Large-Reach Memory Management Unit Caches

Coalesced and Shared Memory Management Unit Caches to Accelerate TLB Miss Handling

Abhishek Bhattacharjee
Department of Computer Science, Rutgers University
abhib@cs.rutgers.edu

ABSTRACT

Within the ever-important memory hierarchy, little research is devoted to Memory Management Unit (MMU) caches, implemented in modern processors to accelerate Translation Lookaside Buffer (TLB) misses. MMU caches play a critical role in determining system performance. This paper presents a measurement study quantifying the size of that role, and describes two novel optimizations to improve the performance of this structure on a range of sequential and parallel big-data workloads. The first is a software/hardware optimization that requires modest operating system (OS) and hardware support. In this approach, the OS allocates page table pages in ways that make them amenable for coalescing in MMU caches, increasing their hit rates. The second is a readily-implementable hardware-only approach, replacing standard per-core MMU caches with a single shared MMU cache of the same total area. Despite its additional access latencies, reduced miss rates greatly improve performance. The approaches are orthogonal; together, they achieve performance close to ideal MMU caches.

Overall, this paper addresses the paucity of research on MMU caches. Our insights will assist the development of high-performance address translation support for systems running big-data applications.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures; C.1.0 [**Processor Architectures**]: General

Keywords

Virtual Memory, Memory Management Units, Translation Lookaside Buffers

1. INTRODUCTION

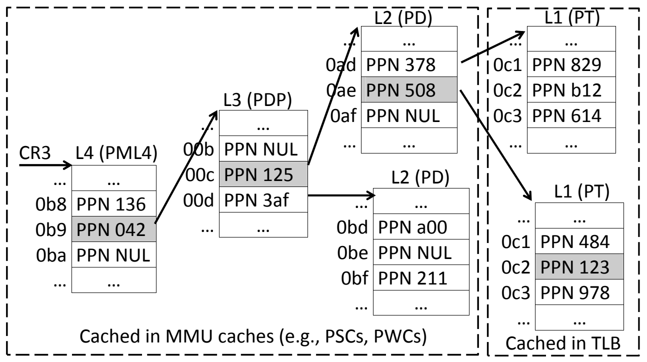
As the computing industry enters the era of big data, fields such as scientific computing, data mining, social networks, and business management depend on processing massive, multidimensional data-sets. Designers have responded to this shift by proposing hardware that reflects this change. For example, recent research has studied microarchitecture [12, 20], caches and memory [13, 21, 26], and accelerators [36] appropriate for software with massive data-sets.

In this context, evaluating virtual memory for big data is warranted. Virtual memory is a powerful abstraction that enhances programmer productivity by automating memory management. At its core, it requires translation of program-level virtual addresses to system-level physical addresses. Processor vendors provide per-core Translation Lookaside Buffers (TLBs) to cache commonly used address translations or page table entries (PTEs). Historically, the programmability benefits of virtual memory have outweighed TLB miss overheads, which typically degrade system performance by 5-15% [2, 8, 10, 18, 23, 28].

Unfortunately, emerging big-data workloads (with massive memory footprints) and increased system consolidation (i.e., virtualization) degrade TLB performance in two ways. First, TLBs struggle to map the full working set of applications, increasing miss rates. Second, larger working sets usually lead to larger page tables, which become increasingly hard to cache, increasing TLB miss penalties. For example, ×86 systems use four-level page tables. A TLB miss hence requires four long-latency memory accesses to these page tables and usually finds them in the last-level cache or main memory [2, 6]. Recent work shows that, as a result, TLB miss overheads for modern systems can be as high as 5-85% of total system runtime [4].

In response, some processor vendors (e.g., Intel and AMD) design structures that not only cache PTEs from the last level of multilevel radix tree page tables (in the TLB), but also cache entries from higher levels of the tree in small per-core Memory Management Unit (MMU) caches [14, 5]. MMU caches are accessed on TLB misses; MMU cache hits enable skipping multiple memory references in the page table walk (reducing the entire walk to just a single memory reference in the best case). Unfortunately, MMU caches have not been deeply-studied (with the first major study appearing only recently [2]).

This paper presents the first quantification of MMU cache performance on a real system running sequential and parallel workloads representative of emerging software. We show that current system performance is degraded because of MMU cache misses. In response, we study low-overhead and readily-implementable MMU cache optimizations. Our optimizations can be classified into those that require hardware and software support, and those with hardware-only changes. The two techniques are orthogonal and while effective individually, in tandem they achieve performance close



PSC: L2

L4/L3/L2 Ind.	Page
b9/0c/ae	508
…	…

PSC: L3

L4/L3 Ind.	Page
b9/0c	125
…	…

PSC: L4

L4 Ind.	Page
b9	042
…	…

TLB

Virtual Page	Physical Page
b9/0c/ae/c2	123
…	…

Figure 1: 64-bit x86 page table walk for virtual address `(0b9, 00c, 0ae, 0c2, 016)`. TLBs cache L1 PTEs and MMU caches store L2-L4 PTEs. Conventional hardware caches can store all entries.

to ideal MMU caches. Specifically, our contributions are:

First, we use on-chip event counters to profile MMU cache overheads for parallel, sequential, and server workloads on a real system. Non-ideal MMU caches degrade performance by as much as 10-17%. Surprisingly, we find (because we use a real system) that real MMU cache sizes differ from those assumed in past simulation-based studies [2].

Second, we propose a hardware/software coalescing technique to reduce MMU cache misses. In this approach, we design low-overhead operating system (OS) page table allocation mechanisms that maximize coalescing opportunity. We then propose complementary enhancements to standard MMU caches to detect and exploit coalescing patterns, improving performance by as much as 10-12%. Interestingly, past OS work on page allocation focuses exclusively on large data pages [24, 32]; we show that careful allocation of the page table itself can yield significant performance benefits.

Third, we propose replacing per-core MMU caches with a unified MMU cache shared by all cores. While this centralized structure suffers higher access latencies, its higher hit rate more than compensates, providing overall performance improvements of 10-12%. One might expect only parallel programs (with threads sharing common data structures) to benefit from shared MMU caches; we find, however, benefits for multiprogrammed sequential applications too. In these cases, a central structure better allocates resources to applications on demand, rather than statically partitioning total MMU cache space among cores, regardless of their needs.

Finally, we combine coalescing and sharing, achieving close to ideal performance (as high as 17%), while remaining robust to increased access latencies. We make two additional observations. First, unlike coalesced and shared TLBs, coalesced and shared MMU caches consistently improve and *never* degrade application performance for our profiled applications. Second, coalesced and shared MMU caches achieve performance and hardware complexity benefits over naively increasing per-core MMU cache size.

Overall, we set the foundation for simple MMU cache optimizations, which will become particularly compelling as software demands ever-increasing quantities of data.

2. BACKGROUND

Figure 1 illustrates how MMU caches store parts of the page table, and how they accelerate TLB misses. For illustrative purposes, we show an ×86 radix tree multilevel page table though MMU caches are useful for any radix tree page table (e.g., ARM, SPARC). TLB misses prompt a hardware page table walk of a four-level page table with the Page Map Level 4 (PML4), Page Directory Pointer (PDP), Page Directory (PD), and Page Table (PT). To ease terminology, we henceforth refer to them as L4, L3, L2, and L1 tables.

On a TLB miss, a hardware page table walker first splits the requested virtual address into four 9-bit indices used to index the various levels, and a page offset. The example of Figure 1 assumes a TLB miss for virtual address `0x5c8315cc2016`, which has the (L4, L3, L2, L1) indices of `(0b9, 00c, 0ae, 0c2)` and a page offset of `(016)`. Then the walker reads the `CR3` register, which maintains the base physical address of the L4 page table. This page table page maintains 512 8-byte PTEs, each of which points to a base physical page address of an L3 page table page. The walker adds an offset equivalent to the L4 index (`0b9`) to the base physical address in `CR3` to select an L4 PTE. This points to an L3 page table page base, from which the L3 index (`00c`) is offset to select the desired pointer to the L2 page table page base. This process continues until the L1 PTE, which maintains the actual translation, is read.

TLBs cache PTEs from the L1 page table levels. Small per-core hardware MMU caches, in comparison, store L4, L3, and L2 PTEs. The motivation to do so is that upper-level PTEs are frequently reused because they map a bigger chunk of the address space. For example, ×86 L1 PTEs map 4KB base pages while L2, L3 and L4 PTEs map 2MB, 1GB, and 512GB address chunks. Therefore, a TLB miss first checks the MMU cache for L2, L3, and L4 PTEs. Low-latency MMU cache hits avert expensive lookups for these levels in the memory hierarchy. Past studies show that without MMU caches, PTEs are typically found in the last-level cache (LLC) or main memory [2, 3, 5].

There are multiple implementation possibilities for MMU caches. Intel uses *Paging Structure Caches* (PSCs) [14], which are indexed by parts of the virtual address. Figure 1 shows PSC contents after accessing virtual address `(0b9, 00c, 0ae, 0c2, 016)`. Separate PSCs are maintained for each page table level; L4 entries are tagged with the L4 index, L3 entries are tagged with both L4 and L3 indices, while L2 entries are tagged with L4, L3, and L2 indices. On a TLB miss, all PSC entries are searched in parallel. The MMU selects the entry matching the longest prefix of the virtual address, allowing the page table walk to skip the maximum number of levels. Once the longest prefix match is found, the remainder of the page table walk occurs by accessing the memory hierarchy. For example, searching for `(0b9, 00c, 0ae, 0c2, 016)` yields an L2 PSC match. Therefore, only the L1 entry (`508` concatenated with index `0ae`) is looked up in the hardware caches (and possibly main memory). An L3 match, however, requires both L2 and L1 lookups.

While our work primarily uses PSCs (due to space constraints), our insights are equally applicable (and we study) other designs too. An important alternative design is AMD's *Page Walk Cache* (PWC) [5]. Unlike PSCs, PWCs are tagged with physical addresses; as a result, the page table levels must be looked up sequentially in the PWC (i.e., the L4 PTE is first looked up, then the L3, and L2 PTEs).

3. RELATIONSHIP TO PRIOR WORK

There is a distinct paucity of past work on MMU caches.

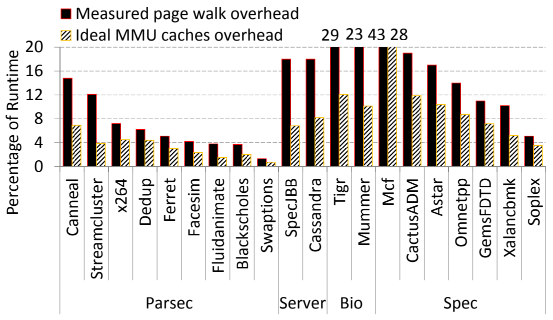


Figure 2: Percentage of runtime devoted to handing TLB misses for a real system compared to the overheads of a perfect MMU cache.

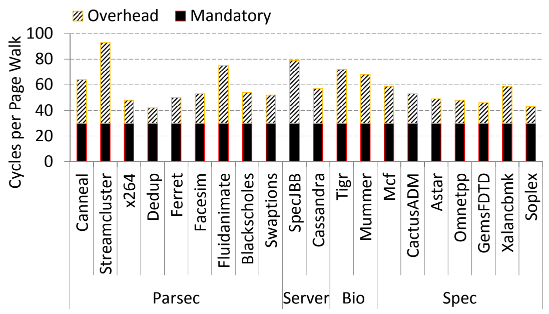


Figure 3: Average number of cycles per page table walk. Note that the best case walk involves a cache hit for the L1 PTE reference.

Recent work [2] sheds light on some of their design aspects. We are, however, the first to quantify the performance overhead of MMU caches on emerging big data applications.

MMU cache coalescing is partly inspired by TLB coalescing [25]. In our work, the OS detects when successive PTEs from page table level L_n are used. When this happens, the OS attempts to allocate the L_{n-1} page table pages pointed to by the L_n PTEs to consecutive physical pages. In tandem, complementary hardware detects instances of such behavior and exploits them to reduce MMU cache misses.

Although MMU caches, like coalesced TLBs, conceptually exploit contiguity, they do so by exploiting allocation of the page table itself rather than the data. This leads to key differences. Unlike coalesced TLBs, coalesced MMU caches require OS modifications to guide page table page allocation. This is an interesting counterpoint to all traditional work on page allocation for TLB performance, which focuses exclusively on data page allocation (e.g., superpages [24, 32]). As such, achievable page table page contiguity amounts and their overheads are completely different than all past data page allocation work. Therefore, coalesced MMU caches require an entirely different design space exploration.

Similarly, shared MMU caches, though partly inspired by shared-last level TLBs [6], have completely different design constraints and characteristics. Unlike TLBs which map only 4KB pages, MMU cache entries map 512GB, 1GB, or 2MB sizes, depending on whether the entry is from the L4, L3, or L2 levels. Since MMU cache entries map much larger regions of memory, they exhibit completely different sharing patterns. Also unlike TLBs, each MMU cache entry maps different address space amounts. Shared MMU cache designs must be made aware that the miss penalties and sharing potential for each level is different. Finally, MMU caches are orders of magnitude smaller than TLBs, meaning that sharing and eviction play vastly different roles.

Finally, combining coalescing and sharing on MMU caches actually outperforms coalesced and shared TLBs in many cases. Because MMU caches store a larger chunk of the address space than TLBs, shared MMU caches are smaller and enjoy lower access penalties. Coalescing further increases shared MMU cache hit rates, completely outweighing additional access times.

4. MMU CACHE OVERHEADS

We begin by measuring MMU cache overheads on a real system to quantify opportunities for improvement.

4.1 Experimental Platform

We use a 64-bit Intel Core i7 with 8 cores and 8 GB memory. Each core has four-way set-associative L1 data (64-entry) and instruction TLBs and unified L2 TLBs (512-entry). The LLC is 8MB. We run Linux 2.6.38 (with transparent hugepages [1] so that large pages are available) and use three hardware event counters to (1) measure the number of cycles spent on page table walks; (2) the number of page table walks; and (3) the number of cycles spent on the application runtime.

The Core i7 uses PSCs; because Intel manuals do not detail PSC organizations, we ran microbenchmarks inspired by past work [34] to deduce their sizes. We found that the L4, L3, and L2 STCs are 2-entry, 4-entry, and 32-entry (4-way associative). Past studies assume that all PSC sizes are the same, giving considerably different results [2].

We also quantify ideal MMU cache performance to assess upper-bounds for our optimizations. Ideal MMU caches skip L4, L3, and L2 PTE references in the walk, leaving only an L1 PTE memory reference (which may be found in either the standard hardware caches or main memory). Using microbenchmarks, we see that L1 PTE references are overwhelmingly found in the LLC or main memory, matching prior work [2, 6]. Therefore, ideal MMU caches (and ideal page table walks) require only L1 PTE accesses, which all hit in the LLC (a 30-cycle penalty [15]).

4.2 Evaluation Workloads

We study the behavior of MMU caches across a range of parallel, sequential, and server workloads. We pick workloads representative of emerging recognition, mining, and synthesis domains from the Parsec suite [9], which we run with *Native* inputs on all 8 cores. Furthermore, we profile two multithreaded Java workloads – SpecJBB and Cassandra (from the *Data Serving* application in Cloudsuite [12]). These workloads are representative of the types of big-data applications driving modern server design. Finally, we run sequential applications from Spec [16] (with *Ref* inputs) and two bioinformatics applications (Biobench [17]) with the largest inputs.

4.3 Real System Measurements

Figure 2 shows the number of cycles measured on page table walks as a percentage of total execution time and compares this with the fraction of runtime that would be expended if the MMU caches were perfect (100% hit rate

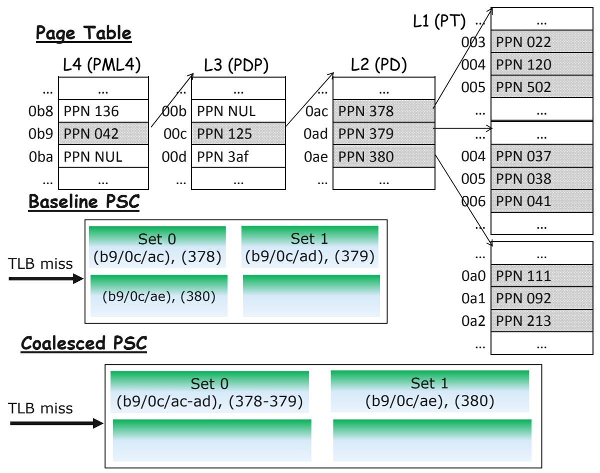


Figure 4: If consecutive L2 entries point to L1 pages placed at consecutive physical page frames, coalesced MMUs can store information of multiple translations in a single entry. Note that standard set-indexing schemes must be changed to accommodate this.

ignoring cold misses, with only L1 PTE memory references, all of which hit in the LLC).

Figure 2 shows that TLB miss overheads are substantial, particularly for `Canneal`, `Streamcluster`, the server and the bioinformatics workloads, and several Spec benchmarks (e.g., `Mcf` and `CactusADM`). Overheads typically arise due to pseudo-random access patterns (e.g., `Canneal`) or simply because the application's memory footprint far exceeds TLB capacity (e.g., `SpecJBB` and `Cassandra`, whose memory footprints comfortably exceed multiple gigabytes). Since TLB misses are overwhelmingly on the critical path of execution [25], runtime is severely degraded.

Figure 2 shows, however, that perfect PSCs boost performance. `SpecJBB`, `Cassandra`, `Tigr`, `Mummer`, and `Mcf` eliminate overheads of 10-17%. Figure 3 further plots the average number of cycles per walk. Each bar is divided into the number of cycles taken by a perfect MMU cache (30 cycles for L1 PTE lookup's LLC hit) and the overheads arising from additional MMU cache misses. In general, server and bioinformatics benchmarks, as well as `Mcf` suffer most.

5. COALESCED MMU CACHES

Having assessed the performance potential of ideal MMU caches, we focus on optimization techniques to achieve close-to-ideal performance. Our first optimization, coalesced MMU caches, uses modest OS support and simple complementary hardware enhancements to the MMU cache. In this section, we present software and hardware design details for coalesced MMU caches. We present motivational data when appropriate, deferring detailed evaluations to Section 9.

5.1 Concept

High-level idea. At a high-level, we must modify the OS to allocate page table pages to encourage coalescing. Novel coalesced MMU cache hardware must then detect this behavior and collapse multiple PTEs into single entries. Figure 4 illustrates these two components.

First, we have page tables conducive to coalescing, where consecutive L2 PTEs map consecutive physical page frames (where L1 page table pages are maintained). We propose OS code that produces this type of page table allocation.

Second, we propose hardware support that detects coalescing opportunity and exploits it. Figure 4 depicts the difference in operation between a standard L2 PSC and a coalesced L2 PSC. The coalesced MMU cache detects that L2 PTEs at `0xac` and `0xad` are coalescible (because they point to consecutive physical page frames) and merges them into a single L2 PSC entry. Overall, the coalesced PSC requires two entries to cache the three L2 PTEs, whereas the baseline PSC requires three entries. Importantly, coalesced PSCs change traditional cache set-indexing. In the baseline case, successive L2 PTEs stride across PSC sets; coalescing, however, requires successive L2 PTEs to map to the same set so that they can be merged (e.g., `0xac` and `0xad`).

Coalescing opportunity. Coalescing requires (1) consecutive L2 PTEs to be used; and (2) successive L2 PTEs to map to L1 page table pages in consecutive physical page frames. Though the OS guides (2), upper-bounds on coalescing opportunity are determined by (1), which is a program property.

Fortunately, Figure 5 shows great coalescing opportunity by plotting, on average, how many consecutive PTEs are allocated around each L2 PTE. We collect these numbers on real-system page tables with the methodology described in Section 8. Any value higher than 1 indicates some coalescing potential, with higher values signifying greater opportunity. We find that program has ample coalescing opportunity, with L2 contiguity usually above 10. Even for the sole exception, `Omnetpp`, the value is 3, which can be exploited for coalescing.

Coalescing scope in this work. Beyond L2 PSCs, L4 and L3 PSCs can also be coalesced. We focus only on L2 PSCs because their miss rates are most critical for performance.

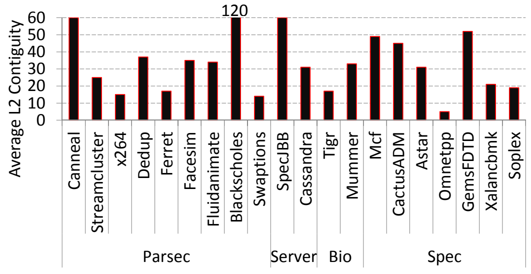


Figure 5: Average number of consecutive entries in the L2 PTE sets the upper-bound on coalescing opportunity.

5.2 Software Support for Coalescing

We would ideally modify the Linux kernel to influence page table allocation. Regrettably, such wholesale modification to the Linux kernel, particularly in the memory allocator, is complex and infeasibly time-consuming, particularly for early-stage research explorations. Instead, like past work [33], we detail where to place code in the kernel, and calculate its overheads using simulations and careful estimations.

Reservation-based page table page allocation. To encourage L1 page table page allocation on consecutive physical pages (if they are mapped by consecutive L2 PTEs), we envision a reservation-based mechanism, partly inspired by past work on superpages [24, 32]. At a high level, on a page fault, we see whether a new L2 PTE and an L1 page

```
int handle_mmu_fault(...){
  /* counters, mm semaphore acquired */
  pte_alloc();
  /* set pte offset maps */
}
int _pte_alloc(...){
  /* check reserved pages */
  for(i = 0; i < MAX_RESERVE; i++)
    if(reserve.check_pmd(i) == pmd)
      goto: found_page;
  pgtable new =
         pte_alloc_one(mm, address, 0);
  found_page:
  /* ensure all pte setup (page locking
     and clearing) done */
  /* reserve pages */
  for(i = -(MAX_RESERVE>>2);
        i<(MAX_RESERVE>>2); i++)
    reserve.add(pte_alloc_one(mm, address, i))
  /* check expiry */
  for(int i = 0; i < MAX_RESERVE; i++)
    pgtable reserved = reserve.read(i);
    if(reserved.reserve_num <
                  (CURR_NUM - EXPIRY_NUM))
       reserve.delete(i);
  /* put pointer to L1 level into L2 table */
  pmd_populate()
}
```

Figure 6: Reservation-based page table page allocation algorithm implemented in Linux kernel functions.

table page are needed. If so, we optimistically also *reserve* consecutive physical page frames around the L1 page table page, so that they can be allocated if the program uses a consecutive L2 PTE entry. This simultaneous L2 PTE and L1 page table page contiguity promotes coalescing.

Basic algorithm. Figure 6 sketches the code for reservation-based page table page allocation. Page faults call `handle_mmu_fault()`, which requires L1 page table page allocation. When this happens `_pte_alloc()` is called.

In `_pte_alloc()`, we add code that checks whether any of the already-reserved pages are for the L2 PTE currently requested. If so, we can use a reserved L1 page table page to encourage coalescing. If not, we default to the conventional case where `_pte_alloc_one()` allocates L1 page table pages. After this, we reserve consecutive pages around the allocated L1 page table page, hoping that the L2 PTEs around the currently requested L2 PTE will eventually be used. There are two important considerations when reserving pages. First, we have to choose how many pages to reserve. A higher number increases the chances of reserving a useful page; at the same time, it also increases reservations of pages that will ultimately be unhelpful. In our code, `MAX_RESERVE` determines the number of reservations.

Second, we must reclaim reserved pages if they remain unused after a sufficient amount of time. To do this, we maintain a running counter of the number of page table pages allocated. Every time we reserve a page, we tag it with the current counter value. Then, in `_pte_alloc()`, we check the counter values of all reserved pages against the current counter value. If the difference exceeds `EXPIRY_NUM`, these pages are unreserved, freeing memory. This parameter must also carefully chosen; a number too low reduces coalescing opportunity while one too high increases memory pressure.

Software overheads. Our code is invoked rarely (only

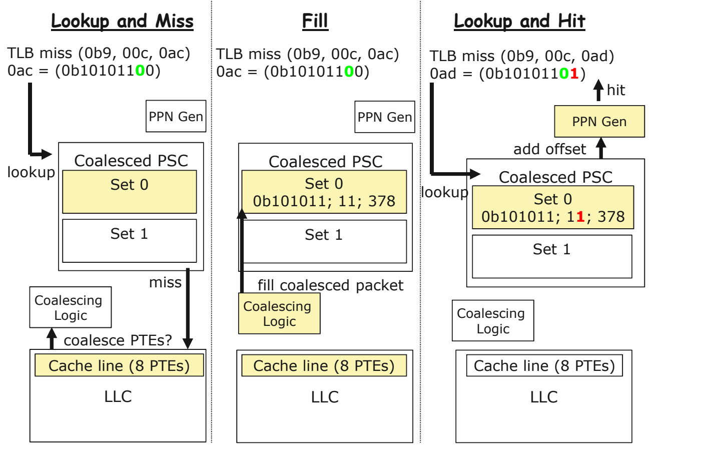


Figure 7: Lookup, hit, miss, and fill operations in a coalesced MMU cache, assuming the page table from Figure 4. Note that the diagram only shows partial tags (the full tag prefixes the tags shown with (0x0b9, 0x00c).

on page faults that also allocate page table pages). Furthermore, we track reserved pages as a linked list. Because we use small values of `MAX_RESERVE` and `EXPIRE_NUM` (detailed in subsequent sections), this list rarely exceeds 10-15 pages. This means that our code makes at most 30-45 memory references, which is orders of magnitude lower than the many hundreds made in the page fault handler (needed to access VMA trees, page tables, and various book-keeping structures) [11]. Finally, our data structures require no additional locks beyond semaphores used by the standard fault handler. This adds little overhead to some page faults.

5.3 Hardware Support for Coalescing

Address decomposition. When accessing a standard MMU cache, the virtual page number is decomposed into a tag and index. The least significant bits are used as the index in order to stride successive PTEs into different sets, reducing conflict misses. A coalesced MMU cache, however, requires consecutive PTEs to map to the same set (in order to permit coalescing). Index bits must therefore be left-shifted by n to permit coalescing of up to 2^n PTEs. For example, a standard four-set MMU cache uses VPN[1:0] to choose the correct set; an MMU cache that coalesces up to four PTEs uses VPN[3:2]. The bits left of the index become the new tag, while the lower bits (VPN[1:0]) index into a valid bit array (detailed next). A fundamental question in this work is to understand the best tradeoff between exploiting coalescing by left-shifting index bits, at the potential cost of increased conflict misses.

Coalesced entries. A standard MMU cache entry is a 5-tuple consisting of *(valid bit, tag, attribute, physical page, replacement policy bits)*. A coalesced MMU cache is made up of the 5-tuple, *(valid bit array, tag, attribute, base physical page, replacement policy bits)*. The valid bit array indicates the presence of a PTE in a coalesced packet. We only coalesce entries that share the same attribute bits (this can be relaxed with more hardware). Furthermore, we record the base physical page (i.e., the physical page number of the PTE whose valid bit is the first to be set in the array).

Lookup and miss operation. Figure 7 shows how the coalesced MMU cache (specifically, a PSC) operates on a lookup and miss operation. Since our example uses a two-set

PSC, VPN[1] is the index bit (Figure 7 shows the bottom 8 bits of the VPN and highlights the index bit). In our example, the empty PSC experiences a miss and accesses the next set of PSCs (L4 and L3 PSCs). Regardless of PSC hits, the page table walker will ultimately make a memory reference for the L2 PTE (and eventually the L1 PTE) which is typically found in the LLC or main memory.

Fill operation. Figure 7 shows that a PTE memory reference eventually loads a cache line of PTEs into the LLC. A typical 64-byte cache line stores eight 8-byte PTEs. These eight PTEs are scanned by *coalescing logic* that sits between the LLC and the MMU cache. Once the coalescing logic determines coalescible PTEs around the requested one, all the entries are merged into a single translation that is filled into the MMU cache. Note that we do not consider coalescing opportunities that may exist across LLC lines as they require additional memory references in the page table walk.

Lookup and hit operation. Figure 7 shows a lookup that hits in the coalesced PSC. The left-shifted index bit selects the desired set. Bits to the left of the index are compared with the tag bits (for space reasons, Figure 7 only shows the bottom 6 bits of the tag). Then, bits to the right of the index (highlighted in red) select a valid bit. If set, the desired PTE exists in this coalesced entry. *Physical page generation logic* then takes the valid bit array and the base physical page from the PSC entry to calculate the desired physical page. This is accomplished by adding the base physical page to an offset value (which is equal to the number of bits separating the first set bit in the valid array and the bit selected by this request). In our example, 1 is added to the base physical page of 378 to give physical page 379.

Replacement, invalidations, and attribute changes. We assume standard LRU replacement policies for this work. We also assume a single set of attribute bits for all coalesced entries. On shootdowns or invalidations, we flush out entire coalesced entries, losing information on PTEs that might otherwise be unaffected. Gracefully uncoalescing MMU cache entries and advanced replacement policies will perform better and will be the subject of future studies.

Hardware overheads. Like coalesced TLBs [25], coalesced MMU caches have modest hardware overheads. Lookup times are essentially unaffected because the change in tag matches, index selection, and valid bit lookup are simple and require no special hardware. Physical page generation requires only combinational logic (especially as the coalescing amount is bounded). Such readily-implementable logic is already available to prefetchers and branch predictors. Coalescing logic (which is just a combinational logic block) is accessed only on the fill path rather than the lookup. One may expect coalesced MMU caches to require additional ports to fill entries without conflicting with subsequent reads. We find, however, that more ports yield no performance benefits. We therefore assume a single port for our studies.

6. SHARED MMU CACHES

Unlike coalescing, which requires both hardware and software support, shared MMU caches are entirely transparent to software, requiring only simple hardware changes. We now detail these changes.

6.1 Concept

High-level idea. We replace per-core MMU caches with a single central MMU cache (of total equivalent capacity). We move the per-core page table walkers next to this structure. The shared MMU cache has a higher access latency from additional network traversal and because it is a larger structure. However, higher hit rates more than compensate for longer access times, boosting performance.

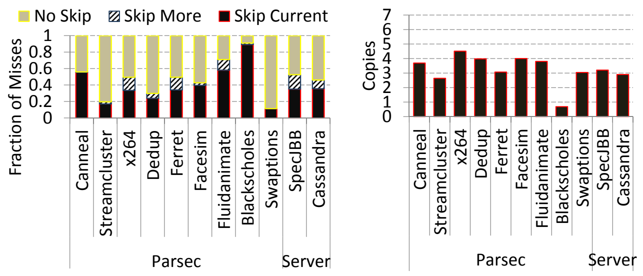


Figure 8: (Left) Avoidable MMU cache misses if each core could access all MMU caches; and (right) redundancy in per-core MMU cache entries across cores.

Benefits for parallel programs. Parallel programs employ threads that cooperate on the same data structures. Past work shows that as a result, TLB entries are often shared among multiple threads (and hence cores) [6, 7]. Since TLB entries map much smaller chunks of memory than MMU cache entries (4KB versus 2MB/1GB/512GB), the latter are likelier to be shared among cores. Shared MMU caches therefore provide two benefits.

First, the same MMU cache entries are often required by multiple cores. The left diagram in Figure 8 illustrates this (from real-system memory traces using the methodology in Section 8). For every MMU cache miss, we see how often other cores maintain the desired entries. This captures how often walks could have been accelerated had each core had access to all MMU caches. We show how often: (1) a core misses on a particular radix tree level while the same level's entry exists in another core's MMU cache (Skip Current); (2) another core actually stores a *lower* level in its MMU cache (Skip More); and (3) cases where shared MMUs would not accelerate the page table walk (No Skip). While (1) and (3) are easy to understand, an example of (2) is when a core misses on an L3 PTE while another core maintains not only that L3 PTE but also its associated L2 PTE.

Figure 8 shows that close to 40% misses could be averted because another core maintains desired PTEs. Often, other cores maintain entries from *even lower levels* of the walk. If these PTEs could be accessed, even more walk memory references could be skipped. Shared MMU caches exploit precisely this property.

Second, shared MMU caches store one PTE instance rather than multiple copies across cores. The right graph in Figure 8 shows, for private MMU cache hits, the number of PTE copies across the other cores. An 8-core CMP has up to seven other copies; the higher the number of copies, the greater the potential performance improvements from shared MMU caches. We see 3 to 5 copies of PTEs across cores for most benchmarks. A shared MMU cache eliminating these copies frees up space to store additional PTEs.

Benefits for multiprogrammed sequential workloads. Our initial goal is to use shared MMU caches to improve parallel program behavior *without* overly-degrading sequential applications. One may expect shared MMU caches to degrade sequential application performance (because of higher access latencies without inter-core sharing). In reality, we do not harm sequential applications and even achieve consistent performance benefits. This is because sharing better

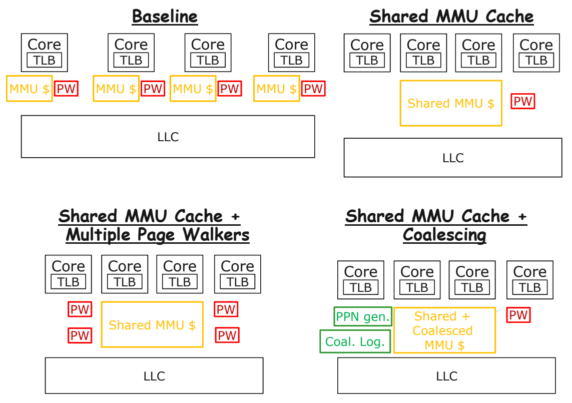


Figure 9: Comparison of baseline MMU caches with shared MMU caches (with and without multiple page table walkers), as well as a coalesced and shared MMU cache. Though not shown, each core also has a private L1 cache.

utilizes the total MMU cache capacity by only allocating entries to applications when needed (rather than statically provisioning a set amount per core, which wastes capacity).

6.2 Hardware Implementation Options

Figure 9 compares the hardware for conventional per-core MMU caches against various shared configurations. Shared MMU caches have the following hardware characteristics.

Shared MMU cache entries. Shared MMU cache entries have the same structure as private MMU caches. Entries include valid, tag, data, and protection bits. Each entry has a process ID (TLBs and MMU caches already maintain this [15]) so that context switches do not flush the MMU cache. Moreover, any shootdowns of individual entries are handled similarly to per-core MMU caches.

Placement and access latency. Figure 9 shows that we aggregate per-core MMU caches in a shared location available to all cores, moving the page table walkers in tandem. While this does increase hit rates, there are two sources of overhead. First, a larger centralized structure incurs a higher access latency for lookup. Second, the shared structure is placed further away from each core, adding an on-chip network traversal time.

Like past work [6], we assume that the last-level cache is distributed but the shared MMU cache is a monolithic structure. We use CACTI [22] to estimate the additional access times; however, we also vary access times to study their influence on performance.

Multiple page table walkers and ports. Figure 9 shows that per-core MMU caches have individual page table walkers. Therefore, each MMU cache can operate in parallel, serving simultaneous TLB misses on different cores. A key design question is whether the shared MMU cache should accommodate simultaneous accesses from multiple cores. We therefore consider the performance of maintaining multiple versus a single walker next to the shared MMU cache. We also consider multiported MMU caches. Per-core MMU caches have one read/write port but a shared version may need multiple ports to manage traffic from multiple cores.

Choice of sharing. MMU caches that have separate structures for L4, L3, and L2 PTEs (e.g., PSCs) can choose which structures to share. We assume that all structures are shared though follow-up studies will investigate this issue further.

Miss penalties. Entries from different levels of the page table walk suffer varying miss penalties. Higher-level PTEs enjoy higher sharing (as they map larger portions of the address space) but also suffer longer miss penalties. For example, L4 entries are reused more than L2 entries but also require two additional memory references on a miss. It is important that higher-level PTEs enjoy greater sharing to offset these penalties. Ultimately, performance depends on whether the shared MMU cache is like Intel's PSCs or AMD's PWCs. For example, PWCs are checked for every level of the radix page table. L4 and L3 entries must exist for the L2 entries to be useful.

Overheads. Coalesced MMU caches require modest hardware *and* software support. Instead, shared MMU caches need simple hardware tweaks and are transparent to software. Shared MMU caches essentially provide performance improvements "for free"; compared to per-core MMU caches, they have the same entry organization, same total capacity, no increase in ports, and the same number of page table walkers (we will show that even fewer walkers perform well).

7. COMBINED APPROACHES

Our coalescing and sharing strategies are orthogonal; we can therefore combine them to extract even greater benefits than either scheme alone. Combining the approaches requires no additional software or hardware support. In general, we will show that parallel programs benefit even more from coalescing on a shared MMU cache because the probability of sharing multiple coalesced PTEs is higher than that of sharing one PTE. We also find that on the rare occasion that a shared MMU cache degrades performance because of additional access latencies (usually for sequential applications with small memory footprints), coalescing boosts hit rates sufficiently to yield overall performance gains.

8. METHODOLOGY

We now detail our methodology, focusing on experimental infrastructure and evaluation benchmarks.

8.1 Experimental Infrastructure

Like past work, we note that software simulators are either too slow to run workloads for long enough duration to collect meaningful performance numbers for virtual memory studies or that they ignore full-system effects [2, 4, 8, 6, 30]. In response, we use a novel experimental methodology consisting of a microarchitectural software simulator which provides timing information, driven by a full-system trace extracted from a real system. This is similar to past approaches (e.g., like CMP$im [16]); however, unlike Pin traces, which cannot capture full-system effects (and page table accesses), we modify the Linux 2.6.38 kernel to provide full memory reference traces to drive our simulator.

Novel real-system memory traces. Similar to the methodology proposed in recent work [4], we have independently modified the Linux kernel's virtual memory allocator to track system memory references. Our tool records the virtual address, physical address, and full page table walk of memory references on a real system. We do this by essentially converting memory references into "fake" page faults that trap into the OS, where we log information.

Our strategy is to, at profile startup, poison the application's page table by setting each PTE's reserved higher order bits. We then flush the TLB. The empty TLB immediately suffers misses for requested memory addresses. Ordinarily, these TLB misses would be handled by the hardware page table walker, without invoking the OS. Since we have poisoned the page tables, however, the hardware walker traps to a custom kernel trap handler. After logging any desired information here, we load a clean TLB entry (though the page table remains poisoned so that future TLB misses also trap), allowing execution to continue.

Our approach traces fast enough to log tens of billions of memory references in 2-3 hours (much faster than Pin and software simulators). Furthermore, it retains full-system effects, providing full memory address information.

Performance evaluations. Most prior studies [2, 6, 30] focus exclusively on TLB miss rates because of slow simulation speeds. We too evaluate hit rates from our real-system traces, but go beyond by also determining performance. Since we transform memory references into interrupts, the timing of memory accesses in the trace is unrealistic. Therefore, we ally the traces with the software simulation strategy presented in prior work [25]. Our approach does not account for instruction replays that would actually occur on a TLB miss. Therefore our performance gains are conservative and would likely be higher.

We model an 8-core chip multiprocessor (CMP) with 32KB L1 caches and 8MB LLC (with a 30-cycle access time). Each core has 64-entry L1 and 512-entry L2 TLBs. We model two types of MMU caches (1) Intel's PSCs (32-entry, 4-way associative L2 PSCs, 4-entry L3 PSCs, 2-entry L4 PSCs); and (2) AMD's PWC (32-entry, 4-way associative). We assume 8GB main memory, with 150-cycle roundtrip time. Like *all* past architectural research on TLBs [2, 3, 5, 6, 25], we note that detailed memory models [29], while beneficial, have excessively high runtimes that are not appropriate or necessary for virtual memory hardware studies.

8.2 Evaluation Workloads

We use the same evaluation workloads as our real-system experiments, tracing up to 10 billion memory references, with the largest input sets. For all workloads (except `Cassandra`), we collect traces after 5 minutes to ignore program setup phases. For `Cassandra`, we collect traces after the workload completes the ramp-up period and reaches a steady state [20]. Furthermore, we use a 7GB Java heap and a 400MB new-generation garbage collector space.

We run 8 threads per parallel workload. Our studies on coalesced MMU caches for sequential applications model 8 simultaneous copies of the workload. However, shared MMU cache studies require multiprogrammed combinations of sequential applications. From the 7 sequential workloads (`Spec` and `Biobench`), we construct workloads with 2 copies of 4 applications (using all 8 cores), a total of 35 workloads. Note that each sequential application runs in its own address space; therefore when we run two copies of the sequential applications, they do not share the same translation entries and artificially increase sharing.

8.3 Our Approach

We assess the performance benefits of three designs: (1) per-core coalesced MMU caches; (2) a shared MMU cache; and (3) a shared and coalesced MMU cache. While we focus on overall application runtime improvement (rather than just the improvement of TLB miss handling time), we also provide hit rate data when appropriate. We also show

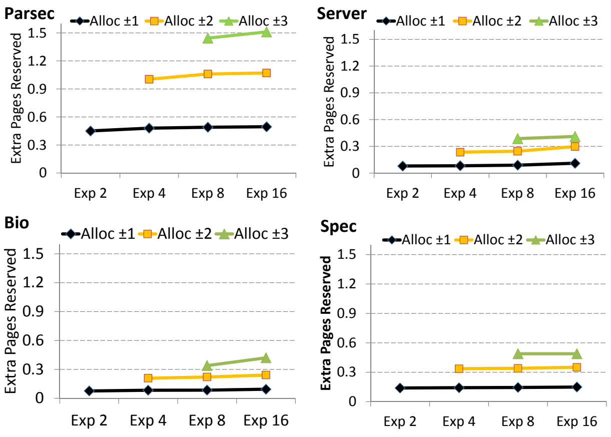


Figure 10: Additional pages reserved normalized to total page table pages.

how sensitive a shared and coalesced MMU cache is to access latencies of the larger shared structure, compare performance to shared and coalesced TLBs, and show how well our schemes perform versus naively increasing MMU cache size. Due to space limitations, most of our results focus on Intel's PSCs, though we have also studied AMD's PWCs. A small section compares the performance of both organizations.

9. EXPERIMENTAL RESULTS

9.1 Configuring Coalesced and Shared MMU Caches

Previous sections detailed the design parameters for coalesced and shared MMU caches. Before evaluating their performance benefits, we must configure our mechanisms to balance performance with overheads.

Coalescing. Our OS code must promote coalescible page tables with `MAX_RESERVE` and `EXPIRE_NUM` values that best balance coalescing opportunity and overheads. Reservation-based L1 page table page allocation must not significantly increase workload memory usage. Like past work on large pages [24], we track the number of additional pages that are reserved but unused as a fraction of the total pages.

Figure 10 plots the average unused reserved pages for several configurations. *Alloc* $\pm n$ (`MAX_RESERVE`) indicates which pages are reserved on an L1 page table page allocation (e.g., ± 1 means that the current page number plus one and minus one are both reserved). Furthermore, *Exp n* (`EXPIRE_NUM`) tracks how many additional page table allocations each reservation is active for, before it is relinquished. The higher the allocated amount and expiration amount, the higher the chance of producing reservations that are ultimately used; however, this also increases the risk of allocating useless reservations that would never be used.

The number of pages reserved, as a fraction of an application's total pages, is negligible for *every single* workload (all under 0.1%). Nevertheless, we conservatively study the overheads as a fraction of the number of total page table pages (rather than total pages) in Figure 10. All *Alloc* ± 1 schemes are particularly effective, restricting the number of reserved pages. The server, bioinformatics, and Spec benchmarks overheads are under 10% while Parsec overheads are higher (30%) because they typically use smaller page tables.

We have also quantified what fraction of reserved pages are eventually used. We have found that even modest reser-

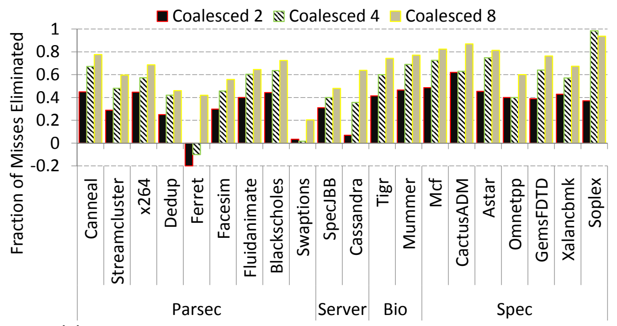


Figure 11: Miss elimination rates as index bits are left-shifted.

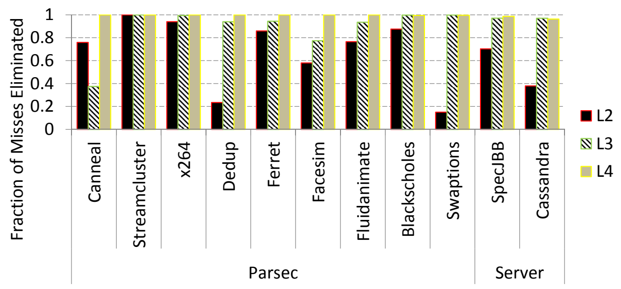


Figure 12: Miss elimination using shared MMU caches.

vation counts of ±1 result in high utilization (on average 73% of reserved pages are used). Utilization is particularly high for the server and bioinformatics workloads, a positive sign as these benchmarks suffer most from TLB misses. Furthermore, reserving more pages only negligibly increases how many of them are eventually used. We therefore only reserve ±1 with an expiration of 4 page allocations.

Beyond the OS code, a key hardware design tradeoff is to ensure that left-shifting index bits to enable coalescing is not offset by increased conflict misses. We therefore show the number of misses eliminated if a maximum of 2, 4, and 8 L2 PTEs are coalesced (left-shifting index bits by 1, 2, and 3 bits). We did not consider higher levels of coalescing because the coalescing logic scans a single cache line (which has a maximum of 8 PTEs) for coalescing opportunity.

Figure 11 shows that coalescing up to 8 PTEs eliminates the most misses (except for `Soplex` where the additional conflict misses lower miss eliminations slightly compared to coalescing 4 PTEs). Interestingly, some coalescing configurations increase miss rates due to higher conflict misses (e.g., `Ferret`) when coalescing only 2 or 4 entries. From these results, our coalesced MMU caches target coalescing 8 PTEs.

Sharing. We now showcase the hit rate increases of a shared MMU cache on an 8-core CMP versus per-core MMU caches, with equivalent total capacities. Figure 12 shows, for all parallel applications, how many L4, L3, and L2 level misses are eliminated when the per-core MMU caches are replaced by a single shared MMU cache. Shared MMU caches frequently eliminate most misses (with L2 miss eliminations ranging from 18% to 98%). Due to space constraints, we do not show the hit rate improvements of the multiprogrammed workloads; however, we will show overall performance results for these in subsequent sections.

9.2 Performance Improvements

Based on the previous sections, we quantify the performance benefits of the following configurations of MMU caches.

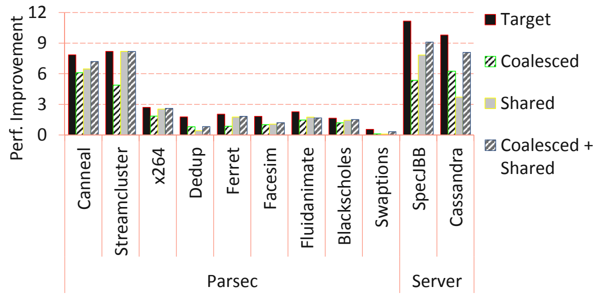


Figure 13: Percentage performance improvements from coalesced and shared MMU caches for parallel programs.

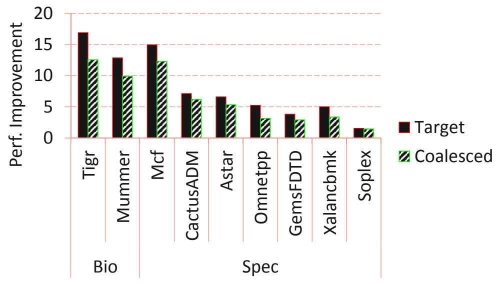


Figure 14: Percentage performance improvements from coalesced MMU caches for sequential programs.

Coalescing: We coalesce up to 8 PTEs per entry (left-shifting the index by 3 bits), using `MAX_RESERVE` of 2 (±1) and `EXPIRE_NUM` of 4 pages.

Sharing: We aggregate per-core MMU caches into shared structures of the same total capacity, with one page table walker and a single read/write port. For our 8-core CMPs, access times increase by 2× (from CACTI) in addition to network traversal time (which we model).

Combined: We combine coalescing and sharing with the same settings as the individual optimizations.

Parallel applications. Figure 13 quantifies performance improvements for parallel and multithreaded server applications. For each application, we compare the best achievable performance increase (*Target*) with the actual performance improvement from coalescing, sharing, and combining both approaches. Figure 13 shows that either coalescing or sharing boosts performance for every workload. Sharing achieves this despite the additional access latency. In most benchmarks, sharing slightly outperforms coalescing though there are exceptions (e.g., `Cassandra`). Overall, combining both approaches improves performance close to the target (7-9% for `Canneal`, `Streamcluster` and the server workloads).

Sequential applications. Figure 14 compares performance improvements from coalescing (sharing results are shown separately) from the best-achievable target. Again, there are performance gains in all cases (with `Tigr`, `Mummer`, and `Mcf` improved over 10%). In general, this approach falls about 2-3% from the ideal target.

For shared MMU caches, our initial goal was to minimize the frequency and magnitude of performance degradation for sequential applications. We achieve these goals by limiting performance degradation and comprehensively exceed them by combining coalescing with sharing.

Figure 15 orders, for the 35 multiprogrammed workloads,

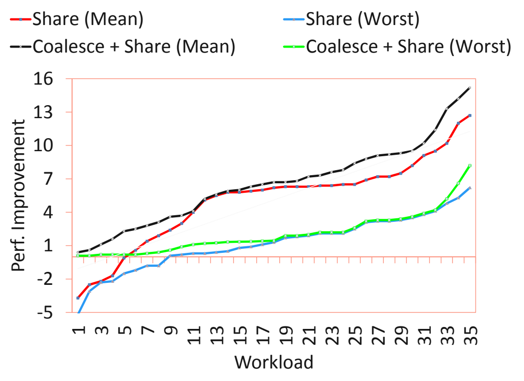


Figure 15: Percentage performance improvement (average and worse-case) of shared and combined approaches.

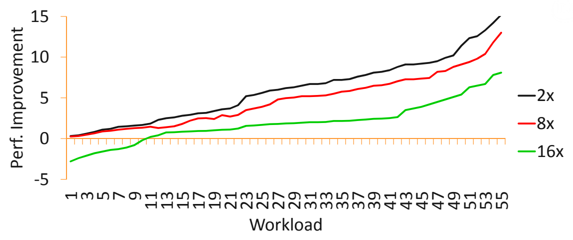


Figure 16: Impact of access latencies of shared and coalesced MMU caches on average performance.

lowest to highest performance improvements. We show the average performance improvement of the entire workload (a weighted average since some applications may be more memory-intensive than others) and the performance of the worst-affected workload (this allows us to comprehensively check all applications). First, Figure 15 shows that the overwhelming majority of workloads are consistently improved for both average, and worst-case performance (with an average of 7% and high of 15%). This occurs because the shared structure better allocates entries to different applications as needed, rather static allocation. Nevertheless, 5 workloads suffer average performance degradation (with 7 for worst-case performance). In these applications, the additional hit rates do not justify the increased access time.

Fortunately, Figure 15 also shows that combining coalescing with sharing completely offsets any case of performance degradation, even for *every single* worst-case workload. On average, performance improvements are now 9% with a best case of 18%. Coalescing and shared MMU caches are therefore consistent performance boosters.

9.3 Influence of Access Latencies

CACTI indicates that a shared structure suffers access latency increases of 2× for 8 cores. We also, however, quantify how higher access latencies mute performance. Figure 16 shows the average performance improvement of shared and coalesced MMU caches for all 55 workloads, assuming the default 2× access latency increase, and overly-pessimistic 8× and 16× latencies.

Figure 16 shows that even when access latencies are increased by 8×, performance improvements are very similar to 2× access latencies. Even in the worst-case, the performance difference is roughly 3%. This is because the increase in hit rates bought from coalescing and sharing far outstrip the higher access latencies. At 16× latency, while the overwhelming majority of applications still benefit from optimized MMU caches, 9 workloads do suffer a minor degradation (all under 3% reduction in performance). Note, however, that 16× access latencies are highly-pessimistic (and well beyond CACTI predictions). We believe that distributed MMU cache approaches (similar to distributed caches) will mitigate these overheads; we leave these studies for future work.

9.4 Comparison with TLB Optimizations

Past work has shown the benefits of coalesced TLBs [25] and shared last-level TLBs [6]. In this context, we compare two scenarios: (1) standard TLBs with combined coalesced and shared MMU caches; and (2) combining coalescing and sharing on both TLBs and MMU caches (like past work, we assume that a shared last-level TLB imposes an additional 8 cycle access time [6]). One might initially consider the benefits of a third scheme, shared and coalesced TLBs with standard MMU caches. However, a shared last-level TLB is placed close to the LLC. At that point, standard per-core MMU caches require messages to be relayed from the uncore back to cores, making this an impractical design option.

Figure 17 shows how well shared and coalesced MMU caches perform compared to sharing and coalescing both TLBs and MMU caches. We consider all 35 multiprogrammed workloads, but also multithreaded, server, and single instances of single-threaded workloads (giving a total of 55 applications). Again, the performance improvements are ordered from lowest to highest performance compared to a baseline with standard TLBs and MMU caches.

One might initially expect optimizations on both the TLB and MMU cache to outperform optimizations on just the MMU cache. Surprisingly, optimized MMU caches improve 9 of the 55 workloads versus optimized TLBs (and MMU caches). In fact, coalescing and sharing TLBs degrades 7 of these workloads, whereas only MMU cache optimizations consistently improve performance in every case. This is because a larger shared TLB can, in some cases, add excessive access latencies that are not outweighed by increased hit rates. A standard TLB, on the other hand, quickly checks for hits (and hence a miss does not incur additional access latencies), and forwards requests to a smaller shared MMU cache (which does not suffer as high an access time increase as a much larger shared TLB). Of course, future studies that consider advanced shared TLBs that are distributed may offset some of these overheads; nevertheless, we show that if we extend current TLB and MMU caches with very simple enhancements, optimized MMU caches can outperform optimized TLBs. Finally, Figure 17 shows that even when the optimized TLB and MMU caches outperform the optimized MMU caches, the performance difference is often small (usually 3-4%).

9.5 Multiple Ports and Page Table Walkers

A key question is whether shared MMU caches should have multiple ports and page table walkers to accommodate multiple simultaneous requests (which in the per-core case, would be simultaneously serviced by independent ports and walkers). We have found that increasing port counts (from the baseline single read/write port to 8 ports for 8 cores) negligibly affects performance. In general, this is because shared MMU caches experience low traffic and are small, shortening

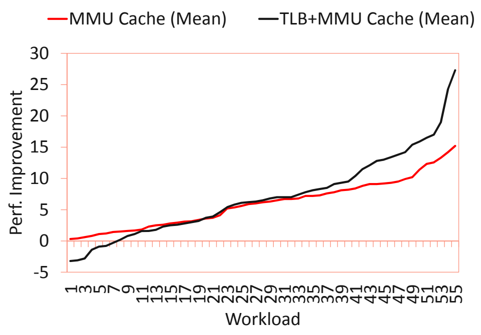


Figure 17: Percentage performance improvements using (1) baseline TLBs and shared and coalesced MMU caches; and (2) coalesced and shared TLBs *and* MMU caches.

access times (compared to TLBs) even when shared.

Multiple walkers, however, do boost performance. On average, including 4 page table walkers increases performance by an average of 1-2% across workloads (though this number is closer to 4% for the server applications). Beyond this, however, their performance improvements are negligible. These gains arise when multiple cores simultaneously experience MMU cache misses; in these cases, multiple walkers can overlap miss handling.

9.6 Comparisons with Larger MMU Caches

At first blush, one might consider enlarging per-core MMU caches rather than coalescing or sharing. However, past work on coalesced TLBs [25] show that hardware changes for coalescing have negligible impact on area, lookup times, and fill times. Moreover, shared MMU caches can actually *decrease* area since fewer walkers are required compared to per-core MMU caches. Either way, the total MMU cache capacity is preserved. Instead, increasing MMU caches requires additional area (and may increase access latencies).

Since coalesced and shared MMU caches often provide close-to-ideal MMU cache performance, per-core MMU caches must be enlarged substantially for similar hit rates. By evaluating a number of configurations, we have found that many benchmarks require substantially larger MMU caches to achieve comparable performance to coalesced and shared MMU caches of the baseline size; for example, `SpecJBB`, `Cassandra`, `Tigr`, and `Mcf` require 2-3× larger per-core MMU caches. Hence, we believe that coalescing and sharing are a far more effective approach to boost performance.

9.7 Comparison with Page Table Walk Caches

We have so far focused on PSCs but we have also studied PWCs. We find that on average, PSC performance across all schemes is about 3% higher than PWCs. This is because PWCs require a serial walk (the L4 PTE is looked up, then the L3 and then the L2). This has two implications: (1) the L4 and L3 PTEs are needed for an L2 PTE hit; and (2) multiple lookups in the MMU cache are necessary. Unfortunately, L4 and L3 PTEs can be evicted in shared PWCs by the higher L2 PTE counts. Furthermore, multiple lookups to a shared structure with higher lookup latency begins to offset some of the higher hit rate gains. Nevertheless, the benefits are still substantial. Furthermore, advanced designs that prioritize L4/L3 PTEs in the replacement policy could help address some of these problems.

10. BROADER INSIGHTS AND IMPACT

Instruction references. Like past work [2, 3, 6, 7, 8], we focus on data references because they traditionally determine performance. However, these techniques will also improving instruction reference behavior, particularly as instruction footprints increase in emerging server workloads [13, 20].

Coalescing versus prefetching. Prefetching speculates on what will be used in the future and when it will be used. In contrast, coalescing is aware exactly what items will be required in the future (i.e., any coalescible PTE *must* eventually be touched by the processor). However, when this item will be required is unknown; nevertheless, since this item is merged with the requested PTE, there are no capacity overheads. The only overheads arise from changes in set-indexing (which must not increase conflict misses) and additional hardware (which is modest compared to prediction tables typically required by prefetchers [31, 35]).

Orthogonality to past work on superpages. Our work has an interesting relationship with prior work on superpages [24, 27, 32]. Superpages are large pages (e.g., 2MB, 1GB in ×86 systems), allocated by the OS to reduce TLB pressure by orders of magnitude. While they can be effective, there are cases when they increase paging traffic [1]. Interestingly, we have shown that OS code that carefully allocates page table pages can be consistently low-overhead; based on our results, one can envision schemes where coalesced page table pages are used in conjunction or even as a replacement for superpages. We leave this line of research to future studies.

Applicability to other architectures. For example, ARM and Sparc, which use tree-based page tables (and for Sparc, software caches of the page table called *Translation Storage Buffers* [8] to accelerate walks) can also benefit from coalescing and sharing PTEs.

Scaling for future architectures. An advantage of per-core MMU caches is that as more cores are added on chip, greater capacity is achieved automatically with more MMU caches also stamped out. Since each MMU cache's size remains largely constant, its access times do not lengthen. Shared monolithic MMU caches however, require a careful balance between increased capacity and access times and must therefore be re-evaluated as core counts change.

We believe that the key to mitigating the redesign of shared MMU caches with increased core counts is to study distributed shared MMU cache organizations. Distributed designs offer a mechanism to increase cache capacity without overly degrading access latency. While past work on distributed caches [19] are a good starting point, the different granularity of MMU cache entries (hundreds of pages instead of cache lines), variation in entry granularity (L4, L3, and L2 PTEs map different amounts of main memory), and its tight access time requirements need dedicated studies in their own right.

11. CONCLUSION

This paper is the first to quantify real-system performance of MMU caches and presents the foundation for many optimization studies for these structures. We show that coalescing and sharing approaches can each improve performance by up to 10-12%. Since the approaches are complementary, we can combine them to achieve close to ideal performance.

Overall, we show the need to optimize hardware for virtual memory beyond TLBs, particularly with the advent of emerging software. We show that novel directions involving intelligent page table allocation, page table construction, and low-overhead hardware optimizations eliminate many TLB miss overheads (particularly for upcoming server and bioinformatics workloads). These insights will become particularly compelling for big-data systems.

12. REFERENCES

[1] Andrea Arcangeli, "Transparent Hugepage Support," *KVM Forum*, 2010.

[2] T. Barr, A. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *ISCA*, 2010.

[3] ——, "SpecTLB: A Mechanism for Speculative Address Translation," *ISCA*, 2011.

[4] A. Basu, J. Gandhi, J. Chang, M. Swift, and M. Hill, "Efficient Virtual Memory for Big Memory Servers," *ISCA*, 2013.

[5] R. Bhargava *et al.*, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *ASPLOS*, 2008.

[6] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2010.

[7] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," *PACT*, 2009.

[8] ——, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," *ASPLOS*, 2010.

[9] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *PACT*, 2008.

[10] D. Clark and J. Emer, "Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement," *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1985.

[11] A. Clements, F. Kaashoek, and N. Zeldovich, "Scalable Address Spaces Using RCU Balanced Trees," *ASPLOS*, 2012.

[12] M. Ferdman *et al.*, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," *ASPLOS*, 2012.

[13] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," *MICRO*, 2011.

[14] Intel Corporation, "TLBs, Paging-Structure Caches and their Invalidation," *Intel Technical Report*, 2008.

[15] ——, "Intel Developer's Manual," 2012.

[16] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation - A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites," *VSSAD Technical Report*, 2007.

[17] A. Jaleel, M. Mattina, and B. Jacob, "Last Level Cache (LLC) Performance of Data Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads," *International Symposium on High Performance Computer Architecture*, 2006.

[18] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," *ISCA*, 2002.

[19] C. Kim, D. Burger, and S. Keckler, "NUCA: A Non-Uniform Cache Architecture for Wire-Delay Dominated On-Chip Caches," *IEEE Micro Top Picks*, 2003.

[20] P. Lofti-Kamran *et al.*, "Scale-Out Processors," *ISCA*, 2012.

[21] G. Loh and M. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," *MICRO*, 2011.

[22] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," *Micro*, 2007.

[23] D. Nagle *et al.*, "Design Tradeoffs for Software-Managed TLBs," *ISCA*, 1993.

[24] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.

[25] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large Reach TLBs," *MICRO*, 2012.

[26] M. Qureshi and G. Loh, "Fundamental Latency Tradeoffs in Architecting DRAM Caches," *MICRO*, 2012.

[27] T. Romer *et al.*, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," *ISCA*, 1995.

[28] M. Rosenblum *et al.*, "The Impact of Architectural Trends on Operating System Performance," *SOSP*, 1995.

[29] P. Rosenfield, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[30] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," *ISCA*, 2000.

[31] S. Somogyi *et al.*, "Spatio-Temporal Memory Streaming," *ISCA*, 2009.

[32] M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *ASPLOS*, 1994.

[33] M. Talluri, M. Hill, and Y. Khalidi, "A New Page Table for 64-bit Address Spaces," *SOSP*, 1995.

[34] Vlastimil Babka and Petr Tuma, "Investigating Cache Parameters of x86 Processors," *SPEC Benchmark Workshop*, 2009.

[35] C.-J. Wu *et al.*, "PACman: Prefetch-Aware Cache Management for High Performance Caching," *MICRO*, 2011.

[36] L. Wu, R. Barker, M. Kim, and K. Ross, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," *ISCA*, 2013.