Translation Caching: Skip, Don't Walk (the Page Table)

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

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

ABSTRACT

This paper explores the design space of MMU caches that accelerate virtual-to-physical address translation in processor architectures, such as x86-64, that use a radix tree page table. In particular, these caches accelerate the page table walk that occurs after a miss in the Translation Lookaside Buffer. This paper shows that the most effective MMU caches are *translation caches*, which store partial translations and allow the page walk hardware to skip one or more levels of the page table.

In recent years, both AMD and Intel processors have implemented MMU caches. However, their implementations are quite different and represent distinct points in the design space. This paper introduces three new MMU cache structures that round out the design space and directly compares the effectiveness of all five organizations. This comparison shows that two of the newly introduced structures, both of which are translation cache variants, are better than existing structures in many situations.

Finally, 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, such as the inverted page table, outperformed organizations based upon radix trees for supporting large virtual address spaces. However, these studies did not take into account the possibility of caching page table entries from the higher levels of the radix tree. This paper shows that any of the five MMU cache structures will reduce radix tree page table DRAM accesses far below an inverted page table.

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

ISCA'10, June 19–23, 2010, Saint-Malo, France.

Keywords

TLB, Memory Management, Page Walk Caching

1. INTRODUCTION

This paper explores the design space of memory-management unit (MMU) caches for accelerating virtual-to-physical address translation in processor architectures, like x86-64, that implement paged virtual memory using a radix tree for their page table. In particular, these caches accelerate the page table walk that occurs after a miss in the Translation Lookaside Buffer (TLB). In fact, a hit in some of these caches enables the processor to *skip* over one or more levels of the tree and, in the best case, access only the tree's lowest level.

For several generations of x86 processors, from the Intel 80386 to the Pentium, the page table had at most two levels. Consequently, whenever a TLB miss occurred, at most two memory accesses were needed to complete the translation. However, as the physical and virtual address spaces supported by x86 processors have grown in size, the maximum depth of the tree has increased, first to three levels in the Pentium Pro to accommodate a 36-bit physical address within a page table entry, and more recently to four levels in the AMD Opteron to support a 48-bit virtual address space. In fact, with each passing decade since the introduction of the 80386, the depth of the tree has grown by one level.

Recent work has shown the impact of TLB misses on overall system performance ranges from 5-14% for nominally sized applications, even in a non-virtualized environment [9]. As the application's memory footprint increases, TLB misses have a significantly larger impact on performance, approaching 50% in some cases [19]. Although the use of large pages can lessen this impact, with further increases in the memory footprint their effectiveness declines. Therefore, both AMD and Intel have implemented MMU caches for page table entries from the higher levels of the tree [3, 9]. However, their caches have quite different structure. For example, AMD's Page Walk Cache stores page table entries from any level of the tree, whereas Intel implements distinct caches for each level of the tree. Also, AMD's Page Walk Cache is indexed by the physical address of the cached page table entry, whereas Intel's Paging-Structure Caches are indexed by portions of the virtual address being translated. Thus, in this respect, the Page Walk Cache resembles the processor's data cache, whereas the Paging-Structure Caches resemble its TLB.

This paper's primary contribution is that it provides the first comprehensive exploration of the design space occupied by these caches. In total, it discusses five distinct points in this space, including three new designs. Specifically, it presents the first head-to-head comparison of the effectiveness of these designs. In general, the results of this comparison show that the *translation caches*,

which store partial translations and allow the page walk hardware to skip one or more levels of the page table, are the best. In addition, the new translation caches that are introduced by this paper are better than the existing caches in many situations and workloads.

Finally, 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, such as the inverted page table, outperformed organizations based upon radix trees for supporting large virtual address spaces [23, 15]. However, these studies did not take into account the possibility of caching page table entries from the higher levels of the radix tree. This paper shows that radix tables cause up to 20% fewer total memory accesses and up to 400% fewer DRAM accesses than hash-based tables because of the locality in virtual address use.

This paper is organized as follows. The next section provides the background for the rest of the paper. Specifically, it summarizes the relevant aspects of x86/x86-64 virtual-to-physical address translation. Section 3 describes the design space, identifying the essential differences between AMD's Page Walk Cache, Intel's Paging-Structure caches, and the new structures developed in this paper. Section 4 qualitatively compares these structures, and Section 5 describes this paper's methodology for quantitatively comparing them. Section 6 presents quantitative simulation results of their effectiveness as compared to one another, and Section 7 compares the effectiveness of a radix tree page table with these structures to competing page table designs. Additionally, Section 7 examines the interaction between the MMU caches and large pages. Section 8 discusses the related work. Finally, Section 9 summarizes this paper's conclusions.

2. 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. As the virtual address space

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.

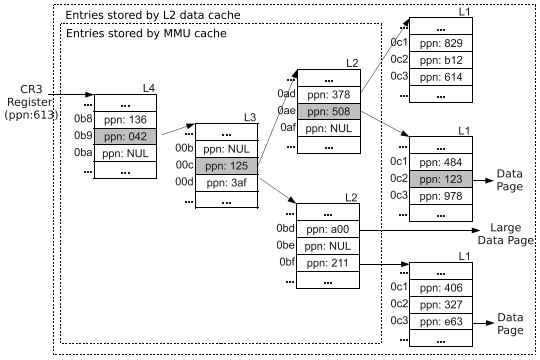

Figure 2: An example page walk for virtual address (`0b9, 00c, 0ae, 0c2, 016`). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures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

grows, additional index fields (*e.g.*, L5) may be added, reducing the size of the *se* field.

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 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 `0x00005c8315cc2016`. 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 address that is used to address 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). The process is repeated by extracting the L3 index field from the virtual address and appending it to this physical page number to address 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.

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.

3. CACHING PAGE WALKS

While radix-tree page tables require many accesses to translate a single address, the accesses to the upper level page table entries have significant temporal locality. Walks for two consecutive pages in the virtual address space will usually use the same three upper level entries, since the indices selecting these entries come from high-order bits of the virtual address, which change less frequently.

While the MMU does access the page table through the memory hierarchy, it only has access to the L2 data cache in at least one major commercial x86 design [9]. Since the L2 data cache is relatively slow on modern CPUs, accessing three upper-level page table entries on every page walk will incur a penalty of several tens of cycles per TLB miss, even if all entries are present in the L2 data cache.

Therefore, the x86 processor vendors have developed private, low-latency caches for the MMU that store upper level page table entries [9, 3]. In this section, we describe the design space and provide a nomenclature for the different tagging and partitioning schemes used by these MMU caches.

MMU caches may store elements from the page table tagged by their physical address in memory, as a conventional data cache might. We call such MMU caches *page table caches*. Examples include AMD's *Page Walk Cache* and the L2 data cache, although it is not private to the MMU. Alternatively, MMU caches can be indexed by parts of the virtual address, like a TLB. We call such MMU caches *translation caches*. Intel's *Paging-Structure Caches* are translation caches.

For either of these tagging schemes, elements from different levels of the page table can be mixed in a single cache (a *unified* cache), or placed into separate caches (a *split* cache). Finally, each cache entry can store an entry from one level along the page walk, or it can store an entire path (a *path* cache).

3.1 Page table caches

The simplest example of a page table cache is the processor's L2 data cache. The page walker generates a physical address based upon the page table page to be accessed and an index from the virtual address. This physical address is then fetched from the processor's memory hierarchy starting with the L2 data cache.

Page table caches use this same indexing scheme. Elements are tagged with their physical address in the page table. These tags are the size of the physical page number plus the size of one page table index. L1 entries are not cached in any of the designs presented here (since the TLB itself caches those entries).

Base Location	Index	Next Page
125	0ae	508
042	00c	125
613	0b9	042
...	...	...

Figure 3: An example of the contents of a UPTC. Each entry is tagged with the address of a page table entry, consisting of the 40-bit physical page number of the page table page and a 9-bit index into it. The entry then provides a 40-bit physical page number for the next lower level page table page. (Only 12 bits of the physical page numbers are shown, for simplicity.)

	Base Location	Index	Next Page
L2 entries	125 ...	0ae ...	508 ...
L3 entries	042 ...	00c ...	125 ...
L4 entries	613 ...	0b9 ...	042 ...

Figure 4: An example of the contents of a SPTC. Each entry holds the same tag and data as in the UPTC.

3.1.1 Unified Page Table Cache (UPTC)

The simplest design for a dedicated page table cache is a single, high-speed, read-only cache for page table entries, tagged by their physical address in memory. Entries from different levels of the page table are mixed in the same cache, all indexed by their physical address. Such a cache is analogous to a private, read-only L1 data cache for page table entries. However, like a TLB, coherence between this cache and the page table can be maintained by software with little overhead. AMD's Page Walk Cache has this design [9].

Figure 3 shows an example of the Unified Page Table Cache (UPTC) after the MMU walks the page table to translate the virtual address (0b9, 00c, 0ae, 0c2, 016). If the MMU subsequently tries to translate the virtual address (0b9, 00c, 0ae, 0c3, 103), the page walk will begin by looking for the page table entry 0b9 in the L4 page table page (located at 613 and referenced by the CR3 register). Since this page table entry is present in the UPTC, it does not need to be loaded from the memory hierarchy.

This entry indicates that the L3 page table page has physical page number 042. The same process is then repeated to locate the L2 and L1 page table pages. Once the address of the L1 page table page is found, the appropriate entry is loaded from memory to determine the physical page address of the desired data.

Without a page table cache, all four of these accesses to page table entries would have required a memory access, each of which may or may not hit in the L2 data cache. In contrast, with the page table cache, the three top entries hit in the private page table cache, and only one entry (the L1 entry) requires a memory access, which may or may not hit in the L2 data cache.

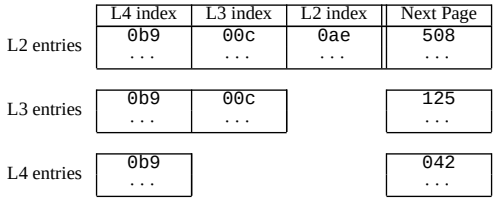

Figure 5: An example of the contents of a STC. Each index is 9-bits, and the data holds a 40-bit physical page number of the next page table level. An entry in the L2 cache must match on all three indices, an entry in the L3 must match on two and the L4 on one.

L4 index	L3 index	L2 index	Next Page
0b9	00c	0ae	508
0b9	00c	xx	125
0b9	xx	xx	042
...	...	...	...

Figure 6: An example of the contents of a UTC. An "xx" means "don't care".

3.1.2 Split Page Table Cache (SPTC)

An alternate design for the page table cache separates the page table entries from different levels into separate caches. Figure 4 illustrates a Split Page Table Cache (SPTC). In this design, each individual entry contains the same tag and data as it would in the unified page table cache. The primary difference is that each page table level gets a private cache, and entries from different levels do not compete for common slots.

3.2 Translation caches

As an alternative to tagging cache entries by their physical address, they can be tagged by their indices in the virtual address. An L4 entry will be tagged by the 9 bit L4 index, an L3 entry with the L4 and L3 indices, and an L2 entry with the L4, L3, and L2 indices. We call this device a *translation cache*, because it is storing a partial translation of a virtual address.

With this tagging scheme, data from one entry is not needed to lookup the entry at the next lower level of the page table. All of the lookups can be performed independently of each other. In the end, the MMU will select the entry that matches the longest prefix of the virtual address because it allows the page walk to skip the most levels.

3.2.1 Split Translation Cache (STC)

Like an SPTC, the Split Translation Cache (STC) stores entries from different levels of the page table in separate caches. However, as shown in Figure 5, the STC uses a different way of tagging the entries. The Intel *Paging-Structure Caches* [3] exemplify the STC organization.

The example in Figure 5 shows the split translation cache after the MMU walks the page table to translate the virtual address (`0b9, 00c, 0ae, 0c2, 016`). If the MMU subsequently starts to translate the virtual address (`0b9, 00c, 0dd, 0c3, 929`), it will attempt to locate the L1, L2 and L3 page table pages in their corresponding caches using portions of the virtual address. The location of the L3 page table page would be stored in the L4 entry cache and tagged by the L4 index, (`0b9`). Similarly, the location of the L2 page table page would be stored in the L3 entry cache and tagged by the L4 and L3 indices, (`0b9, 00c`). Finally, the location of the L1 page table page would be stored in the L2 entry cache and tagged by the L4, L3 and L2 indices, (`0b9, 00c, 0dd`).

These searches can be performed in any order, and even in parallel. In the above example, the cache can provide the location of the appropriate L3 and L2 page table pages, but not the L1 page table page, as (`0b9, 00c, 0dd`) is not present in the L2 entry cache. Ultimately, the MMU would use the (`0b9, 00c`) entry from the L3 entry cache because it allows the page walk to begin further down the tree, at the L2 page table page.

3.2.2 Unified Translation Cache (UTC)

Just as the page table caches can be built with either a split or a unified organization, a Unified Translation Cache (UTC) can also be built. Moreover, just like the UPTC, the UTC mixes elements from all levels of the page table in the same cache.

Figure 6 shows the UTC after the MMU walks the page table to translate the virtual address (`0b9, 00c, 0ae, 0c2, 016`). If the MMU subsequently starts to translate the virtual address (`0b9, 00c, 0dd, 0c3, 929`), it will first look in the UTC for the physical page numbers of the L1, L2 and L3 page table pages. As with the previous example that used the STC, the MMU finds two matching entries in the UTC. Ultimately, the MMU decides to use the UTC's second entry, which is an L3 entry that has the L4 and L3 indices (`0b9, 00c`) as its tag, because this tag matches the longest prefix of the virtual address. Thus, the MMU can skip the L4 and L3 page table pages and start walking from the L2 page table page.

3.3 Translation-Path Cache (TPC)

Note that in the UTC example in Figure 6, the tags for the three entries representing a single path down the page table all have the same content. The L4 and L3 entries use less of the virtual address than the L2 entry does, but the fragments that they do use are the same. Consequently, it is possible to store all three physical page numbers from this example in a single entry. In such a *Translation-Path Cache* (TPC), a single entry represents an entire path, including all of the intermediate entries, for a given walk instead of a single entry along that walk.

L4 index	L3 index	L2 index	L3	L2	L1
0b9	00c	0ae	042	125	508
...	...	...	...	...	...

Figure 7: An example of the contents of the TPC after the virtual address (`0b9, 00c, 0ae, 0c2, 016`) is walked. The TPC holds three 9 bit indices, as the translation caches do, but all three 40-bit physical page numbers are stored for all three page table levels.

The example in Figure 7 shows the TPC after the MMU walks the page table to translate the virtual address (`0b9, 00c, 0ae, 0c2, 016`). All data from that walk is stored in one entry. If the MMU subsequently starts to translate the virtual address (`0b9, 00c, 0ae, 0c3, 929`), the entry referencing the L1 page table page is discovered just as it would have been in the unified translation cache. Specifically, the MMU finds the entry in the cache with the tag (`0b9, 00c, 0ae`) and reads the physical page number `508` of the L1 page table page from this entry.

If the MMU later starts to translate the virtual address (`0b9, 00c, 0de, 0fe, 829`), this address shares a partial path (`0b9, 00c`) with the previously inserted entry. Therefore, the translation-path cache will provide the physical address of the appropriate L2 page table page.

3.4 Design space summary

In summary, the caches described in this section fit into the following two-dimensional design space (annotated with the section number in which each design is described):

	Unified	Split	Path
Page Table Cache	3.1.1	3.1.2	N/A
Translation Cache	3.2.2	3.2.1	3.3

The unified page table cache (UPTC) is the design that appears in modern AMD x86-64 processors. The split translation cache (STC) is the design that appears in modern Intel x86-64 processors. The remaining three designs have been introduced in this paper.

Note that there is no useful page table counterpart to the translation-path cache. This is a direct result of the indexing scheme. While a "path" of physical addresses could be stored as an MMU cache index, it would have to be searched sequentially because the MMU cannot create a path of physical page numbers directly from the virtual address. It must look up each physical page in turn. Therefore, storing complete paths would yield no benefit over the other page table cache designs.

4. DESIGN COMPARISON

All of the designs presented in the previous section are able to accelerate page walks by caching information from the upper levels of the page table. However, these designs have differences in their indexing, partitioning, coverage, and complexity. This section discusses the effects of these differences.

4.1 Indexing

The indexing scheme determines how the cache is searched. Cache indices can be derived from the physical addresses of components of the page table or they can be derived from the virtual address and correspond to the levels of the page table.

Page table caches use the physical addresses of the page table entries as indices. In fact, they operate identically to any physically indexed cache—a UPTC is essentially another data cache in the memory hierarchy dedicated to the page table. The MMU will generate a physical address for the page table entry at each stage of the page walk, and that address will be used as an index into the appropriate page table cache. While this leads to a simple design, it requires the cache lookups to occur in a top-down order. The result of the L4 entry search is required before the L3 entry search can begin, because the L4 entry gives the physical page number of the L3 page table page, which is needed to generate the physical address of the L3 page table entry. Similarly, the L2 search is dependent on the result of the L3 search. In the case where the cache holds all three entries (L4, L3, and L2), the cache must be accessed three times to generate the physical address of the L1 page table entry.

In contrast, translation caches use components of the virtual address as indices. For example, the TLB is an L1 translation cache that uses the virtual page number as its index. In general, for translation caches, the MMU uses a subset of the virtual page number as the index. This allows the translation caches to be searched in any order (L4 first, L2 first, or in parallel). Thus, on a TLB miss, the L2 translation cache can immediately be searched. Upon a hit which yields the L2 translation, the address of the L1 page table entry can be computed immediately. If no L2 translation is available, the L3, then L4, translation caches can be searched. Upon a hit, the page walk would begin at that point in the tree.

4.2 Partitioning

MMU caches can either be unified or partitioned, or they can store complete path information. The partitioning of the cache determines how the entries of the cache are allocated to different levels of the page table. This effectively determines how well the entries from different levels are isolated from each other.

The impact of the partitioning scheme largely depends on whether the application densely or sparsely utilizes its virtual address space. For applications that densely use their virtual memory, a few L4 and L3 entries are heavily utilized and there is significant reuse of the L2 entries. In contrast, for applications that sparsely use their virtual memory, there will be little reuse for L2 entries, but a larger number of L4 and L3 entries will exhibit reuse. The partitioning strategy and replacement policy determine how these entries will compete for slots, which can have a significant impact on the effectiveness of the MMU cache.

For applications that densely use their virtual memory, there will be many more L2 entries than upper level entries in use. However, for a page table cache design, these upper level entries are critical for translation performance. If a random replacement scheme is used in conjunction with a unified cache, these important entries can be frequently replaced, resulting in memory accesses to entries at or near the top of the table. However, if entries from different levels are kept in separate caches, a random replacement policy is less detrimental.

For applications that sparsely use many gigabytes of virtual memory, L2 entries will have very little reuse, and effective caching of L3 entries is critical. In a split entry cache, a static allocation of entries to each level must be made. If this allocation is optimized for small applications, it will have many more L2 entries than L3 entries, harming performance in this situation. Moreover, if an application makes heavy use of large pages and limited use of small pages, the dedicated L2 entries will be of little or no use. In contrast, in a unified cache, the allocation happens dynamically, but recently accessed L2 entries that will not be reused might evict L3 entries that might otherwise be reused. A level-aware replacement policy can help to avoid this.

The Greedy Dual algorithm is a popular content-aware replacement scheme [25]. This algorithm will replace recently used entries early if they are easier to reload into the cache. This scheme can be adapted for MMU caches by preferentially replacing lower-level entries with upper-level entries, thus reducing conflict between entries of high and low reuse. While Greedy Dual is not an algorithm that can easily be implemented in hardware, it is possible to implement a similar algorithm with minimal modification to an LRU cache.

In our modified LRU algorithm, entries from lower levels of the page table are inserted into the LRU queue at a recency position behind the most-recently-used position. If these lower level entries are reused, they are promoted to the most recently used position. However, if they are not reused, the portion of the cache in which lower level entries compete with upper level entries is small. These positions can be fixed, for simplicity, or they can change to adapt to different workloads. We propose a *variable insertion-point LRU replacement policy* whereby entries from lower levels of the cache are inserted into a recency position below the most recently used position that is proportional to the current number of upper level entries stored. For example, if there are two L4 entries and six L3 entries currently in the cache, a new L2 entry is inserted in the ninth most recently used position in the cache.

The path translation cache avoids these partitioning problems, as each slot holds an entry from all levels. This prevents the competition for slots while not requiring a static allocation of slots to

		Unified	Split	Path
Page Table	C	1	$l-1$	N/A
	T	$p-3$	$\{p-3,...,p-3\}$	
	D	$p-12$	$\{p-12,...,p-12\}$	
Translation	C	1	$l-1$	1
	T	$(l-1)\cdot n$	$\{n,...,(l-1)\cdot n\}$	$(l-1)\cdot n$
	D	$p-12$	$\{p-12,...,p-12\}$	$3(p-12)$

Table 1: The number of caches (C), the number of tag bits per entry (T), and the number of data bits per entry (D) for each design.

		Unified	Split	Path
Page Table	C	1	3	N/A
	T	49	$\{49,49,49\}$	
	D	40	$\{40,40,40\}$	
Translation	C	1	3	1
	T	27	$\{9,18,27\}$	27
	D	40	$\{40,40,40\}$	120

Table 2: MMU cache parameters for x86-64 processors. We have used the architectural definition of physical address width, 52 bits [1]. Actual implementations may use fewer bits.

levels. However, since this cache may hold many paths with the same upper level entries, its effective capacity for holding upper level entries is less than three equally sized split entry caches.

4.3 Coverage

Characterizing the coverage of an MMU cache is not straightforward. In particular, the exact meaning of coverage for an MMU cache must first be considered. For example, suppose an address translation hits on an L3 entry in a page table or translation cache but does not hit on an L2 entry. In this case, the translation was accelerated by the MMU cache, but nonetheless required a memory access to fetch the page table's L2 entry. Thus, it is arguable whether or not the MMU cache provided coverage. We take the strict position that coverage means that no memory accesses were required to fetch L4, L3, or L2 page table entries.

In general, with the same number of entries, translation caches are able to cover a larger portion of the address space than page table caches. The reason is that a translation cache can make more efficient use of its entries than a page table cache. For a page table cache to provide coverage it must simultaneously hold an L4, L3, and L2 entry, whereas a translation cache can provide coverage with only an L2 entry. In other words, the translation cache may be able store additional L2 entries in place of the L4 and L3 entries that are required to provide coverage in the page table cache.

When the application is simply too large for the MMU cache to provide full coverage, the unified caches with intelligent replacement policies, the split caches, and the TPC are able to accelerate translations for more of the address space than the unified caches with conventional replacement policies. This is due to the fact that the former caches upper level entries, which provide partial translation for larger regions, are less likely to be evicted in favor of lower level entries. Since these effects are highly dependent on workload, the relative hit rates of the cache designs are studied experimentally in Section 6.

4.4 Complexity

All of the organizations are effectively fully associative caches, which can be implemented by a CAM array to match the tags and a RAM array to store the data entries. However, the different cache organizations have different tag and data widths, and will potentially require differing numbers of entries to achieve similar hit rates. These factors will lead to different implementation complexities for the different organizations.

Table 1 shows the number of caches, tag bits per entry, and data bits per entry that are needed for each organization. These characteristics are parameterized by the number of levels of address translation, l, the number of bits in a physical address, p, and the number of offset bits in a page table index for a particular level, n. In x86-64 processors, $l = 4$, $p = 52$, and $n = 9$, which leads to the values shown in Table 2.

It should be noted that for current architectural parameters, translation caches require significantly smaller tags. This will make a translation cache smaller and more power efficient than an equivalent page table cache, as the CAM array is likely to dominate the cost of the structure.

5. METHODOLOGY

The MMU cache architectures presented in Section 3 were evaluated by running application memory traces through a memory system simulator. The trace-based approach is warranted here for two reasons. First and foremost, the number of memory references required for a page walk is effectively independent of all architectural parameters except for the MMU and the L2 cache organization. A cycle-accurate simulation would have presented a more limited view of the differences for a single point in the processor design space. Second, from a practical standpoint, it would have been nearly impossible to run the types of large memory footprint applications that benefit most from these structures on a slow cycle-accurate simulator.

5.1 Application Memory Traces

The AMD SimNow [8] platform simulator was used to run various benchmarks under FreeBSD 8.0-Release for x86-64. A custom analyzer plugin to SimNow records each virtual memory access made by the simulated system. This trace includes all memory loads and stores made by the tested operating system and processes, but it does not include instruction or page table loads. TLB and MMU invalidations are included in the trace by monitoring the value of the CR3 register, which must change on a context switch. Finally, the plugin counts the total number of instructions executed during the trace.

Virtual memory access traces were captured from these applications, including the SPEC CPU2006 floating-point suite [14], SPECjbb2005 [21], ASCI Sweep3d [2] and HPCC RandomAccess [11]. 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.

5.2 Memory System Simulation

A custom memory system simulator was built to simulate the various MMU cache designs. The simulator includes an MMU that closely resembles the L1 and L2 TLBs in the AMD Opteron [9]. It consists of a 64-entry, fully-associative L1 TLB with random replacement, and a 512-entry, 4-way set associative L2 TLB with LRU replacement. Furthermore, the simulator is able to model all five cache designs described in Section 3. The simulator stores tags (virtual addresses), but not data (physical addresses), to eliminate any operating system dependent behavior from the simulation. This simplifies the design of the simulator and generalizes the results. Unless otherwise specified, the simulator divides all memory into 4KB pages.

A 1MB L2 cache was included in the model, simulated using the Dinero IV cache simulator [12]. Both application data accesses and MMU page table accesses are simulated using a shared L2 cache

model. In general, instruction loads are not instrumented are not included in this study. The cache parameters were based on the same AMD Opteron processor that was the basis for the TLB parameters. The L1 cache was not simulated, since the page walk hardware does not use it on the Opteron.

While our simulation environment did not permit us to directly measure power and system performance, the reduction in memory accesses we directly measure here should translate directly into reduced interconnect power consumption and latency. Recall that previous work has shown the uncached system performance impact of unvirtualized TLB misses to be up to 14% for nominally sized applications [9] and up to 50% for large applications [19].

5.3 Synthetic Application Memory Traces

To study the behavior of an application that uses more virtual memory than we can practically trace on our real machines, a trace synthesizer was developed that simulates the memory access pattern of an in-memory database, performing a hash join. Such joins are common, and the performance of large joins is representative of overall database performance [6].

The simulated join is an inner join on two equally sized tables, A and B. The hash join process starts by creating a hash table containing the entries of B, using an open addressing collision resolution scheme. The database then iterates through A, checking to see if each entry is present in the hash table. The result is then placed in an output table [13].

Since the simulation is designed to scale to arbitrary sizes, the simulation works probabilistically rather than operating on a real data set. First, an element is read from the region of memory holding table A. Then, a random element is read from the region of memory storing the hash table, since the hash function will uniformly distribute accesses throughout the table. After the first element is read, a second element is sometimes also read, based on the probability of a hash collision. The collision probability was derived from the expected collision chain length [17]. Finally, an element is written to the result table, and the process repeats itself with the next consecutive element of table A.

6. CACHE DESIGN SIMULATIONS

This section evaluates the five different MMU cache organizations using a wide variety of applications. The TLB miss penalty, structure sizing, and replacement policies are explored. The results show that the unified translation cache with a modified LRU replacement scheme is the best design for the entire range of applications. For the small benchmarks, this cache design is able to reduce the number of memory accesses required per TLB miss from 4 without a dedicated cache to 1.13. It also adapts dynamically to large applications, avoiding the conflicts present in traditional unified caches without the static partitioning required in split caches.

6.1 TLB miss penalty

The purpose of any MMU cache is to lessen the penalty of a TLB miss and the cost of walking the page table. This penalty can be broken down into the number of accesses to the private MMU cache and the number of memory hierarchy accesses required per TLB miss. Without a private cache, there will be four memory hierarchy accesses per walk, one per level. These memory accesses can be further broken down into L2 data cache hits and DRAM accesses, which are far more costly.

6.1.1 Small Memory Applications

Even applications that use a modest amount of memory can have frequent TLB misses. Table 3 shows the frequency of TLB misses for each of the SPEC CFP2006 benchmarks, the SPEC JBB2005 Java server benchmark and the ASCI Sweep3d benchmark. Specifically, the table shows the average number of instructions, program memory accesses and program DRAM accesses (L2 data cache misses) that occur between TLB misses. The number of instructions issued between TLB misses varies from tens of thousands for compute-intensive workloads to hundreds, for data-intensive workloads. For SPECjbb2005, less than five DRAM accesses are made between TLB misses. For data-intensive workloads that may be memory bottlenecked, the DRAM accesses related to page walks are significant.

			DRAM accesses/Walk		
Workload	Ins/Walk	Mem/Walk	2MB	1MB	512KB
bwaves	3637.7	2183.8	102.6	104.4	106.2
gamess	37927.8	16905.0	1.1	1.1	1.1
milc	202.3	83.1	3.9	3.9	4.0
zeusmp	3105.2	562.0	77.0	77.8	79.0
gromacs	25399.1	12025.0	42.2	55.4	69.5
cactus	3916.9	2919.4	28.7	30.2	31.9
leslie3d	4185.2	1679.8	67.5	70.5	72.0
namd	49024.9	18498.7	9.0	12.3	16.2
deal	29235.3	10046.7	12.3	14.5	16.9
povray	38328.8	19498.9	1.9	1.9	1.9
Gems	50817.5	19447.8	1.4	1.4	1.4
tonto	30414.8	13711.6	4.5	7.2	28.8
lbm	1844.5	908.1	97.5	101.7	106.7
sphinx3	1858.2	574.4	26.3	28.6	30.2
(avg)	19992.7	8503.2	34.0	36.5	40.4
specjbb	351.0	162.0	2.4	3.4	4.8
Sweep3d	6098.6	3161.3	81.9	83.9	85.4

Table 3: The frequency of TLB misses for each workload, shown as the number of instructions, memory accesses and DRAM accesses between TLB misses. These results are shown for three different L2 data cache sizes and the TLB configuration described in Section 5.2.

Table 4 compares the behavior of the different MMU caches. For each kind of MMU cache the table shows how many times the MMU cache, the L2 data cache and DRAM are accessed per TLB miss under each of the benchmarks. All caches are using a least-recently-used replacement policy. In these simulations, the unified and path caches have 24 entries and the split caches have 3×24 entries. While there are some outliers, most of the applications exhibit similar behavior.

As a baseline, Table 4 also presents results with no MMU cache. As expected, with no cache there are four memory accesses per walk. Interestingly, there are only 0.15 DRAM accesses per walk for SPEC CFP2006, meaning that there is a 96% hit rate for page table entries in the L2 data cache. This number varies from application to application, but it never drops below 90%. This demonstrates that page table access for these applications has very high reuse.

Adding any MMU cache drops the average number of memory hierarchy accesses per walk from 4.00 to no more than 1.13 (0.99+0.14) for SPEC CFP2006. Note that DRAM accesses only decrease approximately 7%, from 0.15 to 0.14. This means that most of the avoided memory accesses come out of L2 hits, not DRAM accesses. The decrease in TLB miss latency from a MMU cache with these applications comes from the decreased access latency of an MMU cache as compared to the L2 data cache, not an improved hit rate. For Sweep3d, there is so much locality in virtual address use that memory accesses per TLB miss is further reduced to 1.07.

Since these caches do not store L1 page table entries, this result

	No Cache			UPTC (24 entry)			SPTC (3x24 entry)			UTC (24 entry)			STC (3x24 entry)			TPC (24 path)		
	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM
bwaves	-	3.72	0.28	2.98	0.82	0.25	2.99	0.80	0.24	1.06	0.81	0.26	1.03	0.80	0.24	1.03	0.80	0.23
gamess	-	3.95	0.05	2.94	1.13	0.05	2.94	1.13	0.05	1.15	1.13	0.05	1.15	1.13	0.05	1.15	1.13	0.05
milc	-	3.90	0.10	3.00	0.91	0.10	3.00	0.91	0.09	1.01	0.91	0.09	1.00	0.91	0.09	1.00	0.91	0.09
zeusmp	-	3.81	0.19	2.99	0.85	0.17	2.99	0.85	0.17	1.02	0.85	0.17	1.02	0.85	0.17	1.02	0.85	0.17
gromacs	-	3.80	0.20	2.96	0.91	0.20	2.96	0.91	0.20	1.08	0.90	0.20	1.08	0.91	0.20	1.08	0.91	0.20
cactus	-	3.84	0.16	2.96	1.12	0.16	2.99	0.98	0.15	1.23	1.10	0.15	1.13	0.98	0.15	1.13	0.99	0.15
leslie3d	-	3.79	0.21	2.98	0.88	0.20	2.99	0.85	0.19	1.06	0.88	0.20	1.04	0.85	0.19	1.04	0.86	0.19
namd	-	3.86	0.14	2.88	1.15	0.13	2.88	1.14	0.13	1.23	1.15	0.13	1.23	1.14	0.13	1.23	1.14	0.13
deal	-	3.89	0.11	2.93	1.07	0.11	2.93	1.07	0.11	1.15	1.07	0.11	1.15	1.07	0.11	1.15	1.07	0.11
povray	-	3.92	0.08	2.92	1.10	0.08	2.92	1.10	0.08	1.15	1.10	0.08	1.14	1.10	0.08	1.14	1.10	0.08
Gems	-	3.93	0.07	2.89	1.18	0.07	2.89	1.18	0.07	1.20	1.18	0.07	1.20	1.18	0.07	1.20	1.18	0.07
tonto	-	3.93	0.07	2.94	1.07	0.08	2.94	1.07	0.08	1.12	1.07	0.08	1.12	1.07	0.08	1.12	1.07	0.08
lbm	-	3.79	0.21	2.99	0.83	0.19	2.99	0.83	0.18	1.02	0.83	0.19	1.01	0.83	0.18	1.01	0.83	0.18
sphinx3	-	3.76	0.24	2.99	0.80	0.23	2.99	0.79	0.23	1.02	0.80	0.23	1.02	0.79	0.23	1.02	0.79	0.23
(avg)	-	3.85	0.15	2.95	0.99	0.14	2.96	0.97	0.14	1.11	0.98	0.14	1.09	0.97	0.14	1.09	0.97	0.14
specjbb	-	3.83	0.17	2.98	0.97	0.17	3.00	0.93	0.17	1.14	0.97	0.17	1.11	0.93	0.17	1.11	0.95	0.17
Sweep3d	-	3.77	0.23	2.95	1.01	0.22	2.98	0.87	0.20	1.13	0.94	0.21	1.07	0.87	0.20	1.07	0.88	0.20

Table 4: The number of caching structure accesses (S), L2 data cache hits (L2), and DRAM accesses (DRAM) per TLB miss for the various LRU cache designs over the SPEC CFP2006, SPECjbb2005 and Sweep3d benchmarks.

is very close to the minimum number of memory accesses per TLB miss of 1.00. On these applications, all the caches have similar hit rates. In nearly all TLB misses, all the MMU caches can provide the physical page number of the L1 page table page without having to do any memory accesses. From here, the L2 cache still provides most of the L1 entries at a hit rate of 88% (only 0.14 of the 1.13 memory accesses are DRAM accesses). These hits come from the fact that the L2 cache is much larger than the MMU caches and can store eight entries in a single cache line.

One benchmark, `tonto`, has more DRAM accesses when an MMU cache is used. However, the rounding in Table 4 exaggerates this effect. The actual increase in DRAM accesses is only 0.001 per TLB miss. The MMU cache changes the access pattern to the L2 data cache, so page table entries may be replaced sooner than they would be without an MMU cache, slightly increasing DRAM accesses.

SPECjbb has low address locality at a page and cache line granularity compared to other small memory applications leading to high TLB and cache miss rates. However, there is still enough locality in upper level bits of the address to allow reasonably high MMU cache hit rates. On this workload, 2.87 of the three upper level page table entries are still served by the MMU cache, leaving 0.99 and 0.14 accesses for the L2 data cache and DRAM respectively. These 0.14 DRAM accesses per TLB miss are significant compared to the 3.4 DRAM accesses between TLB misses that come from program execution itself.

Since these caches all provide high hit rates, their primary difference is in the number of accesses to the cache required per walk. Since the translation and path cache search for L2 entries first, they are typically only accessed one time per TLB miss. This reduces both latency and power consumption. The page table caches are accessed an average of three times per walk, since they must walk down the page table. If the size of the virtual address space is expanded by adding an additional level, this penalty will increase.

6.1.2 Large Memory Applications

In contrast to the results for small memory applications, the different MMU caches have substantially different hit rates for applications with random access patterns over gigabytes of virtual memory, such as an in-memory database hash join or HPCC RandomAccess. For these applications, the reuse of lower level page table entries is low, and there are many upper level page table entries in use. Consequently, caching as many L3 entries as possible, each of which covers a 1GB region of virtual memory, is critical,

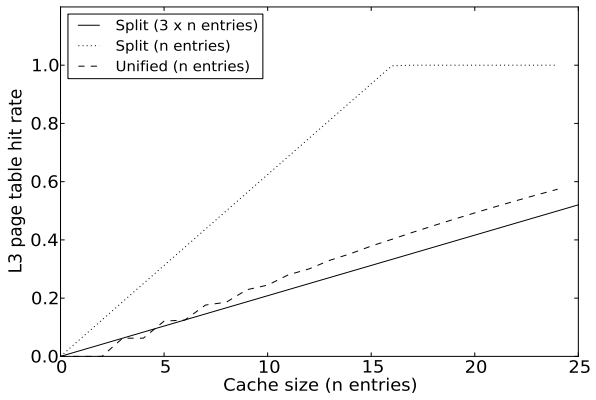

Figure 8: Hit rate compared for the caches with the database join simulation using a 16GB hash table. The different split and unified designs have equivalent coverages.

but caching L2 entries is of little or no benefit. For a unified cache, a TLB miss that hits on an L3 entry but not an L2 entry will load a new L2 entry into the cache. With LRU replacement, the cache will have about the same number of L2 and L3 entries. The TPC also keeps track of an L2 entry for each L3 entry stored, but it does so in the same associative entry. Therefore, for such applications, a TPC of n paths is roughly equivalent to a UPTC of $2 \cdot n$ entries. When the TPC becomes large, there is the possibility that different stored paths will contain the same L3 entry, reducing hit rate. The split caches do not exhibit this effect, so once the cache is large enough to hold all L3 entries in use, there is a 100% hit rate.

A database hash join running over a 16GB region (Figure 8) demonstrates this scaling. Here, to have a 50% hit rate on the L3 page table, a 22 entry unified cache is required, while only an 11 path TPC, or a 3×11 entry STC is required.

An application using many terabytes of virtual memory would only have high reuse on L4 page table entries, causing 2/3 of unified cache entries to be wasted, instead of just 1/2 for the 16GB application. This is because for every L4 entry stored, an L3 and an L2 entry are also stored, which are effectively wasted. This causes the TPC to be equivalent to a unified cache of three times the size for such workloads.

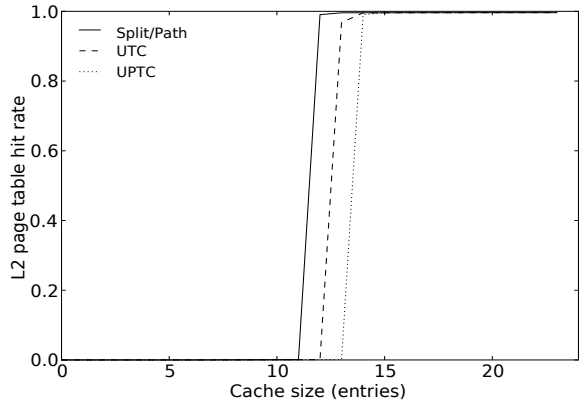

Figure 9: Hit rate for the L2 table for ASCI Sweep3d. This application is very sensitive to sizing. Note that size represents per-level size (all three levels hold n entries) for the split cache included in Figures 9-11.

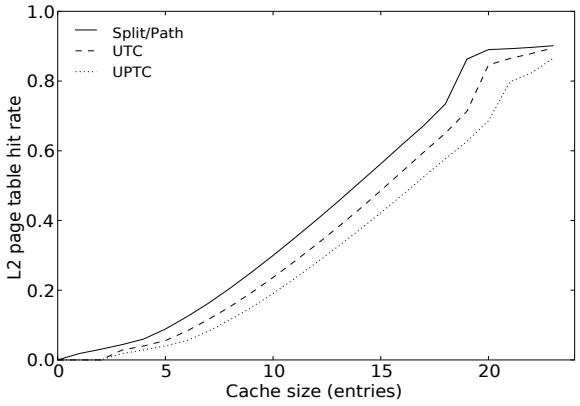

Figure 10: Hit rate for the L2 table alone for the `CactusADM` component of SPEC CPU2006.

Cache	LRU	Random	Increase
UPTC	0.61	1.00	63%
UTC	0.53	0.79	49%
TPC	0.51	0.65	28%
SPTC	0.51	0.64	25%
STC	0.51	0.63	23%

Table 5: The average number of misses per walk for random and LRU replacement, normalized to Random UPTC (lower is better), and the relative increase in misses using random replacement over LRU replacement.

6.2 Sizing considerations

Appropriate sizing of caches is critical for hit rate in many applications. The ability to not store levels that are skipped in the translation cache allows them to be smaller for a given hit rate. Additionally, the fixed allocation of entries for each level in the split cache designs demands that all levels be large to properly adapt to differing workloads. The TPC and unified caches dynamically allocate entries and adapt well.

ASCI Sweep3d operates on a set of different memory regions. When the first is processed, it moves to the next, and so on. After processing the last region, the program wraps around to the first, and the cycle repeats. If the cache is not large enough to hold all the regions, entries corresponding to earlier regions are pushed out before they are used again, and hit rate is very poor (Figure 9). Since upper levels of the page table are skipped in the translation cache, the UTC can be slightly smaller than the UPTC.

For the small memory applications, there are relatively few upper level page table entries that are in use. As a result, the hit rate of the unified caches holding n entries only slightly trails that of the split caches, which hold a total of $3 \times n$ entries. In this example (Figure 10), a unified page table cache holding 23 entries is equivalent to a split cache holding $3 \times 19 = 57$ entries. Only four entries from the unified cache are stored in upper levels of the split caches. These entries are combined with lower level entries in the TPC, allowing a 19 path TPC to be equivalent to the 23 entry unified cache.

While these applications use more lower level entries than upper level entries, having small upper-level caches in a split cache dramatically reduces hit rate for large applications. If the split cache is reduced in size to 3×8 entries to match the total size of the unified cache, the L3 table hit rate in the database benchmark is reduced from 99% to 44%. Skewing the distribution of entries from higher levels to lower levels will further impact hit rate. Therefore, it is imperative that all levels of a split cache be large, resulting in considerable area overhead.

6.3 Replacement policy

In the unified caches, entries of high reuse (upper level entries) are mixed with entries of lower reuse (lower level entries). This causes the cache to be relatively sensitive to the LRU replacement policy which ensures that frequently accessed components (L4 and L3 entries) are not evicted. In the path and translation caches, these upper level entries are skipped, so they do not need to be protected.

Table 6.3 shows the impact of using a random replacement scheme on the number of MMU cache misses (measured by the number of required memory accesses to locate upper level page table entries). While all structures have a lower hit rate using a random replacement scheme, the unified designs are much more reliant on an LRU replacement scheme.

The primary problem with the unified cache designs for the large applications is that entries with high reuse are evicted to make room for entries of low reuse. For example, in the database join the LRU unified caches hold a relatively useless L2 entry for every useful L3 entry in the cache. If a content-aware replacement scheme is used, this problem can be significantly reduced. If the Greedy Dual algorithm is implemented in the UTC instead of using LRU, the size required for 90% L3 entry hit rate in the 16GB database join is reduced from 52 to 30 entries.

Using our modified LRU replacement scheme (described in Section 4.2) with fixed insertion points, this algorithm actually has a higher hit rate than Greedy Dual for the database benchmark. Only 23 entries are required for 90% hit rate. However, the fixed insertion point for L2 entries reduces hit rate significantly for some other applications, such as Sweep3d. The L2 entries used by Sweep3d, if they are inserted near the least recently used position, are replaced before they are reused resulting in a near zero hit rate.

Using the variable insertion-point LRU scheme solves this. For SPEC CFP2006, SPECjbb2005 and Sweep3d, VI-LRU has a hit rate that is equal to or slightly greater than standard LRU. For the database join, where there are many L2 entries to cache, VI-LRU adapts far better than other replacement schemes. Only 16 entries are required for 90% hit rate in the join benchmark, as opposed to 52 for standard LRU (Figure 11). The VI-LRU cache nearly eliminates the conflict between levels seen in the unified cache, and

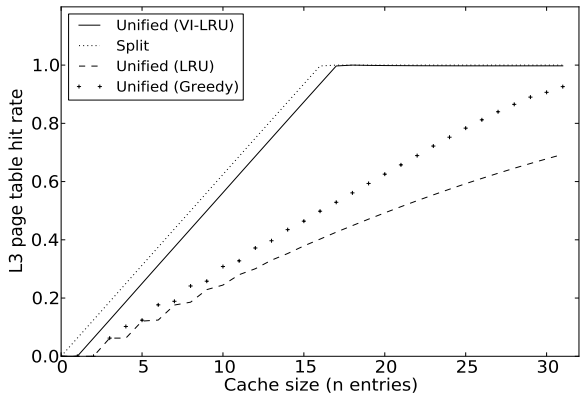

Figure 11: An n entry unified translation-cache with VI-LRU has nearly the same hit rate as a $3 \times n$ split cache.

an n entry VI-LRU UTC has nearly the same hit rate as a $3 \times n$ entry split translation cache.

7. ALTERNATE PAGE TABLE FORMATS

We compared the memory access behavior of the cached page table with its biggest rivals, hash-table based Inverted Page Tables and direct-mapped Translation Storage Buffers. These structures are attractive since they contain only one level, and are therefore insensitive to address space size. However, the unavoidable presence of hash and structural collisions, low access locality and their inability to handle multiple page sizes efficiently cause them to require far more memory accesses than a cached radix table.

7.1 Hashed page tables

We have shown that MMU caches can significantly reduce the overhead of using a radix tree page table, however the possibility remains that the radix tree page table itself should be replaced. The traditional competitor to the radix tree page table is the inverted page table, which uses a hash table to store a large and sparsely used address space efficiently [15]. These designs are usually seen as superior to a multi-level table, because they only need to be referenced once, whereas the radix table requires one access per level. However, hash collisions are unavoidable, so many accesses may require more than one reference to follow a collision chain.

Additionally, we have shown in this paper that MMU caches can reduce the number of memory accesses per walk to nearly one as well. To compare the cached radix table against an inverted page table, a simulator was constructed that maintains and references a hash table storing all the memory locations used during a process' lifetime. The hash table used closely models that used by the Intel Itanium [4]. The number of accesses to this table were counted, as well as the number of such accesses that hit in the L2 data cache.

When the hash table contains twice as many buckets as there are pages to store, the hash table walker references approximately 1.2 locations per TLB miss, *regardless of benchmark or access pattern*. This number comes from the average length of a collision chain, which is a function only of the fullness of the hash table if a sufficiently uniform hash function is used [17]. This number compares poorly to the average number of L2 and memory accesses required per walk of the SPEC CFP2006 applications using page table caching of 1.13. While the hashed page table is insensitive to address space size, it is unable to take advantage of the great locality seen in virtual address space usage like MMU caches can.

	Uncached		**TPC**	
Page Size	4KB	2MB	4KB	2MB
L2 Hits	2.90	2.92	1.11	1.15
DRAM	1.10	0.06	1.09	0.06

	IPT (1)		**IPT (2)**		**IPT (16)**	
Page Size	4KB	2MB	4KB	2MB	4KB	2MB
L2 Hits	0.01	0.00	1.16	1.16	0.55	0.54
DRAM	1.29	1.29	1.14	1.14	1.48	1.49

Table 6: L2 hits and DRAM accesses to the page table per walk for a radix tree page table for the 16GB database join benchmark. Results are shown for an uncached radix tree, a cached radix tree, and a half-full inverted page table with various numbers of clustered translations per tag.

Compounding this issue is the fact that references into the hash table show no spatial locality. Whereas consecutive pages in virtual memory are usually mapped by consecutive entries in the radix table, they are not usually mapped by consecutive entries in a hash table. Since there is usually locality in the access pattern of L1 page table entries in a radix table, these entries are much more effectively cached by the L2 data cache than the entries of the hashed table are. For the SPEC CFP2006 applications examined earlier, only 44% of the 1.2 accesses/walk are served by the L2 data cache. Overall, the inverted page table increases the number of DRAM accesses per walk by *over 400%*.

Spatial locality can be increased by storing multiple adjacent translations with a single tag, as used in *clustered page tables* [23]. This technique also reduces the overhead (virtual address tag and chain pointer) for the hash table. However, for this technique to be effective each virtual tag must be associated with many translations. This means that some translations will need to load multiple cache lines. Additionally, the frequency of hash collisions is not reduced over a standard inverted page table.

Even if the virtual address space is used without locality, as in the database join, the radix tree page table still requires fewer DRAM accesses than a hashed page table. Table 6 shows the memory use per TLB miss for a join using a 16GB hash table. With 4KB pages, the radix tree page table requires fewer DRAM accesses/walk than the inverted page table until a 48GB inverted page table is used.

For this application, clustering does improve L2 cache hit rate since the page table is smaller. However, the tag and data often lie in different cache lines, which requires an increase in the total overall memory references required to perform a translation. Matching the total cluster size to the size of a cache line improves this. However since cache line size may change from implementation to implementation, the appropriate cluster size may change as well.

For larger applications such as this, large page support becomes important. When it comes to supporting the simultaneous use of multiple page sizes, radix trees have an advantage over inverted page tables. With the radix tree, if 2MB pages are used for mapping most of the virtual address space, the entire page table can be cached in the L2 data cache, because the 2MB page mapping takes the place of an L2 entry in the page table and eliminates the need for an entire L1 page table page (see Figure 2). This reduces the number of DRAM accesses per walk dramatically for the radix table designs and also the number of overall memory hierarchy accesses to below that seen in the inverted page table, as shown in Table 6.

In contrast, the simultaneous use of large and small pages does not reduce the size of an inverted page table, and so its memory accesses do not change. In essence, the hash function must take into account the size of the virtual page, but it cannot know the page's

size *a priori* if multiple page sizes are in use. Consequently, for a large page, the inverted page table must still have a page table entry corresponding to each of the small pages that make up the large page. Each of these page table entries will, however, designate the mapping as part of a large page, and the TLB will be loaded with a single large page mapping.

7.2 Translation Storage Buffers

The SPARC architecture has traditionally handled TLB misses in software. To accelerate TLB misses, the processor supports a software-managed, direct-mapped cache of translations called the *Translation Storage Buffer* [5]. On a TLB miss, the CPU derives an index from the lower entry bits in the virtual address and checks to see if a corresponding entry is present in the TSB. Although earlier processors performed this TSB lookup in software, some current processors implement it in hardware. Like the inverted page table, a TSB entry stores a tag (a virtual page number) and a translation (a physical page number). Unlike the inverted-page table, there is no chaining. If a translation is not present in the TSB, a software fault occurs.

To compare this design to the radix-tree design, a TSB simulator was also developed. Like the inverted page table simulator, the TSB simulator counts L2 and DRAM accesses per TLB miss. In addition, the TSB simulator also counts software faults that occur when a translation is not present in the cache. Traces are simulated in a two-pass manner. The first pass populates the cache with the translations present in the trace. The second pass actually simulates accesses to the cache, counting hits and misses. This ensures that only conflict misses are counted, providing a lower-bound for cache misses.

Our results show that the TSB uses the L2 data cache poorly as compared to the radix-tree. For example, the `zeusmp` component of the SPEC CFP benchmark generates 0.058 DRAM accesses per TLB miss using the radix-tree with an MMU cache whereas the TSB required 0.078 DRAM accesses per TLB miss. This increase is due solely to the larger size of a TSB entry as compared to a radix tree page table entry. The TSB entry contains a tag and data, whereas the radix-tree only needs to hold data. This increased size reduces the number of entries that will fit in a single L2 cache line from eight to four.

In addition, the TSB also generates 0.024 software faults per TLB miss using the current architectural maximum size of 1 megabyte. These are likely to be extremely expensive, generating not only data cache misses, but also instruction cache misses. While increasing the size of the L2 data cache would reduce the number of L2 cache misses, it would not reduce the number of software faults.

8. RELATED WORK

Some early work on caching page table entries was done before the introduction of AMD and Intel's MMU caches. This work was targeted at accelerating software TLB miss handling. Bala *et al.* introduced a software cache for page table entries [7]. This cache is read by the software page fault handler and manages entries in physical memory to avoid cascading TLB misses that come from reading page table entries in virtual memory space. Wu and Zwanepoel expanded this to a hardware/software design [24]. They propose a single level translation cache to handle only L2 entries. If a translation hits in their structure, the MMU loads the L1 entry directly, as in the caches presented in this paper. If the translation misses, a software fault is triggered.

In a virtualized system using nested paging, both the guest virtual machine and the underlying virtual machine monitor have their own page table. In effect, the virtual machine monitor's page table is used to create a private guest physical address space for the virtual machine. Thus, the guest's page table is used to translate from virtual addresses to guest physical addresses, and the virtual machine monitor's page table is used to translate from guest physical addresses to host physical addresses. Nested paging with radix tree-based page tables leads to a two-dimensional page table walk because every access to the guest's page table during a page walk may result in a page walk on the virtual machine monitor's page table.

Bhargava *et al.* showed that an extended version of AMD's Page Walk Cache could effectively cache most of the upper level page table entries in both the guest's page table and the virtual machine monitor's page table [9]. In addition, they proposed the introduction of a *Nested TLB* (NTLB) that caches guest physical to host physical translations. In effect, a hit in the NTLB allows the two-dimensional page walk to *skip* the page walk on the virtual machine monitor's page table. Thus, if every access to the guest's page table hits in the NTLB, then the number of accesses to the Page Walk Cache and the memory hierarchy is the same as it would be for native execution.

The translation and path caches presented in this paper could also be extended to support nested paging. Moreover, the NTLB is not inextricably tied to the Page Walk Cache or page table caches in general. A NTLB could be beneficially combined with translation and path caches. A NTLB hit allows the two-dimensional page walk to skip the entire page walk on the virtual machine monitor's page table for a single guest page table access, but not the accesses to the upper levels of the guest's page table. This requires a translation or path cache. Moreover, a translation or path cache could accelerate page walks on the virtual machine monitor's page table when a NTLB miss occurs.

Talluri and Hill recognized the importance of efficient use of spatial locality in the page table and developed their clustered and subblock page tables to increase the spatial locality found in inverted page tables [22, 23]. We simulate a similar system in Section 7.1 and compare it to both traditional inverted page tables and cached radix tree tables.

McCurdy *et al.* previously investigated the importance of the L2 data cache in storing page table entries [19]. They show that applications that use large pages can show improved performance even in the face of decreased TLB hit rates. This is due to the shallower, and therefore smaller, page table when using large pages.

Jacob and Mudge [15] show that MMU related memory accesses can cause higher than expected cost due to user program and data being evicted by page table entries. This effect further emphasizes the importance of efficient storage of page table entries.

Saulsbury *et. al.* propose a prefetching scheme for TLBs that preload pages based recently accessed pages [20]. Unlike the techniques presented in this paper, their techniques require page table modification. More recent work [16, 10] has proposed architecturally independent prefectching techniques based on access patterns and inter-core cooperation. These techniques all focus on reducing the frequency of TLB misses while our work focuses on reducing the cost of servicing a TLB miss. Both techniques could easily be combined.

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 [18]. 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.

9. CONCLUSION

Since the x86 architecture began using a radix tree page table for address translation in the 80386, the depth of the page table has increased by one level with each passing decade. Unfortunately, without an MMU cache, the page table walk for address translation requires a memory reference for each level of the radix tree. Therefore, MMU caches have become critical components of current and future x86 processors. This paper has presented a quantitative and qualitative comparison of the design space of such MMU caches, including three new designs. While this paper has focused on x86 processors, the results should apply generally to any architecture that uses a radix tree page table.

While AMD and Intel have both developed MMU caches for their microprocessors, this paper has introduced a unified translation cache with a modified LRU replacement scheme that is superior to both existing devices. It adapts well to varying workloads, unlike a split translation cache, as implemented in Intel's Paging Structure Cache. It also prevents conflict between entries of low and high reuse, unlike the LRU unified page table cache, as implemented in AMD's Page Walk Cache.

This paper has also shown that MMU caches dramatically change the trade-offs in page table design for large address spaces. Radix tree page tables make more effective use of the processor's L2 cache than either inverted page tables or translation storage buffers. Radix tree page tables have a smaller page table entry size, because both inverted page tables and translation storage buffers must include a tag in the page table entry. Thus, the L2 cache is able to hold more page table entries from the radix tree, increasing its coverage and reducing DRAM accesses. So, while these alternate structures are superior to the radix tree page table on its own for large address spaces, a well designed MMU cache renders the radix tree organization far superior.

10. REFERENCES

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

[2] *The ASCI sweep3d Benchmark Code*.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Part 1*.

[4] *Intel Itanium Architecture Software Developer's Manual - Volume 2: System Architecture, Revision 2.2*.

[5] *UltraSPARC III Cu User's Manual*.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[7] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 18, Berkeley, CA, USA, 1994. USENIX Association.

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

[9] 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.

[10] 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.

[11] J. J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical Report 05-544, University of Tennessee - Knoxville, http://icl.cs.utk.edu/hpcc/, 2005.

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

[13] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

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

[15] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 295–306, New York, NY, USA, 1998. ACM.

[16] 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.

[17] D. E. Knuth. *The Art of Computer Programming 3. Sorting and Searching: The Classic Work Newly Updated and Revised*. Addison-Wesley Longman, Amsterdam, 2. a. edition, 1998.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] M. Wu and W. Zwaenepoel. Improving tlb miss handling with page table pointer caches. Technical Report TR97-296, Rice University, 1996.

[25] N. Young. On-line caching as cache size varies. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 241–250, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.