Efficient Address Translation for Architectures with Multiple Page Sizes

Guilherme Cox and Abhishek Bhattacharjee
Department of Computer Science, Rutgers University
{guilherme.cox, abhib}@cs.rutgers.edu

Abstract

Processors and operating systems (OSes) support multiple memory page sizes. Superpages increase Translation Lookaside Buffer (TLB) hits, while small pages provide fine-grained memory protection. Ideally, TLBs should perform well for *any* distribution of page sizes. In reality, set-associative TLBs – used frequently for their energy-efficiency compared to fully-associative TLBs – cannot (easily) support multiple page sizes concurrently. Instead, commercial systems typically implement separate set-associative TLBs for different page sizes. This means that when superpages are allocated aggressively, TLB misses may, counter-intuitively, increase even if entries for small pages remain unused (and vice-versa).

We invent **MIX TLBs**, energy-frugal set-associative structures that concurrently support all page sizes by exploiting superpage allocation patterns. MIX TLBs boost the performance (often by 10-30%) of big-memory applications on native CPUs, virtualized CPUs, and GPUs. MIX TLBs are simple and require no OS or program changes.

CCS Concepts • **Computer systems organization** → **Pipeline computing**; **Multicore architectures**

Keywords Virtual memory; TLB; superpages; coalescing

1. Introduction

The operating system's (OS') choice of page sizes for an application's memory needs critically impacts system performance. Modern processors and OSes maintain multiple page sizes. Superpages (or large pages) increase Translation Lookaside Buffer (TLB) hit rates [1, 2, 3]. Small pages provide fine-grained page protection and permissions [1, 3, 4]. This paper's objective is to design a TLB that leverages any distribution of page sizes, with the following properties:

① **Good performance**: TLB hardware should not be underutilized and conflict misses should be avoided.

② **Energy efficiency**: TLBs can consume a significant amount – as much as 13-15% [5, 6, 7, 8, 9] – of processor energy. Our design should be energy-efficient.

③ **Simple implementation**: TLBs reside in the timing-critical L1 datapath of pipelines, and must be simple to meet timing constraints. This means that TLB lookup, miss handling, and fill must not be complex.

Meeting all three objectives, while handling multiple page sizes, is challenging. Meeting ② means that we use set-associative rather than fully-associative TLBs. However, set-associative TLBs cannot (easily) support multiple page sizes. This is because, on lookup, they need the lower-order bits of the virtual page number to select a TLB set. But identifying the virtual page number requires the page size, so that the page offset bits can be masked off. This presents a chicken-and-egg problem, where the page size is needed for TLB lookup, but lookup is needed to determine page size. In general, industry and academia have responded in two ways, which compromise ① and/or ③.

Split TLBs: Most processor vendors use split (or partitioned) TLBs, one for each page size [11, 12, 10]. This sidesteps the need for page size on lookup. A virtual address can look up all TLBs in parallel. Separate index bits are used for each TLB, based on the page size it supports; e.g., the set indices for split 16-set TLBs for 4KB, 2MB, and 1GB pages (assuming an x86 architecture) are bits 15-12, 24-21, and 33-30 respectively. Two scenarios are possible. In the first, there is either hit in one of the split TLBs, implicitly indicating the translation's page size. In the second, all TLBs miss [10].

Unfortunately, while split TLBs achieve ③, and arguably ②, they often underutilize TLBs and compromise ①. The problem is that if the OS allocates mostly small pages, superpage TLBs remain wasted. On the other hand, when OSes allocate mostly superpages, performance is (counter-intuitively) worsened because superpage TLBs thrash while

ASPLOS '17, April 08-12, 2017, Xi'an, China

DOI: http://dx.doi.org/10.1145/3037697.3037704

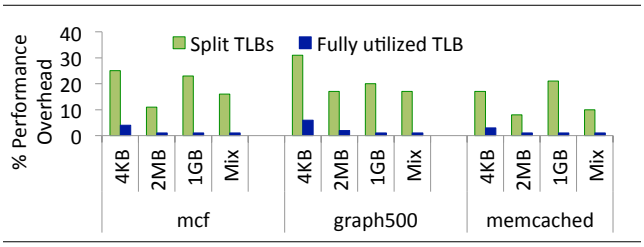

Figure 1. Percentage of runtime devoted to address translation, running natively on Intel Haswell with Linux (green). We assume cases where the OS allocates only one page size (4KB, 2MB, 1GB) and when page sizes are mixed. We compare performance against an ideal case where all TLB resources are well-utilized (blue).

small page TLBs lie unused [13, 14]. Figure 1 quantifies the extent of the problem, showing the percentage of runtime that mcf, graph500, and memcached devote to address translation. Results are collected using performance counters on Intel Haswell systems with 84GB of memory, running Linux with the methodology of Sec. 6. We assume that the OS allocates only a fixed page size (i.e., 4KB, 2MB, 1GB) or mixed pages. One would expect that using large pages consistently improves performance. In reality, performance remains poor even with, for example, 1GB pages (green bars). Further, we compare these numbers to a hypothetical ideal set-associative TLB which can support all page sizes (blue); the gap with the green bars indicates the performance potential lost due to poor utilization of split TLBs.

Multi-indexing approaches: In response to this problem, past work has augmented set-associative TLBs to concurrently support multiple page sizes [10, 15]. Unfortunately, while this does improve ①, it does so at the cost of ② and ③. The central problems, described in Sec. 5.1, are variable access latencies, increased access energy, and complex implementation. Even in the rare cases when they are implemented commercially, they don't support all page sizes (e.g., Intel's Haswell, Broadwell, and Skylake L2 TLBs cache 4KB and 2MB pages together but not 1GB pages, which require separate TLBs [11, 12]).

Our contributions: This work proposes (**MIX**) **TLBs**, fast ①, energy-efficient ②, and readily-implementable ③ structures that concurrently support all page sizes. MIX TLBs use a single set-indexing scheme – the one for small pages (e.g., 4KB pages on x86) – for translations of all page sizes. While this simplifies the design, it also presents a problem. We use bits within the superpage page offset to select a TLB set. This means that a superpage is mapped to multiple (potentially all) TLB sets, an operation we refer to as **mirroring** (see Sec. 3). We overcome this problem, however, by observing that OSes frequently (though they don't have to) allocate superpages (not just their constituent small pages) in adjacent or *contiguous* virtual and physical addresses. We detect these adjacent superpages, and **coalesce** them into the same TLB entry (see Sec. 3). If we coalesce as many, or close to as many, superpages as the number of mirror copies – which we usually can in real-world systems – we counteract the redundancy of mirrors, achieving energy-efficient performance.

This paper showcases MIX TLBs, their ease of implementation, and performance improvements of 10-30%. Using real-system characterization and careful simulation, we compare MIX TLBs to traditional set-associative designs, and previously proposed TLBs for concurrent page sizes [10, 15]. We also characterize superpage allocation patterns. Our results focus on Linux, but we've also studied FreeBSD and Solaris. One might initially expect that highly loaded sytems with long uptimes would be hard-pressed to defragment memory sufficiently to allocate superpages adjacently. Indeed, we observe that if system memory is sufficiently fragmented, OSes rarely generate superpages at all. However, we also observe that if OSes can generate even a few superpages, they have usually defragmented memory sufficiently to generate other adjacent and contiguous superpages too. MIX TLBs outperform their counterparts in both cases. When superpages are scarce, MIX TLBs use all TLB resources for small pages. When superpages are present, MIX TLBs seamlessly leverage any distribution of page sizes.

2. Scope of This Work

Systems are embracing workloads with increasing memory needs and poorer access locality (e.g., massive key-value stores, graph processing, data analytics, deep learning frameworks, etc.). These workloads stress hardware TLB performance; as a result, address translation overheads often consume 15-30% of runtime today [16, 17, 18, 19, 46].

MIX TLBs also aid virtualized systems, where address translation is even more pernicious. Virtualized systems require two dimensions of address translation - guest virtual pages are converted to guest physical pages, which are then translated to system physical pages [4, 13, 20]. Two-dimensional page table walks are expensive, requiring 24 sequential memory accesses in x86 systems, instead of the customary 4 accesses for non-virtualized systems. Virtualization vendors like VMware identify TLB misses as a major culprit in the performance difference between non-virtualized and virtualized systems [4, 13, 14].

Finally, vendors have begun embracing shared virtual memory abstractions for heterogeneous systems made up of CPUs and GPUs [21, 22, 23, 24, 25, 26, 27, 28], accessing a single virtual address space. This allows "a pointer is a pointer everywhere" simplications of the programming model [26, 27]. However, now GPUs must also perform address translation, just like CPUs. GPU TLBs are critical to performance as they must service the demands of hundreds to thousands of concurrent threads [21, 22, 27]. Unfortunately, we find that CPU-GPU systems also suffer from TLB utilization issues when using multiple page sizes.

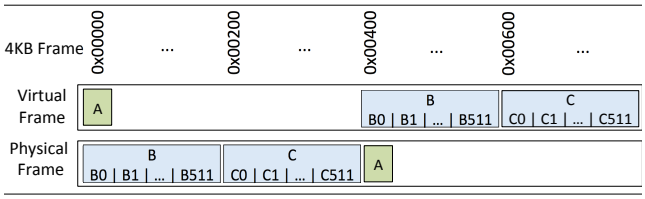

Figure 2. Example address space in an x86-64 architecture. We show 4KB frame numbers in hexadecimal. For example, translation B is for a 2MB page, made up of 4KB frame numbers B0-B511. 2MB translations B-C are contiguous.

3. High-Level Approach

We compare MIX TLBs to traditional split TLBs, using the address space of Figure 2. We show virtual and physical address spaces, with translations for small pages (A), and superpages (B-C). Without loss of generality, we assume an x86-64 architecture with 4KB and 2MB pages (1GB are handled similarly). Note that while we assume 64-bit systems, our examples show 32-bit addresses to save space. These addresses are shown in 4KB frame numbers (full addresses can be constructed by appending 0x000). Therefore, superpage B is located at virtual address 0x00400000 and physical address 0x00000000. Superpages B and C have 512 constituent 4KB frames, indicated by B0-511 and C0-511.

Figure 3 illustrates the lookup and fill operation of MIX TLBs and contrasts it to split TLBs. In step ①, B is looked up. However, since B is absent (both split and MIX TLBs maintain only A), the hardware page table walker is invoked ②. The page table walker reads the page table in units of caches lines; since a typical cache line is 64 bytes, and translations are 8 bytes, 8 translations (including B and C) are read in the cache line. Split TLBs then fill B into the superpage TLB ③. Unfortunately, there remains no room for C despite 3 unused small page TLB entries.

MIX TLBs, on the other hand, cache all page sizes. After a miss ① and a page table walk ②, we must fill B in the correct set. This presents a challenge; since MIX TLBs use the index bits for small pages (in our 2-set TLB example, bit 12) on all translations, the index bits are picked from the superpage's page offset. Thus, superpages do not uniquely map to either set. Instead, we mirror B in both TLB sets.

Mirroring presents a problem. Whereas split TLBs maintain one copy of a superpage translation, MIX TLBs maintain several mirror copies, reducing effective TLB capacity. However, MIX TLBs counteract this problem with the following observation – OSes frequently (though they don't have to) allocate superpages adjacently in virtual and physical addresses. For example, Figure 2 shows that B and C are contiguous, not just in terms of their constituent 4KB frames (e.g., B0-511 and C0-511) but also in terms of the full superpages themselves. MIX TLBs exploit this contiguity; when page table walkers read a cache line of translations ②, adjacent translations in the cache line are scanned to detect contiguous superpages. We propose, similar to past work [16, 17], simple combinational coalescing logic for this ③. In our example, B and C are contiguous and are hence coalesced and mirrored. Coalescing counteracts mirroring. If there are as many contiguous superpages as there are mirror copies (or close to as many), MIX TLBs coalesce them to achieve a net capacity corresponding to the capacity of superpages, despite mirroring.

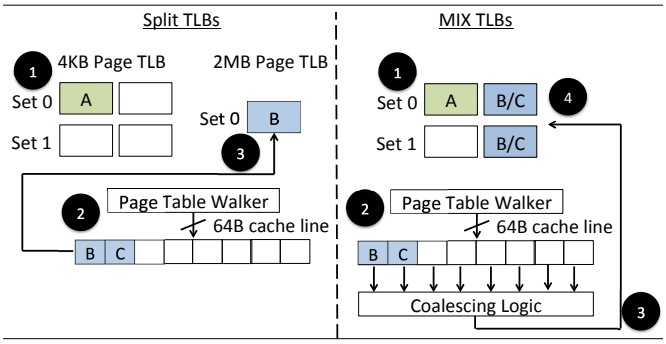

Figure 3. Superpage B lookup and fill for split versus MIX TLBs.

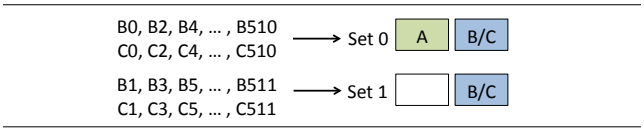

Figure 4. Though superpages B and C are maintained by multiple sets but on lookup, we only probe the set corresponding to the 4KB region within the superpage that the request is to.

Crucially, Figure 4 shows that MIX TLB lookup remains simple. While coalesced mirrors of superpages reside in multiple sets, lookups only probe one TLB set. In other words, virtual address bit 12 in our example determines whether we are accessing the even- or odd-numbered 4KB regions within a superpage; therefore accesses to B0, B2, etc., and C0, C2, etc., are routed to set 0.

Naturally, this overview presents several important questions. We briefly address them below:

Why do MIX TLBs use the index bits corresponding to the small pages? Specifically, one may instead consider using the index bits corresponding to the superpage and apply that on small pages too. In our example, this would be like using virtual address bit 21 as the index (assuming we base the index on 2MB superpages). The advantage of this approach is that each superpage maps uniquely to a set, *eliminating* the need for mirrors (e.g., B maps to set 0, and C maps to set 1).

Unfortunately, this causes a different problem. Now, spatially adjacent small pages map to the same set. For example, if we use the index bits corresponding to a 2MB superpage (i.e., in our 2-set TLB example, bit 21), groups of 512 adjacent 4KB virtual pages map to the same set. Since real-world programs exhibit spatial locality, this elevates TLB conflicts

(unless associativity exceeds 512, which is far higher the 4-8 way associativity used today [11, 12]). One could envision coalescing these small pages if the OS does allocate them contiguously in virtual and physical addresses; however past work shows that while small pages can be contiguous, they usually are not contiguous beyond more than tens of pages [16, 17]. We have evaluated using superpage index bits and have found that they increase TLB misses by 4-8× on average, compared to using small page index bits.

Why do MIX TLBs perform well? MIX TLBs are well utilized for any distribution of page sizes. When the system is highly fragmented and superpages are scarce, all TLB resources can be used for small pages. When the OS can generate superpages, it usually sufficiently defragments physical memory to allocate superpages adjacently too. MIX TLBs utilize all hardware resources to coalesce these superpages.

How many mirrors can a superpage produce and how much contiguity is needed? Assume that the superpage has N 4KB regions, and that our MIX TLB has M sets. N is 512 and 262144 for 2MB and 1GB superpages. Practical commercial L1 and L2 TLBs tend to have 16-128 sets [10, 11, 12]. Therefore, today's systems have $N > M$, meaning that a superpage has a mirror per set (or N mirrors). However, if future systems see $N < M$, there would be M mirrors.

Ultimately, good MIX TLB utilization relies on superpage contiguity. If the number of contiguous superpages is equal (or sufficiently near) the mirror count, performance is good. On modern 16-128 set TLBs, we desire (close to) 16-128 contiguous superpages. Sec. 7.1 shows that real systems do frequently see this much superpage contiguity. Sec. 4 shows how we can coalesce these many contiguous superpages, despite only scanning for contiguity within a single cache line, which maintain 8 translations, on a TLB miss.

4. Hardware Details

We now detail MIX TLB hardware, implementing them differently for the L1 and L2 levels. L1 MIX TLBs must be simple and fast; we sacrifice some coalescing opportunity to meet these requirements. L2 MIX TLBs can tolerate higher access latencies (e.g., Intel and AMD L2 TLBs usually have 5-7 cycle access times [10]). Therefore, L2 MIX TLBs support more coalescing with (slightly) more complex hardware. MIX TLBs require no OS or application changes.

4.1 MIX TLB Entries

MIX TLB entries are similar to traditional set-associative entries. We detail the modest differences between the two. Although actual x86-64 architectures can use up to 52-bit physical addresses, use assume 48-bit physical addresses in our example for simplicity. Extending this approach to 52-bits parallels our example.

Small pages: Figure 5 contrasts traditional TLB and MIX TLB entries for 4KB pages. We use translation A from

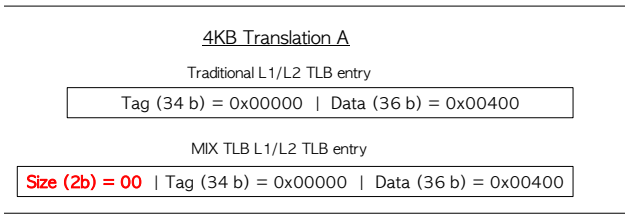

Figure 5. Traditional TLB and MIX TLB entries for the translation corresponding to 4KB page A. We show the TLB entries at the L1 and L2 level, assuming both have 4 sets. MIX TLBs require just an additional 2 bits to record the page size.

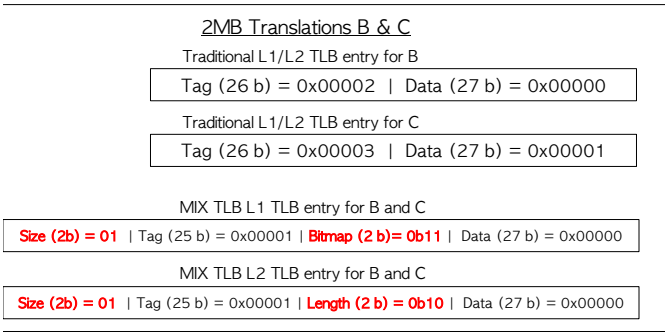

Figure 6. Traditional TLB and MIX TLB entries for the translation corresponding to 2MB pages B-C. L1 MIX TLB and L2 MIX TLB entries use a bitmap and a length field to record contiguous superpages, respectively. We assume 2-set TLBs.

Figure 2, and assume 4-set L1 and L2 TLBs. Therefore, the two least significant bits of the virtual page need not be stored in the tag. MIX TLBs only require a 2-bit page size field to distinguish among the 3 page sizes. Though they are not shown, the entries also maintain page permission bits.

Superpages: Figure 6 compares traditional to MIX TLB entries for superpages, assuming 2-set TLBs. Aside from the page size, MIX TLBs must maintain information about coalesced superpages. L1 entries use a bitmap for this. 2-set MIX TLBs maintain a 2-bit bitmap to record coalescing information of up to two superpages. Furthermore, since this entry caches superpage information, it uses 9 fewer tag bits for this versus small page entries. In fact, we can even drop a 10th bit because 2-bit bitmaps implicitly store information about 2×2MB (4MB) memory regions. These bits can be repurposed for the bitmap. Figure 6 records 0b11 to indicate information about contiguous superpages B and C.

L2 MIX TLBs record longer contiguity, with marginally greater complexity. Instead of a bitmap, we use a contiguity length field. Therefore, a 2-bit length field (though it could use more bits) records contiguity of up to 4 superpages.

MIX TLBs are only marginally bigger than a standard set-associative entry since the bitmap and length fields are

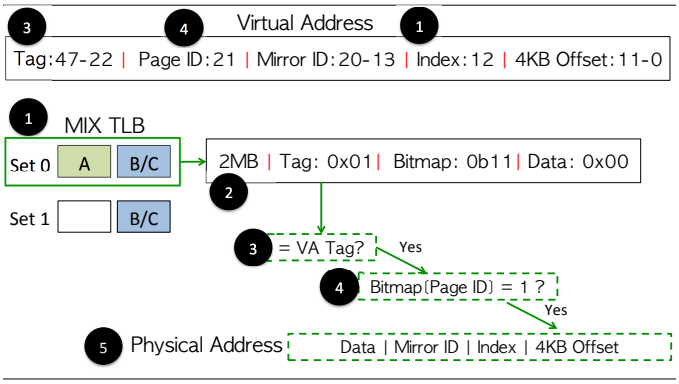

Figure 7. L1 MIX TLB lookup and hit (assuming a 2-set TLB). The physical address is found using bit shifting and concatenation.

repurposed with unused tag bits. Only a 2-bit page size field is added, increasing per-entry size by less than 1%.

Alignment restrictions: To simplify MIX TLB hardware, we only coalesce superpages that are suitably aligned. Specifically, to coalesce up to N superpages, only contiguous superpages that begin at virtual address boundaries of N may be coalesced. Our MIX TLB example in Figure 6, which coalesces up to 2 superpages, therefore only coalesces superpages that begin at multiples of 2×2MB or 4MB. This does reduce coalescing opportunity slightly, but as we show in Sec. 7.2, performance continues to be good.

Bitmap versus length: For the same number of bits, length fields record more information, allowing L2 MIX TLBs to coalesce longer runs of contiguous superpages. The slight downside is the slightly more complex TLB lookup this prompts (which we detail later in this section). L1 bitmaps do have one more advantage – they can record information about "holes" in contiguously allocated superpages.

4.2 MIX TLB Operation

In this section, we describe MIX TLB operation, including hits, misses, and fills.

L1 lookup: Figure 7 shows how L1 MIX TLBs are looked up. Since 4KB page lookups remain unchanged, we focus on superpages. Index bits are selected from the virtual address as per small page size – therefore, assuming a 2-set TLB and 4KB small pages, we use bit 12 as the index. Consequently, there is a question as to what happens with the remainder of the 2MB page offset, bits 20-13, and bits 11-0. We call bits 20-13 the mirror ID, as they identify individual 4KB regions within a superpage (i.e., B0, B1, B2, etc., in Figure 4). Bits 11-0 are the offset within these 4KB regions. Finally, the remaining upper order bits of the virtual address are split into a tag and a page ID. The page ID identifies the specific superpage within a contiguous bundle – since our example assumes a 2-set TLB that can coalesce up to 2 entries, 1 page ID bit suffices to identify the desired superpage.

The index bits identify the MIX TLB set ①. In our example, we cache A and B-C in the set, so both entries are checked in parallel. The page size determines whether the entry is a coalesced superpage bundle ②. Then the tag is compared to the virtual address tag ③. If there is a match ④, the L1 bitmap must be checked to determine whether this superpage exists in the coalesced entry. This is accomplished by indexing the bitmap using the page ID; in our example with B-C, this is set. Therefore, the physical address can be constructed by concatenating the relevant fields of the lookup virtual address with the data field in the MIX TLB entry ⑤.

Note that this process essentially leaves lookup latency unchanged from the conventional TLB because it relies purely on bit shifts and concatenations.

L2 lookup: For L2 MIX TLB lookups, instead of checking a bitmap, the length field is checked against comparators to determine whether the desired virtual page falls within in the range of coalesced translations. Range matches (and their implementation) are well studied [16, 29]. On a range (and hence TLB) hit, the hardware computes the offset of the lookup virtual page against the tag information stored. This offset is added to the TLB entry's data field, calculating the desired physical address, similar to recent work [16, 29]. This does increase the L2 lookup latency; however we've modeled the hardware (see Sec. 6) and found that there is only a slight (i.e., 3% increase) in lookup time.

Miss and fill: Section 3 sketched how MIX TLB are filled. By scanning for contiguous superpages and coalescing only on TLB misses, the coalescing logic is placed off the critical path of lookup. Therefore, it has low overhead, can be designed with simple combinational logic, and adds negligible latency or energy over a baseline set-associative TLB.

Prefetching and capacity strategies: Superpages provide two benefits. First, they record information about a far bigger portion of memory than small pages; hence, they reduce TLB conflict misses. However, they also provide prefetching benefits. For example, a TLB fill of a 2MB superpage obviates the need for 512 separate fills for 4KB pages.

This observation informs the way MIX TLBs coalesce. At first blush, one might consider only filling superpage information into the set that was probed by the lookup virtual address. For example, suppose that in Figure 4, there is a lookup for superpage B, but to the B0 region specifically. On a TLB miss, one option might be to only fill superpage information into set 0. To be sure, this does capture some of the prefetching benefits of superpages (i.e., information for B2, B4, etc., are also filled), but it also loses prefetching potential for some 4KB regions (i.e., B1, B3, etc.). For this reason, we instead fill as many sets as necessary with superpage mirrors to capture information about the full superpage.

In fact, our notion of prefetching goes beyond the prefetching benefits of superpages, because coalescing actually

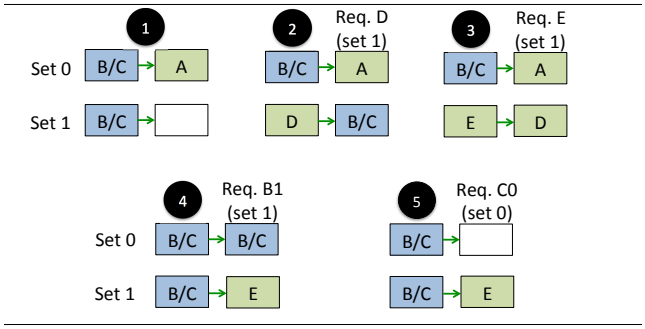

Figure 8. Replacement decisions are made independently on mirror copies, which can cause duplication issues.

prefetches contiguous superpages around the requested superpage. To do this without complicating the page table walker, we prefetch (by coalescing) only contiguous superpages that sit in the same cache line as the page table translation (up to 8 superpages, see Figure 3). Note, however, that commercial Sandybridge/Haswell TLBs maintain 16-128 sets; this means that MIX TLBs should try to coalesce 16-128 superpages to offset mirroring. One needs to scan additional cache lines containing the page table to do this. Instead, we choose a simpler approach. We initially coalesce up to 8 superpage entries. When future memory references touch superpages adjacent to these coalesced entries, sitting in other cache lines, we detect this behavior and coalesce them into the existing MIX TLB entry. This achieves good performance in practice.

4.3 Interactions with Replacement Policies

One issue with MIX TLBs is that information for the same superpages are now distributed among multiple TLB sets. Figure 8 illustrates the challenges this brings up. We show a 2-set, 4-entry MIX TLB, explicitly indicating each set's LRU chain. Suppose that initially ①, it stores information about A and B-C. Now, requests for D and E, small pages mapping to set 1, arrive and are filled into the TLB in steps ② and ③. At this point, set 1's mirror copy of B-C is evicted, while set 0's copy remains. This presents a problem in ④, when we have a request for superpage B but in 4KB region B1, which maps to set 1. We see a TLB miss, and walk the page table; however, once we locate B-C, an important question is whether to mirror B-C into the other sets. On the one hand, the other sets may already have copies of B-C and blindly mirroring leads to duplicate copies. On the other hand, a TLB maintains 64-128 sets; scanning all the sets to check for duplicates is an energy-expensive and impractical approach. Therefore, we adopt the first approach; ④ shows that this leads to a duplicate B-C copy in set 0, evicting A. However, we can mitigate this problem when set 0 is probed in the future ⑤. Since all the entries in the set are checked for a tag match, we identify duplicates and eliminate copies.

4.4 OS Operations

Invalidations: The OS may change page table mappings through program execution and corresponding TLB entries must be invalidated. For small pages, this is achieved in the same way as conventional TLBs. For superpages, this is accomplished in L1 MIX TLBs by resetting the bitmap bit of the superpage in question. This permits superpages adjacent to the invalidated superpage to remain cached.

On L2 MIX TLBs however, this is slightly more complicated, because they maintain a length field. The simplest approach is to invalidate the entry corresponding to the entire coalesced bundle. A more sophisticated approach might split the entry into two separate entries around the invalidated translation. Since we find, in practice, relatively few invalidations, we take the (slightly) lower performance but simpler approach of invalidating the entire coalesced entry.

Permission bits: An important question is whether to coalesce adjacent superpages that use different access permission bits. While this could be accomplished with more storage in the MIX TLB to record differing permissions, we take the simpler (but high-performance) approach of only coalescing superpages when they have the same permission bits.

Dirty and access bits: Translations in page tables maintain access and dirty bits to aid the OS' page replacement policy. In some (though not all) architectures, like x86 and ARM, these bits are set by hardware, and read by the OS.

The x86 architecture mandates that only translations with access bits set to 1 in the corresponding page table entry may be filled into the TLB [30]. MIX TLBs therefore coalesce only translations with access bits set to 1 on TLB fill. Naturally, this does not preclude translations from being added to existing coalesced entries in the TLB once they are demanded by the processor and have their access bits set.

Page table entries also maintain a dirty bit, recording whether the page has been written to. Conventional TLBs maintain a dirty bit per entry; on a store instruction to the translation in the entry, this bit is checked. If the bit is 0, the hardware page table walker injects a micro-op instruction to write the software page table entry's dirty bit [30, 31, 32]. If the bit is 1, such updates are not needed.

On the one hand, MIX TLBs could maintain a dirty bit per superpage in a coalesced bundle. However, MIX TLBs support 16-128 superpages per coalesced bundle; requiring 16-128 dirty bits per TLB entry requires infeasible storage. On the other hand, we could require that only superpages with the same dirty bit value be coalesced; unfortunately, we've found that this drastically reduces coalescing opportunity.

Instead, our approach sets the MIX TLB entry dirty bit only if all the superpages in a coalesced bundle are dirty. If a single one of them is not dirty, the TLB entry's dirty bit is cleared. In practice, this means that every time there is a TLB miss and page table lookup, we check to see if the requested PTE's dirty bit is set. If it is clear, and the MIX

TLB entry where this translation is to be coalesced has its dirty bit cleared (if it is not already clear). If it is set, the dirty bit of the MIX TLB entry is left unchanged. That is, if it was already dirty, it is left dirty again. Naturally, this approach adds cache traffic versus a scenario where the TLB has a dirty bit per translation. In practice though, we've found that performance remains good, and area overheads are modest.

4.5 Hardware and Energy Complexity

MIX TLBs are readily-implementable. As detailed, the size of each MIX TLB entry is roughly 1% bigger than a standard set-associative TLB. We've modeled these overheads using CACTI [33], and find that lookup latency and energy remains unchanged. MIX TLB misses do invoke coalescing logic; however, like prior work [16, 17, 19], we find that this requires only simple combinational logic. While it does impose (slight) delay overheads on TLB fill, we have modeled these overheads in RTL and find that they do not affect overall performance. Furthermore, while it is true that coalescing logic uses some area, MIX TLBs eliminate the need for separate superpage TLBs. Therefore, we ultimately *save* area.

Finally, we consider the energy costs of mirroring. Mirroring (coalesced) superpages into multiple sets does consume more energy than conventional TLBs, which fill one set. Modeling this using CACTI and RTL, however, we find that the much higher hit rates offered from MIX TLBs greatly reduce memory and cache references (for page table walks) and reduce runtime. Ultimately, the resulting energy benefits outweigh the energy overheads of mirroring.

5. Comparison to Past Work

MIX TLBs present a counterpoint to split set-associative TLBs. However, prior work has looked at alternate ways to provide mixed page support too. We now detail this past work, showing why MIX TLBs are superior.

5.1 Multi-Indexing Methods

Recent approaches to tackling the inadequacies of split set-associative TLBs can be summarized into three categories:

Hash-rehashing: We initially perform a TLB lookup (hash) assuming a particular page size (usually the baseline page size). On a miss, the TLB is again probed (rehash), using another page size. This continues until all page sizes are exhausted [10]. There are several drawbacks to this approach. TLB hits have variable latency, and can be difficult to manage in the timing-critical L1 datapath of modern CPUs [34], while TLB misses take longer. One could parallelize the lookups but this adds lookup energy, and complicates port utilization. Consequently, hash-rehashing approaches are used in only a few architectures, and that too, to support only a few page sizes (e.g., Intel Skylake and Haswell architectures support 4KB and 2MB pages with this approach but not 1GB pages).

Skewing: Skewed TLBs are inspired by skewed associative caches [15, 35, 36]. A virtual address is presented to multiple parallel hashing functions. The functions are chosen so that if a group of translations conflict on one way, they conflict with a different group on other ways. Translations of different page sizes reside in different sets [15]. For example, if our TLB supports 3 page sizes, each cacheable in 2 separate ways, we need a 6-way skew-associative TLB.

Skew associative TLBs can be effective but also have problems. Lookups expend high energy as they require parallel reads equal to the sum of the associativities of all supported page sizes. Saving energy by reducing the associativity of page sizes decreases performance. Further, even the simplest skewing functions are usually not appropriate for latency-sensitive L1 TLBs [37, 38]. Finally, good TLB hit rates require effective replacement policies; unfortunately, skewing breaks traditional notions of set-associativity and requires complicated replacement decisions. In practice, skewed TLBs use area- and energy-expensive timestamps for replacement [37, 38]. Because of these problems, we know of no commercial skew-associative TLBs.

Prediction-based enhancements: Recent work [10] enhances hash-rehashing and skewing by using a hardware predictor to accurately guess the page size of a requested translation before TLB lookup. The hash-rehash or skew TLB is first looked up with this predicted page size; only on misses are the other page sizes used. When prediction is accurate, this approach lowers the average TLB hit latency and lookup energy, by first looking up with the "correct" page size. Problems remain with this approach. Predictors become complex as the number of page sizes increase. Further, predictors increase access latency variability since we now also have different latencies for hits with correct prediction, eventual hits after a wrong prediction, etc.

Overall, multi-indexing is complex, latency-variable, and can be energy-intensive. It is generally unsuitable for L1 TLBs. Even when implemented, it can scale poorly as the number of page sizes increase. In addition multi-indexing potentially complicates operations like TLB shootdowns, selective invalidations of global versus local translations, managing locked translations, etc., because of their multi-step lookup [10]. Instead, since MIX TLBs remain largely unchanged in implementation compared to standard set-associative TLBs, they do not suffer from these issues.

5.2 Prior Work on Page Allocation Contiguity

The concept of exploiting OS page allocation for better TLB performance has received recent attention [2, 16, 17, 18, 19, 34]. COLT [16] shows that small pages are often allocated contiguously in virtual and physical memory. These contiguous small pages are coalesced into 4KB TLBs for better performance. This builds on earlier work [2] which exploited certain patterns of page allocation contiguity and alignment.

We make two observations about MIX TLBs and their relationship with COLT. First, they solve a problem that COLT cannot – realizing a single set-associative TLB to cache multiple page sizes. Second, COLT observes that there are cases when small pages are allocated more frequently than superpages. In these cases, coalescing small pages helps TLB performance. Our work also observes that there are situations where superpages are hard to allocate. We provide orthogonal benefits to COLT in these cases, by utilizing TLB entries that would otherwise have been devoted only to superpages. In addition, unlike COLT, we also help performance in the numerous cases when superpages abound. Indeed, for these reasons, COLT can actually be combined with MIX TLBs (see Sec. 7.2).

6. Methodology

We use a mix of real-system measurements, memory tracing, and detailed simulation. We now describe these approaches.

6.1 Real-System CPU Measurements

To assess real-system split TLB performance and OS page allocation patterns, we use a dual-socket Intel processor with 4-way set-associative split TLBs for 4KB pages (64 entries) and 2MB pages (32 entries). 1GB L1 TLBs are fully-associative and 4 entries. We use a 512-entry L2 TLB for 4KB and 2MB pages, but not 1GB pages. Instead, there is a separate 32-entry L2 TLB for 1GB pages. Further, this system is equipped with a 24MB LLC and 80GB of memory.

We focus on Linux (kernel 4.4.0) but we've also run FreeBSD and Solaris and found similar results. Furthermore, our virtualization studies focus on KVM; however, we have also run VMware ESX and see similar results.

6.2 CPU Simulations and Analytical Models

To evaluate MIX TLBs, we need to go beyond existing hardware platforms. Unfortunately, current cycle-accurate simulators cannot run fast enough to collect meaningful data for all the long-running, big-data workloads (with multiple OSes, and hypervisors) we require for our CPU studies. Therefore, like most recent work on TLBs [13, 16, 18, 19, 29, 34, 39], we use a combination of tracing, performance counter measurements, functional cache hierarchy and TLB simulations, and anaytical modeling to estimate overall impact on program execution time. We use Pin [40] to collect memory traces. We extend Pintools with Linux' pagemap to include physical addresses with the virtual addresses that Pin normally generates. We select a Pinpoint region of 10 billion instructions and correlate traces with performance counter measurements to verfiy that the sampled region is representative of the workload.

These traces are passed to a functional simulator – used to assess TLB and cache hit rates – that models multi-level TLBs, hardware page table walkers, and a cache hierarchy. Our baseline is a split TLB hierarchy from Intel Haswell systems; further, we model area-equivalent hash-rehash and skewed TLBs, with prediction-based enhancements. Finally, we model an area-equivalent MIX TLB hierarchy.

We use the hit rates from our functional simulation to, like past work [13, 16, 18, 19, 29, 34, 39], feed into an analytical model that uses the performance counter data to weight the performance impact of TLB hits, misses, and cache accesses.

6.3 GPU Simulation

Our GPU studies use cycle-level CPU-GPU simulation based on gem5-gpu, running Linux, and modeling an x86 architecture. Like recent work [21, 22, 23], we model 128-entry, 4-way set-associative TLBs for 4KB pages per shader core. We also model split TLBs for 2MB pages (32-entry, 4-way) and 1GB pages (4-entry, fully-associative).

6.4 Workloads

Our CPU studies use two sets of applications. The first consists of all workloads from Spec and Parsec [41]. We scale the inputs of these workloads so that their total memory footprint is roughly 80GB. The second set uses big-memory workloads (e.g., gups, graph processing, memcached, workloads from Cloudsuite [42]), also tuned to 80GB.

Our GPU studies, like recent studies [21, 22, 23] use workloads from Rodinia [43]. Ideally, our GPU studies should use the same big-memory sizes as our CPU studies, but this makes simulations infeasibly slow. We therefore scale our inputs to 24GB memory footprints.

7. Evaluation

We now present an evaluation of MIX TLB in two steps – first, a study of OS page allocation patterns and second, quantification of the benefits of MIX TLBs.

7.1 OS Page Allocation Characterization

Several factors affect the OS' page size distribution. For example, suppose we run Linux with transparent hugepage support (THS) [44]. As the program makes memory allocation requests (e.g., malloc(), or mmap()), the OS earmarks virtual pages, using its virtual memory area data structure [45]. If these requests are to large amounts of memory, several contiguous virtual pages are reserved. Virtual pages are lazily allocated physical pages, as the program page faults through the virtual pages. The OS consults its free pool of physical pages for this. THS tries to defragment memory sufficiently to maintain swathes of contiguous free physical pages. If there is enough free physical memory for superpages, THS can assign 2MB physical pages to 2MB regions of virtual addresses, generating superpages. Further, if the program page faults through the virtual pages in ascending order, they are handed contiguous physical pages.

Instead of THS, Linux can also use libhugetlbfs. This is a special library that administrators or programmers have to explicitly link to [29]. Users can specify a superpage preference (e.g., 2MB or 1GB). At link time, programs reserve

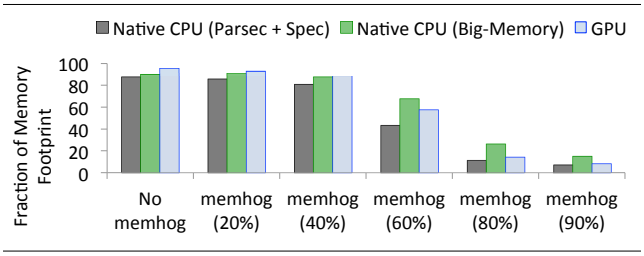

Figure 9. Fraction of memory footprint occupied by superpages, as fragmentation varies. Results shown for native CPUs and GPUs.

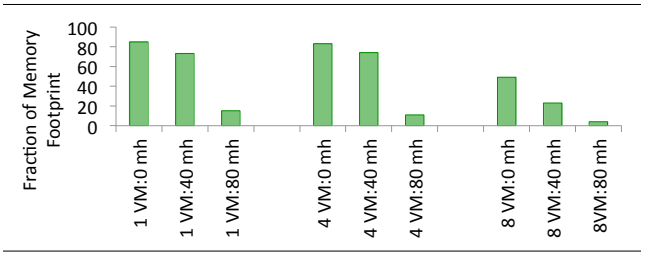

Figure 10. Fraction of memory footprint occupied by superpages, as a function of the memory fragmentation and VM consolidation. Results are for virtualized CPU workloads. N VM: M mh stands for N consolidated VMs, each with memhog running at M%.

a pool of memory. Superpages are allocated from this pool; when this pool is used up, small pages are allocated from other memory locations. Like THS, libhugetlbfs relies on lazy physical memory allocation and OS degfragmentation of physical memory.

Page size distributions: Figure 9 quantifies the page distributions we see on our real-system CPU and GPU experiments, both running Linux. The graphs show the fraction of total memory footprint backed by superpages (both 2MB and 1GB). As we have detailed, physical memory fragmentation impacts the frequency of superpage allocation. Therefore, we vary the level of memory fragmentation by running the microbenchmark memhog [16, 17], which allocates memory randomly across a fraction of system memory, in the background. For example, memhog (40%) on the x-axis indicates that memhog is fragmenting 40% of the 80GB system memory in the background. We show average numbers for classes of workloads (e.g., Parsec + Spec, big-memory workloads) as per-workload numbers follow these trends. Figure 9 shows three regimes of page distributions.

Superpages dominate: With moderate amounts of memory fragmentation, superpages cover most of the application's memory needs. For example, even with memhog fragmenting 40% of physical memory, more than 80% of a CPU's or GPU's workload is covered with superpages, on average.

Neither small pages nor superpages dominate: When memory fragmentation further increases, the memory footprint is more equally divided among small pages and superpages. For example, memhog with 60% finds that 40-60% of CPU, and 55% of GPU footprints are backed by superpages.

Mostly small pages: When fragmentation becomes severe, the bulk of the memory footprint is backed with small pages.

Figure 10 shows that similar trends hold for virtualized workloads. To create memory fragmentation and system load, we first consolidate as many VMs on the same machine as possible. Each consolidated VM is provided 10GB of memory; therefore, 8 of them use up all 80GB of available physical memory. In addition, we run memhog within each VM, fragmenting a percentage of each VM's footprint. We expect that higher VM consolidation and more aggressive memhog use will reduce the frequency of superpages.

Figure 10 shows that OSes running in VMs can counter non-trivial amounts of memory fragmentation, producing lots of superpages. For example, even 4VMs with memhog of 40% each, see more than 70% of memory is allocated in superpages. Naturally, as system load increases, small pages dominate. For example, like recent work [4, 47, 48], we find that as more VMs are in the system and memory pressure increases, optimizations like page sharing [48] and NUMA migrations [49] preclude heavy use of superpages.

Overall, this data suggests many system factors influence page size distributions, and therefore, systems experience a variety of such distributions. It is therefore vital to implement efficient TLB support for mixed page sizes.

Contiguous superpages characterization: MIX TLBs rely on contiguity among superpages when they are present. Figure 11 quantifies the amount of superpage contiguity for the workloads and configurations from Figures 9-10 where at least one superpage is present. Figure 11 quantifies average contiguity per workload (numbered in ascending order of superpage contiguity on the x-axis). Average contiguity is measured as follows. We scan the entire page table and identify runs of contiguous superpages. We divide this contiguity by the number of translations. For example, suppose we have a page table with 4 entries, where the first 2 translations are singletons, but the last two are contiguous. We calculate that the average contiguity is $(1 + 1 + 2 \times 2)/4$. We separate results for 2MB superpages and 1GB superpages, varying the amount of memory fragmentation using memhog.

Figure 11 shows that superpages themselves – and not just their constituent 4KB page regions – are usually allocated contiguously in virtual and physical addresses. For example, consider memhog fragmenting 20% of physical memory. Figure 11 shows that most benchmarks have average 2MB page contiguity greater than 80. This means that 80+ 2MB superpages can potentially be coalesced in our TLBs. Since CPUs (e.g., see Intel's Sandybridge, Haswell, and Skylake TLBs) use 16-set L1 TLBs, this is sufficient to entirely offset mirrors for L1 MIX TLBs. L2 TLBs usu-

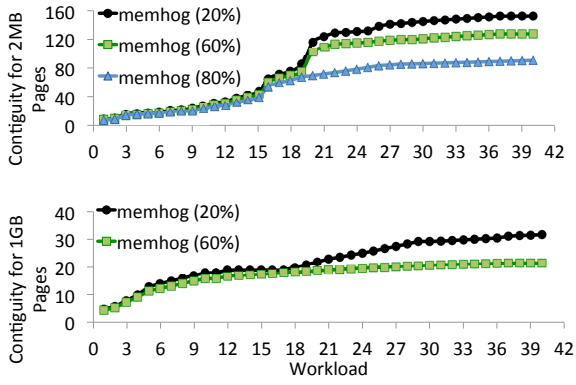

Figure 11. Average superpage contiguity for native and virtualized CPU, and GPU workloads. We show trends as memory fragmentation is increased with memhog, separately for 2MB and 1GB superpages.

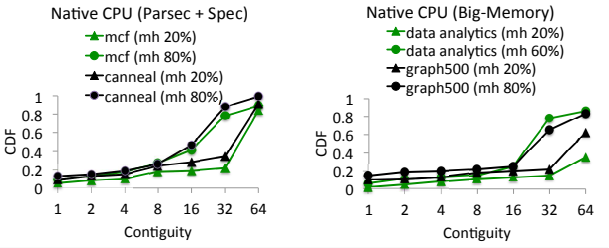

Figure 12. Superpage contiguity CDF as memhog varies, for native CPU workloads.

ally have 64-128 sets; while memhog at 20% and 60% see enough 2MB page contiguity to offset this consistently, contiguity does drop with more fragmentation. Nevertheless, even in these cases, it is enough (80+) that it can sufficiently (though not entirely) enable coalescing to counter mirroring.

Figure 11 also shows contiguity for 1GB pages. Since 1GB pages require much larger defragmented physical memory regions than 2MB pages, they are harder to form. As a result, the number of contiguous 1GB pages is usually lower than 2MB pages. Most workloads see 20-30 contiguous 1GB pages, even with relatively high fragmentation when memhog is 60%. Fortunately, since this covers 20-30GB of memory in an 80GB memory system, this amount of contiguity is good enough for effective coalescing.

Figures 12 and 13 focus on superpage contiguity in terms of the cumulative distribution functions (CDFs) for native CPU, virtualized CPU, and GPU workloads. Once again, memory fragmentation is controlled using memhog, and virtualized results also rely on VM consolidation to generate load. Applications with higher contiguity see the largest increases in CDF values further along the x-axis. Figures 12-13 show that all these workloads see considerable contiguity, even when system fragmentation is high.

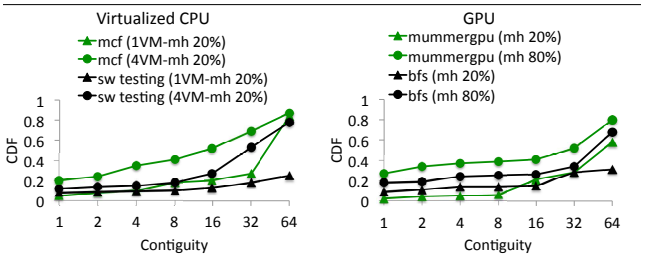

Figure 13. Superpage contiguity CDF as memhog varies, for virtualized CPU and GPU workloads

7.2 Results

We begin by comparing the performance of MIX TLBs against commercially available Intel Haswell TLB configurations. Note that while we refer to this as a split configuration, it uses split L1 TLBs, but partly-split L2 TLBs (i.e., 4KB and 2MB translations are hashed-rehashed in 1 TLB, while 1GB translations are cached in a separate TLB). We then also compare MIX TLBs to simulated multi-indexing schemes (i.e., hash-rehash and skew TLBs at both the L1 and L2 levels for all page sizes). Finally, we demonstrate how MIX TLBs perform in tandem with past work on COLT.

Comparisons to split TLBs: Figure 14 shows performance improvements using area-equivalent MIX TLBs versus a Haswell style TLB. To conserve space, we pick representative benchmarks from Spec + PARSEC, the big-memory workloads, and the GPU applications. We also show average results for the remaining workloads in each category.

We separate results for several cases. For native CPU workloads, we first use libhugetlbfs to try to use 4KB, 2MB, or 1GB pages exclusively. We then run native CPU workloads on a system with transparent hugepage support or THS enabled, where Linux attempts to allocate as many 2MB pages as possible, backing off to 4KB pages if this is not feasible. We also show results for virtualized CPU workloads, with 1 VM, and then a consolidated system with 4 VMs. The VMs are configured to support whatever mix of 4KB, 2MB, and 1GB pages the guest OS and hypervisor think are appropriate. Finally, we show GPU workloads on non-virtualized systems.

Figure 14 shows that MIX TLBs outperform commercial TLBs comprehensively, frequently in excess of 10%. For setups where small pages are prevalent (e.g., 4KB bars), we see more than 8% performance improvements on native and virtualized CPUs, as well as GPUs. This is because split TLBs cannot use the 2MB/1GB L1 TLBs, and the 1GB L2 TLBs for 4KB pages, while MIX TLBs do not have this utilization problem.

Figure 14 also shows that MIX TLBs perform well when superpages become more prevalent. For example, in the 2MB or THS cases, where 2MB pages become more common, MIX TLBs achieve better performance than split be-

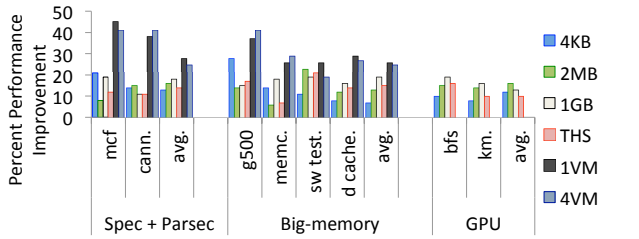

Figure 14. Percent performance improvement from MIX TLBs compared to area-equivalent split TLBs.

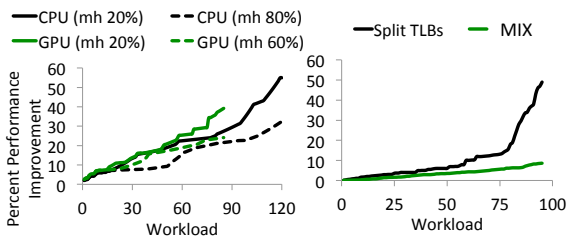

Figure 15. (Left) percentage performance improvement of MIX TLBs versus split TLBs, with memhog varying; (Right) Performance overheads of split TLBs and MIX TLBs compared to ideal hypothetical TLBs which never miss.

cause they can utilize all hardware for 2MB pages, not just 2MB page TLBs. And these gains are even higher, in excess of 12%, for 1GB pages, which can only use small 1GB page TLBs in the split.

Unsurprisingly, MIX TLBs are particularly useful when TLB misses become more expensive. Therefore, for virtualized workloads, where TLB misses necessitate expensive two-dimensional page table walks [4, 20, 50], 40%+ performance improvements are seen. Similarly, GPUs, which experience heavy TLB miss traffic [21, 22, 23], enjoy significant performance benefits for any distribution of page sizes.

Figure 15 sheds further light on performance benefits, in the presence of memory fragmentation. The graph on the left shows MIX TLB performance improvements over split TLBs, as memhog fragments 20% and 80% of CPU memory, and 20% and 60% GPU memory. We arrange the workloads (numbered on the x-axis) in ascending order of performance benefits. As expected, increasing memory fragmentation does reduce performance as it reduces the incidence of superpages; nevertheless, MIX TLBs consistently outperform split TLBs by 20%+.

The graph on the right of Figure 15 compares how well MIX TLBs do versus a hypothetical ideal TLB which never misses and cannot hence be realized. We plot curves for the performance overheads experienced by split TLBs and MIX TLBs versus this ideal TLB. The lower the y-axis values, the better. While almost a third of the split Haswell TLBs

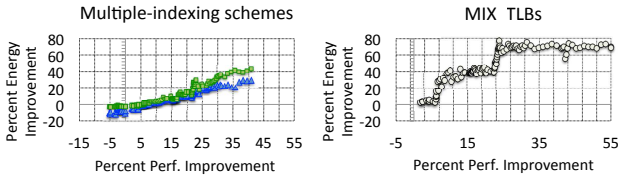

Figure 16. (Left) performance-energy tradeoffs for skew-associative TLBs with prediction (blue) and hash-rehash with prediction (green); and (Right) MIX TLB performance-energy tradeoffs.

experience a 10%+ performance deviation from the ideal scenario, MIX TLBs always achieve under 10%.

Comparisons to multi-indexing methods: We now compare performance and energy benefits of MIX TLBs versus area-equivalent skew-associative and hash-rehash (enhanced with the best prediction strategies [10]) approaches. Figure 16 shows these approaches for native and virtualized CPUs, as well as GPUs. We plot each workload along two dimensions. On the x-axis, we plot percent performance improvement versus the split TLB design. On the y-axis, we plot the percent address translation energy saved, also versus split TLBs. Therefore, we desire points at the top right quadrant of this space. The graph on the left shows skew-associative TLBs (blue) and hash-rehash TLBs (green), while the graph on the right shows MIX TLBs.

Figure 16 shows that MIX TLBs have better performance *and* energy than state-of-art multi-indexing schemes. Even the presence of operations like mirroring, which do increase energy by filling into multiple TLBs sets, are dwarfed by the big energy savings from decreasing TLB misses and hence cache/memory references. Another big source of energy savings comes from the relative simplicity of MIX TLB implementations; for example, we find that skew-associative TLBs suffer area overheads from requiring time-stamp counters for good replacement policies [37]. Therefore, area-equivalent skew-associative TLBs have fewer entries than MIX TLBs. Hash-rehashing is indeed more energy efficient than skew-associativity but still needs to access a predictor structure, hurting its energy compared to MIX TLBs.

Note also that multi-indexing schemes can degrade performance and energy. This occurs when TLB hits are frequent but have to go through more complex multi-step lookups when the page size predictor makes mistakes. MIX TLBs do not suffer from these problems.

Dynamic energy breakdown: MIX TLBs achieve energy efficiency from their shorter runtime, which reduces leakage energy. It is more challenging, however, to identify the sources of savings in dynamic energy because MIX TLBs do have some more sophisticated operations (e.g., mirroring). Figure 17 therefore quantifies the contribution of TLB

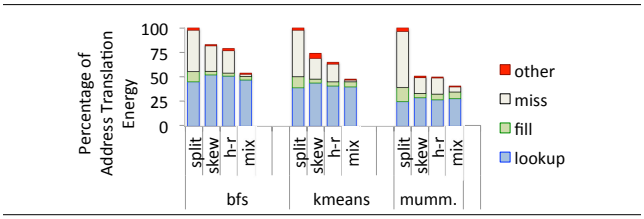

Figure 17. Percentage of address translation dynamic energy devoted to various TLB maintenance operations.

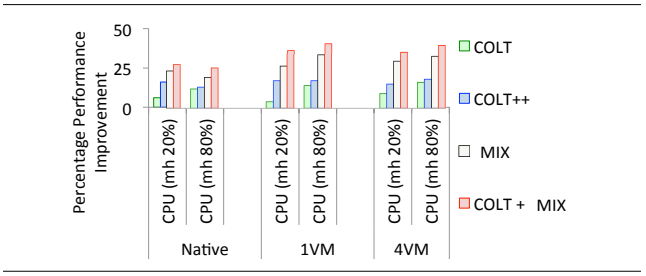

Figure 18. Compared to split TLBs, performance improvements from MIX TLBs and their combination with COLT.

energy devoted to lookups, page table walks, TLB fills after the miss, and other operations like TLB invalidations. We focus on GPU TLB results but the trends remain the same in other applications too. The y-axis is normalized to the total energy expended on Haswell split TLBs.

Figure 17 shows that most energy is used for lookups and misses. This is because all loads and stores result in lookups, while misses are expensive, invoking multiple memory references through the memory hierarchy. In contrast, the energy on TLB fill is much lower. Therefore, mirroring, which occurs only on fills, does not affect overall energy substantially. Note also that unlike multi-indexing approaches, which increase lookup energy due to complex accesses with predictors, MIX TLBs leave lookup energy largely unchanged.

Scaling TLBs: We now focus on studying how TLB scaling, specifically with the number of sets, impacts MIX TLBs. Naturally, with more sets, we need more superpage contiguity to coalesce sufficiently to offset mirroring. Therefore, beyond the 64-128 set count maintained by Sandybridge and Haswell systems, we have also studied hypothetical TLBs with 512 sets. In general, we find that even though many workloads do not exhibit sufficient superpage contiguity to completely offset 512 mirrors, they still achieve 80+ pages of contiguity. This is usually enough for good performance. We have found that 512-set TLBs achieve within 13% of the performance of ideal TLBs which never miss.

Complementing COLT: Finally, MIX TLBs are orthogonal to past work on coalesced TLBs or COLT [16]. The original COLT work proposed coalescing contiguous small page translations into single TLB entries. However, an extension, which we call COLT++, may also coalesce contiguous superpages in split TLBs. Each of the split TLBs independently performs coalescing on their respective page size translations. We quantify the benefits of these approaches in Figure 18, comparing them to two other data points. The first is an area-equivalent MIX TLB. The second combines COLT with MIX TLBs. In this approach, we design a single set-associative TLB that can support multiple concurrent page sizes; however, there we can also coalesce contiguous small pages. To compare fairly against past work [16], we assume that we can coalesce up to 4 contiguous small pages.

Figure 18 shows the average performance improvements of these various approaches versus Haswell-style split TLBs. We compare native and virtualized workloads, varying fragmentation with memhog. COLT can be helpful, but mostly when small pages dominate. In the presence of superpages, they cannot provide benefits. This explains the relatively low performance benefits when fragmentation is low (memhog 20%). COLT++ helps when superpages are frequent. On average, there are 8-10% performance differences versus COLT. However, MIX TLBs outperform even these cases because they can utilize all the TLB hardware for any distribution of page sizes. Further, combining MIX TLBs with COLT provide the highest performance, exceeding 20% benefits in all cases.

8. Conclusion

This work was motivated by the fact that modern TLB hardware is rigid in capacity allocation, despite the elasticity of the OS which can allocate many page size distributions. Many system factors affect these distributions, such as workload characteristics, system fragmentation and uptime, etc. There is a glaring gap between the richness of memory allocation at the software level, and modern TLB hardware.

We show one way of correcting this problem, with MIX TLBs, an energy-efficient TLB that uses all its resources to seamlessly adapt to any distribution of page sizes. We show its benefits for native CPUs, virtualized CPUs, and CPU-GPU systems. Further, we believe that its simple implementation makes MIX TLBs ready for quick adoption.

9. Acknowledgments

We thank Martha Kim, Ján Veselý, and Zi Yan for their insights and feedback during the preparation of initial drafts of this manuscript. We thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147, Google and VMware for its support.

References

[1] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.

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

[3] M. Talluri, S. Kong, M. Hill, and D. Patterson, "Tradeoffs in Supporting Two Page Sizes," *ISCA*, 1992.

[4] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Systems: Can You Have it Both Ways?," *MICRO*, 2015.

[5] D. Fan, Z. Tang, H. Huang, and G. Gao, "An Energy Efficient TLB Design Methodology," *ISLPED*, 2005.

[6] V. Karakostas, J. Gandhi, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal, "Energy-Efficient Address Translation," *HPCA*, 2016.

[7] T. Juan, T. Lang, and J. Navarro, "Reducing TLB Power Requirements," *ISLPED*, 1997.

[8] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen, "Generating Physical Addresses Directly for Saving Instruction TLB Energy," *MICRO*, 2002.

[9] A. Sodani, "Race to Exascale: Opportunities and Challenges," *MICRO Keynote*, 2011.

[10] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-Based Superpage-Friendly TLB Designs," *HPCA*, 2014.

[11] Intel, "Haswell," *www.7-cpu.com/cpu/Haswell.html*, 2016.

[12] Intel, "Skylake," *www.7-cpu.com/cpu/Skylake.html*, 2016.

[13] J. Gandhi, A. Basu, M. Hill, and M. Swift, "Efficient Memory Virtualization," *MICRO*, 2014.

[14] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications," *VMWare Technical Journal*, 2013.

[15] A. Seznec, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB," *IEEE Transactions on Computers*, 2004.

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

[17] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," *HPCA*, 2014.

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

[19] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," *MICRO*, 2013.

[20] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *ASPLOS*, 2008.

[21] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," *ASPLOS*, 2014.

[22] B. Pichai, L. Hsu, and A. Bhattacharjee, "Address Translation for Throughput Oriented Accelerators," *IEEE Micro Top Picks*, 2015.

[23] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," *HPCA*, 2014.

[24] N. Agarwal, D. Nellans, M. O'Connor, S. Keckler, and T. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," *HPCA*, 2015.

[25] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," *ASPLOS*, 2015.

[26] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," *Whitepaper*, 2012.

[27] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," *ISPASS*, 2016.

[28] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. Keckler, "Towards a High Performance Paged Memory for GPUs," *HPCA*, 2016.

[29] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal, "Redundant Memory Mappings for Fast Access to Large Memories," *ISCA*, 2015.

[30] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2016.

[31] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," *ASPLOS*, 2016.

[32] B. Romanescu, A. Lebeck, and D. Sorin, "Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency," *ASPLOS*, 2010.

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

[34] A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *ISCA*, 2012.

[35] A. Seznec, "A Case for Two-Way Skewed Associative Cache," *ISCA*, 1993.

[36] F. Bodin and A. Seznec, "Skewed Associativity Enhances Performance Predictability," *ISCA*, 1995.

[37] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," *MICRO*, 2010.

[38] R. Sampson and T. Wenisch, "Z-Cache Skewered," *WDDD*, 2011.

[39] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2011.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *PLDI*, 2005.

[41] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Simplications," *IISWC*, 2008.

[42] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, , and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," *ASPLOS*, 2012.

[43] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and

K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," *IISWC*, 2010.

[44] A. Arcangeli, "Transparent Hugepage Support," *KVM Forum*, 2010.

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

[46] A. Bhattacharjee, "Translation-Triggered Prefetching," *ASPLOS*, 2017.

[47] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, "Using TLB Speculation to Overcome Page Splintering in Virtual Machines," *Rutgers Technical Report DCS-TR-713*, 2015.

[48] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," *VEE*, 2015.

[49] F. Gaud, B. Lepers, J. Decouchant, J. Funston, and A. Fedorova, "Large Pages May be Harmful on NUMA Systems," *USENIX ATC*, 2014.

[50] J. Gandhi, M. Hill, and M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," *ISCA*, 2016.