systems with 2D page walks was not included in the study and certainly deserves examination.

6.2 Speculative Inverted Shadow Page Tables

Ahn et al. [3] studied the hardware support for 2D page walks in x86 architectures. First, they suggested shrinking the nested page walk by using a linear page table in the nested dimension. This design cuts the 2D page walk from 24 to 9 memory references, but is impractical for guests with 64-bit virtual address space, since linear page tables allocate a PTE for each page in the guest virtual space. Virtual machines with a 64-bit must use a hashed page table instead, because they can efficiently map addresses from a wide range to values in a fixed small range.

The authors also introduced the "speculative inverted shadow page table" (SISPT) to reduce the cost of 2D page walks. SISPT is a variant of shadow page tables and hashed page tables, which acts as a third-level TLB that resides in the physical memory. When the hardware finds the translation in the SISPT, it obtains it with a single memory reference; otherwise, it invokes a page fault to be handled by the operating system. Not all translations are stored in the SISPT, so the guest and nested page tables are maintained to provide a complete memory mapping. Given the large size of the SISPT, it is shared between all virtual machines and the processes inside them. Synchronization between the SISPT and the full page table requires hypervisor intervention at context switches and other page table updates. This is the primary overhead associated with shadow paging (and with every software managed TLB). To eliminate the overhead of hypervisor interventions, the SISPT does not try to remain consistent with guest page table updates. On a TLB miss, the processor speculatively executes with the mapping found at the SISPT while simultaneously performing a full 2D walk. If the processor finds the speculated translation to be incorrect, the processor pipeline is flushed and the SISPT is updated. The SISPT mechanism presents a trade-off between decreased page walk latency and increased complexity of hardware (the added page walker and the speculative execution engine).

Choosing which guest and nested page tables are used to keep the full mapping is independent of the SISPT design choice. The different alternatives have little effect on the overall performance, because the speculative walks hide most of the non-speculative walk latencies. But quick 2D page walks are still essential; without them, the core will be stalled waiting for the correct translation. Thus, our 2D hashed page tables may be used as the backing page tables to shorten the non-speculative page walks.

6.3 Direct Segments

Basu et al. [10] analyzed big-memory server workloads such as databases, in-memory caches, and graph analytics. They found that these workloads pay a high price for page-based virtual memory but often do not need the benefits that virtual memory offers: they rarely use swapping, allocate most of their memory on startup, and grant read-write permissions to nearly all their memory. The authors therefore suggested that such applications may benefit from using paging only in a small portion of their address space, while translating the rest of the address space via segmentation. Their idea, "direct segments," requires modest hardware and software changes. Each core is equipped with three registers, `base`, `limit` and `offset`, which map a large, contiguous portion of the process's virtual address space. The hardware translates the virtual addresses lying in the range [`base`, `limit`) to physical addresses by simply adding the constant `offset`, without resorting to the TLB. The OS is responsible for allocating the required physical memory and setting these registers accordingly. While direct segments eliminate the majority of the TLB misses in big-memory workloads, they are less suitable for compute workloads with unpredictable memory usage and for environments where many processes execute for short periods.

The authors also extended direct segments to reduce the address translation overhead in virtualized setups [18]. They offered three modes of operation: direct segments in the guest dimension (guest direct), in the host dimension (VMM direct), or in both (dual direct). The modes present different trade-offs between performance and restrictions on other virtual memory features. The guest and the VMM modes achieve near bare-metal performance with modifications confined to the guest and the hypervisor, respectively, whereas the third mode achieves almost zero translation overhead. However, the dual direct mode requires changes in the guest and the hypervisor, and it provides limited support for memory overcommitment and sharing pages between virtual machines. The VMM mode allows swapping in the guest, whereas the guest mode supports page sharing and live migration of virtual machines. Fragmentation can prevent the creation of direct segments and can be removed via compaction.

7. CONCLUSIONS

Virtual memory offers many advantages and plays a vital role in computer systems, but it comes at the price of performance degradation. The commonly used x86-64 radix page tables require 4 and 24 memory references per walk in bare-metal and virtualized systems. Current radix designs attempt to hide these expensive accesses to the memory hierarchy with PWCs. But this approach is suboptimal, as emerging big-data workloads reference vast amounts of memory with low locality. In contrast, the number of required memory references along a hashed page walk does not depend on the application memory footprint or access locality. We show that carefully designing hashed page tables shrinks bare-metal and virtualized page walks to just above one and three memory reference per walk, respectively, without resorting to PWCs. The disadvantage of hashed page tables is that they make it more challenging to efficiently provide such functionalities as superpages and page sharing.

8. ACKNOWLEDGMENTS

We thank Thomas Barr, Alan Cox, and Scott Rixner for providing valuable feedback and sharing with us their memory simulator, which allowed us to reproduce and compare against their results [9]. We thank the anonymous reviewers and our shepherd, Alex Liu, for their feedback. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 688386 (OPERA).

9. REFERENCES

[1] Transparent hugepage support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2016. Linux documentation page (Accessed: Apr 2016).

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2006. http://dx.doi.org/10.1145/1168857.1168860.

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 476–487, 2012. http://dx.doi.org/10.1145/2366231.2337214.

[4] AMD, Inc. *AMD-V Nested Paging*, 2008. White Paper available at: http://developer.amd.com/wordpress/media/2012/10/NPT-WP-11-final-TM.pdf. (Accessed: Apr 2016).

[5] AMD, Inc. *AMD64 Architecture Programmer's Manual, Volume 2*, 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf. (Accessed: Apr 2016).

[6] ARM Holdings. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*, 2014. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf. (Accessed: Apr 2016).

[7] Vlastimil Babka and Petr Tůma. Investigating cache parameters of x86 family processors. In *SPEC Benchmark Workshop on Comput. Performance Evaluation and Benchmarking*, pages 77–96, 2009. http://dx.doi.org/10.1007/978-3-540-93799-9_5.

[8] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, and Jeremiah Willcock. Graph 500 benchmark. http://www.graph500.org/specifications, 2011. Version 1.2 (Accessed: Apr 2016).

[9] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip don't walk (the page table). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 48–59, 2010. http://dx.doi.org/10.1145/1815961.1815970.

[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013. http://dx.doi.org/10.1145/2485922.2485943.

[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–35, 2008. http://dx.doi.org/10.1145/1346281.1346286.

[12] Nikhil Bhatia. Performance evaluation of AMD RVI hardware assist. Technical report, VMware, Inc., 2009. http://www.cse.iitd.ernet.in/~sbansal/csl862-virt/2010/readings/RVI_performance.pdf. (Accessed: Apr 2016).

[13] Nikhil Bhatia. Performance evaluation of Intel EPT hardware assist. Technical report, VMware, Inc., 2009. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf. (Accessed: Apr 2016).

[14] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 383–394, 2013. http://dx.doi.org/10.1145/2540708.2540741.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[16] Charlie Curtsinger and Emery D Berger. STABILIZER: Statistically sound performance evaluation. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 219–228, 2013. http://dx.doi.org/10.1145/2451116.2451141.

[17] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other MMU tricks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–237, 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.1609.

[18] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–189, 2014. http://dx.doi.org/10.1109/MICRO.2014.37.

[19] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium: A system implementor's tale. In *USENIX Annual Technical Conference (ATC)*, pages 264–278, 2005.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.3059.

[20] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News (CAN)*, 34(4):1–17, sep 2006. http://dx.doi.org/10.1145/1186736.1186737.

[21] John L. Henning. SPEC CPU2006 memory footprint. *ACM SIGARCH Computer Architecture News (CAN)*, 35(1):84–89, mar 2007. http://doi.acm.org/10.1145/1241601.1241618.

[22] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 39–50, 1993. http://dx.doi.org/10.1145/165123.165128.

[23] IBM Corporation. *PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, 2005. https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf. (Accessed: Apr 2016).

[24] IBM Corporation. *AIX Version 6.1 Performance Management*, first edition, 2007. http://ps-2.kev009.com/basil.holloway/ALLPDF/sc23525300.pdf. (Accessed: Apr 2016).

[25] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Volume 2*, 2010. http://tinyurl.com/itanium2. (Accessed: Apr 2016).

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, 2015. http://tinyurl.com/intel-x86-3a. (Accessed: Apr 2016).

[27] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, 2015. http://tinyurl.com/intel-x86-3b. (Accessed: Apr 2016).

[28] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, 2015. http://tinyurl.com/intel-x86-3c. (Accessed: Apr 2016).

[29] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 295–306, 1998. http://dx.doi.org/10.1145/291069.291065.

[30] David Koester and Bob Lucas. RandomAccess – GUPS (Giga updates per second). http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/. (Accessed: Apr 2016).

[31] Jun Min Lin, Yu Chen, Wenlong Li, Zhao Tang, and Aamer Jaleel. Memory characterization of SPEC CPU2006 benchmark suite. In *Workshop for Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2008. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf. (Accessed: Apr 2016).

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM International Conference on Programming Languages Design and Implementation (PLDI)*, pages 190–200, 2005. http://dx.doi.org/10.1145/1065010.1065034.

[33] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC challenge (HPCC) benchmark suite. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2006. http://dx.doi.org/10.1145/1188455.1188677. An SC tutorial, available via http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/sc06_hpcc.pdf. (Accessed: Apr 2016).

[34] Collin McCurdy, Alan L. Cox, and Jeffrey Vetter. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 95–104, 2008. http://dx.doi.org/10.1109/ISPASS.2008.4510742.

[35] Richard C. Murphy, Kyle B. Wheeler, and Brian W. Barrett. Introducing the Graph 500. In *Cray User Group Conference (CUG)*, 2010. https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf. (Accessed: Apr 2016).

[36] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, 2002. http://dx.doi.org/10.1145/844128.844138.

[37] C. Ray Peng, Thomas A. Petersen, and Ron Clark. The PowerPC architecture: 64-bit Power with 32-bit compatibility. In *IEEE Computer Society International Computer Conference (COMPCON)*, pages 300–307, 1995. http://dx.doi.org/10.1109/CMPCON.1995.512400.

[38] Cristan Szmajda and Gernot Heiser. Variable radix page table: A page table for modern architectures. In *Advances in Computer Systems Architecture, Asia-Pacific Conference (ACSAC)*, pages 290–304, 2003. http://dx.doi.org/10.1007/978-3-540-39864-6_24.

[39] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–200, 1995. http://dx.doi.org/10.1145/224056.224071.

[40] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 84–97, 2003. http://dx.doi.org/10.1145/859618.859629.