2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

CoLT: Coalesced Large-Reach TLBs

Binh Pham* Viswanathan Vaidyanathan* Aamer Jaleel† Abhishek Bhattacharjee*
* Dept. of Computer Science, Rutgers University † Intel Corporation, VSSAD
{binhpham, viswav, abhib}@cs.rutgers.edu aamer.jaleel@intel.com

Abstract

Translation Lookaside Buffers (TLBs) are critical to system performance, particularly as applications demand larger working sets and with the adoption of virtualization. Architectural support for superpages has previously been proposed to improve TLB performance. By allocating contiguous physical pages to contiguous virtual pages, the operating system (OS) constructs superpages which need just one TLB entry rather than the hundreds required for the constituent base pages. While this greatly reduces TLB misses, these gains are often offset by the implementation difficulties of generating and managing ample contiguity for superpages.

We show, however, that basic OS memory allocation mechanisms such as buddy allocators and memory compaction naturally assign contiguous physical pages to contiguous virtual pages. Our real-system experiments show that while usually insufficient for superpages, these intermediate levels of contiguity exist under various system conditions and even under high load. In response, we propose Coalesced Large-Reach TLBs (CoLT), which leverage this intermediate contiguity to coalesce multiple virtual-to-physical page translations into single TLB entries. We show that CoLT implementations eliminate 40% to 58% of TLB misses on average, improving performance by 14%.

Overall, we demonstrate that the OS naturally generates page allocation contiguity. CoLT exploits this contiguity to eliminate TLB misses for next-generation, big-data applications with low-overhead implementations.

1. Introduction

Translation Lookaside Buffers (TLBs) are crucial to system performance due to their long miss penalties [5, 11, 14, 19, 22, 26]. Past work has shown that TLB misses degrade performance by 5% to 14% for even nominally-sized applications. This number worsens to 50% in virtualized environments or when the application's memory footprint increases [8, 21].

Superpages have previously been proposed to increase TLB coverage [15, 16, 23, 25, 28, 29]. A superpage is a memory page that is sized as a multiple of a base page and is typically in the megabyte or gigabyte range. Each TLB superpage entry can thus replace hundreds of baseline TLB entries, boosting TLB coverage.

By construction, superpages target situations where the OS can seamlessly generate vast amounts of contiguity. Unfortunately, a number of issues may preclude this. For example, superpages may magnify an application's memory footprint, increasing paging traffic. Superpages also require specialized and high-overhead algorithms to assign aligned and contiguous physical page frames to contiguous virtual pages [25, 28, 29]. The OS therefore selectively uses superpages such that the benefits of reduced TLB misses are not outweighed by these overheads.

This work observes that there actually exists a second regime of page allocation contiguity, orthogonal to superpaging. Specifically, OS memory allocators use buddy allocators and memory compaction daemons which also, by construction, allocate contiguous physical page frames to contiguous virtual pages. These mechanisms generate *intermediate contiguity* (in the range of tens of pages), which falls short of superpage requirements (hundreds of contiguous pages). However, this contiguity is achieved without superpaging overheads like increased I/O traffic and sophisticated page construction algorithms. In response, we propose Coalesced Large-Reach TLBs (CoLT), a series of hardware mechanisms that allow TLBs to coalesce multiple contiguous virtual-to-physical address translations, increasing their memory reach. CoLT specifically targets large amounts of intermediate contiguity that superpaging cannot exploit. Our contributions are as follows:

- We study, on a real system, how often consecutive virtual pages are allocated consecutive physical pages. We find that even with high system fragmentation, tens of pages are usually contiguous. Furthermore, while superpaging does increase contiguity, much of it falls short of the level necessary to actually create large pages (a 2MB superpage needs 512 contiguous 4KB base pages, while we see tens of contiguous pages). Instead, we show that TLB coalescing effectively leverages this contiguity.
- We propose CoLT for set-associative, two-level TLB hierarchies commonly found in processors today [3, 17]. Our strategies eliminate roughly 40% of L1 and L2 TLB misses, resulting in average performance improvements of 12%.
- We develop CoLT support for small, fully-associative TLBs commonly found in processors to cache superpage entries [3, 17]. We show how to overcome the challenge of designing these small structures, achieving L1 and L2 TLB miss elimination rates of 58% on average. These translate to average performance improvements of 14%.
- Finally, we combine the benefits of coalescing on both set-associative and fully-associative TLBs, improving average performance by 14%.

Overall, we design low-overhead hardware to exploit intermediate levels of page allocation contiguity. Our studies evaluate this approach under a variety of system configurations with heavy system load and different superpaging configurations.

2. Background and Related Work

2.1. Prior TLB Enhancement Techniques

Address translation, especially with virtualization and larger application working sets, is a primary source of system performance degradation [8, 21]. In response, researchers have considered techniques to improve TLB structure, lookup, and placement [9, 13]. More sophisticated techniques such as TLB prefetching and mechanisms to accelerate page walks have also been considered [5, 11, 19, 27]. Our goal is to propose techniques, orthogonal to past work, to further boost TLB performance.

2.2. Superpaging Benefits and Problems

Superpages or hugepages have previously been proposed to lower TLB miss rates [15, 16, 23, 25, 28, 29]. Superpages are typically

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.32

sized as power-of-two multiples of baseline pages. For example, x86 systems use 4KB baseline pages and support 2MB and 1GB superpages. Furthermore, superpages must be aligned in both virtual and physical memory (superpages of size N must begin at virtual and physical addresses that are multiples of N).

While superpages lower TLB miss rates by replacing hundreds to thousands of base page translations with a single superpage translation entry, they have management overheads [23, 28]. For example, dedicated OS code is required to support multiple large page sizes [25, 28, 29]. The process of ensuring that sufficient contiguous physical pages are allocated to virtual pages can suffer high performance overheads, particularly when alignment restrictions are also imposed [15, 23]. Furthermore, if not fully-utilized, superpages can increase the amount of I/O traffic and increase page initialization/fault latency. A particular problem is the use of a single dirty bit for all the baseline pages of a superpage; if set, the entire superpage must be written back to disk even if only one base page has been modified, greatly increasing disk traffic. Therefore, the OS weighs these overheads against the benefits of superpaging. Typically, the OS uses superpages sparingly, only bothering to generate large amounts of contiguity when superpaging is deemed to be worthwhile [4].

2.3. TLB Subblocking and Speculation

Two hardware schemes have been proposed to mitigate superpaging overheads. Talluri and Hill [28] present complete-subblock and partial-subblock TLBs, which record ranges of physical pages per virtual page entry. Complete subblocking, while effective, requires non-trivial modifications to traditional TLB hardware. Partial-subblocking overcomes these overheads but with explicit OS support [28]. Furthermore, subblocking was originally proposed for fully-associative designs rather than the set-associative organizations most commonly used in products today. Finally, sub-blocking effectiveness depends heavily upon page alignment. Both complete and partial subblocking are most effective when the first virtual page of a contiguous group of virtual (and physical pages) is aligned to the subblock length. Partial subblocking goes beyond this, reducing hardware overheads by requiring that base physical pages also be placed in an aligned manner within subblock regions [28]. Overall, these requirements may constrain exploitable contiguity.

Alternately, past work [6] proposes TLB speculation for systems with reservation-based superpaging [23, 28]. Here, physical pages are allocated in aligned 2MB regions of physical memory corresponding to their alignment within a 2MB region of virtual memory. Barr, Cox, and Rixner exploit this property with a SpecTLB structure, which interpolates between existing TLB entries to predict physical translations for TLB misses. While effective, SpecTLB requires reservation-based superpaging, which is not universally used (eg. Linux superpaging [4] does not use reservation-based superpages). SpecTLB also requires an additional TLB-like structure, and can increase instruction replays on incorrect speculations.

2.4. Our Approach

In general, there are three regimes of page contiguity that the OS generates. In the first regime, the OS does not succeed in mapping contiguous physical pages to contiguous virtual pages. In these cases, traditional techniques such as changing TLB organizations or prefetching are likely to be successful in increasing hit rates [6, 11, 19]. At the other end of the spectrum, there exists a regime of extremely high contiguity, when the OS decides that the overheads of superpage construction are well worth the effort. Our goal, in this work, is to exploit a third regime with intermediate contiguity, where tens to hundreds of base pages are contiguous. Our characterization studies will show that the OS naturally produces this contiguity without the overheads of superpages and that this level of contiguity is more prevalent than the other regimes. Our studies will be on a real system under a comprehensive set of system environments which include heavily-fragmented systems and systems with and without superpaging.

We will then realize low-overhead TLB hardware to exploit intermediate contiguity. Unlike prior work on speculation or prefetching [6, 11, 19], CoLT does not augment the standard TLBs with separate structures. Unlike superpages or subblocking, we avoid OS intrusion and do not require prescribed amounts of contiguity. Instead, CoLT studies available contiguity and exploits as much of it as possible.

3. Understanding Page Allocation Contiguity

We now explore why operating systems often allocate contiguous physical page frames to contiguous virtual pages. Since CoLT relies on this behavior, we ascertain which memory allocation policies and mechanisms produce contiguity.

3.1. Defining Page Allocation Contiguity

We say that system contiguity exists when consecutive virtual pages are allocated consecutive physical page frames. For example, if virtual pages *1*, *2*, and *3* are allocated physical page frames *58*, *59*, and *60*, we say that these pages are contiguous. Moreover, since this example involves three pages, we say that this is an instance of *3-page* contiguity.

Our definition is distinct from superpages in two ways. First, superpages require a set amount of contiguity. For example, 2MB superpages on x86 systems require instances of *512-page* contiguity. Instead, we make no restrictions on the amount of contiguity that is useful. Second, unlike superpages, we make no assumption on alignment. Our relaxations on contiguity amounts and alignment restrictions reveal huge great intermediate contiguity.

Note that this definition requires simultaneous contiguity in both virtual *and* physical page numbers. As such, CoLT does not affect cases where only virtual (or physical) pages are contiguous; these will be cached by TLBs in the conventional manner.

3.2. Sources of Page Allocation Contiguity

Operating systems maintain a complex set of policies and mechanisms to determine how to allocate physical memory to applications. A number of these policies promote physical page contiguity. We elaborate on them here, focusing on Linux for our discussion. Note, however, that our observations extend more broadly to other operating systems which tend to utilize similar mechanisms to allocate physical pages.

3.2.1. Buddy allocation. Linux uses a buddy allocator to track physical pages and assign them to virtual pages on demand. Figure 1 illustrates the operation of a buddy allocator, assuming that pages 1, 2, and 3 are already allocated. All free physical pages or page frames (PFs) are grouped into ten lists of blocks, which we refer to as *free lists*. Entry x in the free list tracks groups of 2^x *contiguous* physical pages. For example, pages 4-7 have 4-page contiguity and are hence listed by entry two.

Physical page allocations proceed as follows. Suppose an application requires an N-page data structure. To accommodate this, the

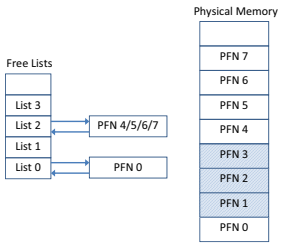

Figure 1: Buddy allocator used for physical page allocation. Already allocated pages are shaded, while free pages are tracked by the free lists.

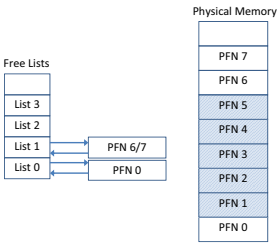

Figure 2: Buddy allocator state after an allocation for 2 pages is finished.

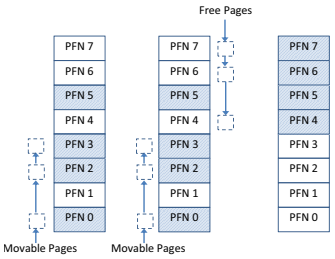

Figure 3: The memory compaction daemon tracks movable and free memory pages, exchanging them to eliminate fragmentation.

application makes a `malloc` call, simultaneously requesting N physical pages from the OS. The buddy allocator first searches the free list entry corresponding to the smallest contiguous page frames bigger than N (entry $\lceil log_2(N) \rceil$). If a block of free physical pages is found in that list, allocation successfully completes. Otherwise, the free list is progressively climbed until an entry with a block of free contiguous physical pages is found. Once a free block is found, the buddy allocator must minimize memory fragmentation. Therefore it iteratively halves the block, inserting these new blocks in their appropriate free list locations, until it extracts a block of N contiguous physical pages. As an example, Figure 2 shows the state of the free list after an application level request for two physical pages to be allocated. At first, entry 1 in the free list is checked; however, since this is empty, entry 2 is scanned. Here, a free block with contiguous physical pages 4, 5, 6, and 7 is found. Hence, the buddy allocator halves this block of four pages, returning pages 4 and 5 to the application and moving pages 6 and 7 to free list entry 1. Apart from allocation, the buddy allocator also updates its state when physical pages are released. At this point, the kernel attempts to merge pairs of free *buddy* blocks if both have the same size and are contiguous.

By construction therefore, the buddy allocator deliberately provides contiguous physical page frames to the application when it asks for multiple page frames together. Since applications usually make `malloc` calls that simultaneously request a number of physical pages together (rather than one page at a time), the buddy allocator is able to provide them a suitable contiguous range of pages. We will show that the buddy allocator successfully produces this contiguity even in the presence of significant system load. While insufficient for superpages, CoLT benefits from this substantially.

3.2.2. Memory compaction. In order to glean contiguous runs of physical pages, the buddy allocator relies on ensuring that memory fragmentation is tightly controlled. However, fragmentation is pronounced when multiple processes with large working sets simultaneously run on the system. Therefore, many operating systems boost the buddy allocator with a separate memory compaction daemon. Figure 3 details the Linux memory compaction daemon in three steps on a heavily-fragmented system.

First, as shown in the left-most diagram of Figure 3, memory compaction runs an algorithm that starts at the bottom of the physical memory and builds a list of allocated pages that are *movable*. While most user-level pages are movable, pinned and kernel pages usually are not. Nevertheless, user-level pages usually outnumber kernel pages, making most pages movable.

Second, the daemon starts at the top of physical memory and builds a list of free pages. Eventually, the two algorithms meet in the middle of the physical page list. At this point, Linux invokes migration code to shift the movable pages to the free page list, yielding the unfragmented diagram at the right of Figure 3.

Since there is a cost associated with moving pages, the compaction daemon is only triggered when there is heavy system fragmentation. As such, its operation naturally produces contiguity, especially in tandem with the buddy allocator. In fact, we will show that this daemon successfully generates contiguity even under heavy system fragmentation.

3.2.3. Transparent hugepage support. Aside from buddy allocation and memory compaction, support for superpages is a primary cause of page allocation contiguity. Unfortunately superpage management comes with high overheads. As a result, Linux's Transparent Hugepage Support (THS), supported since the 2.6.38 kernel [4], uses superpages sparingly. When THS is enabled, the memory allocator attempts to find a free 2MB block of memory. If this block is naturally aligned at a 2MB boundary, a superpage is constructed. In practice, the OS relies on the memory compaction daemon to construct these 2MB regions. Aligned 2MB regions are rare; when a superpage cannot be constructed, the system defaults to the buddy allocator. Even when the 2MB pages are allocated, increased load can eventually make them harmful. In these cases, system pressure triggers a daemon that breaks superpages into baseline 4KB pages.

In practice, THS struggles to maintain many superpages simultaneously. However, it does succeed in creating additional levels of contiguity for two reasons. First, while optimistically-allocated 2MB superpages are often eventually split due to system pressure, they retain contiguity among tens of baseline 4KB pages. Second, THS relies on the memory compaction daemon, triggering it more often and providing the buddy allocator even higher levels of contiguity suitable for CoLT.

3.2.4. System Load and Memory Fragmentation. Finally, page allocation contiguity is deeply affected by the system load. If many processes run simultaneously, main memory is likelier to be fragmented. Therefore, one may initially expect that higher load degrades contiguity. Surprisingly, we will show that contiguity can actually *increase* with greater system load. This occurs because system load has a complex relationship with the memory compaction daemon, triggering it more often when there is higher load. This can, in turn, free up more contiguous physical frames for the buddy allocator, eventually resulting in more contiguity.

4. CoLT Design and Implementation

Having detailed contiguity sources, we now propose three variants of CoLT. Overall, they share three design principles. First, they detect instances of consecutive virtual-to-physical address translations. These entries are coalesced into single TLB entries, so as to reduce miss rates. Second, CoLT coalesces only on *TLB misses*. While TLB hits could also prompt coalescing, this may increase

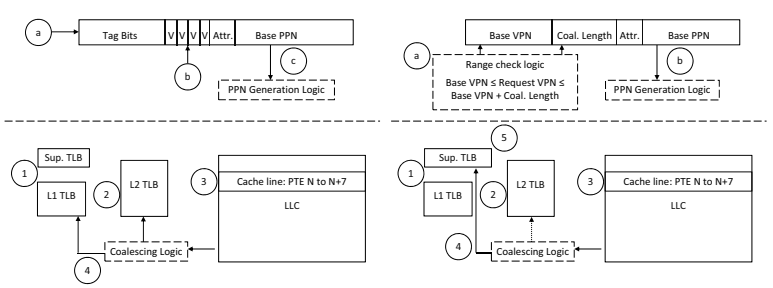

Figure 4: CoLT for set-associative L1 and L2 TLBs.

Figure 5: CoLT for fully-associative TLBs.

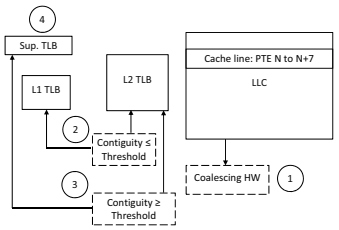

Figure 6: Combined CoLT for all TLBs.

lookup latencies. Third, coalescing is *unintrusive*, unlike speculation and prefetching [6, 11, 19, 27] which can degrade performance. For example, incorrect speculations suffer a high penalty. Incorrect prefetches lead to the eviction of useful entries and higher bandwidth usage. Prior work mitigates these problems by using separate structures to store prefetched translations or perform speculation [6, 11, 19]. In contrast, we coalesce entries directly into the TLBs but ensure that coalescing occurs only around *on-demand* translations. In the worst case, coalesced entries may be unused but are not harmful. This is crucial given that system contiguity does not necessarily imply that all contiguous translations are used in temporal proximity. We ensure coalesced entries are available if needed but do not harm TLB hit rates when they are unused.

We propose three variants of CoLT for two-level TLB hierarchies. This hierarchy contains set-associative L1 TLB and L2 TLBs, used to cache baseline 4KB pages [3, 17]. Superpages are cached in separate small, fully-associative TLBs that are accessed in parallel with the L1 TLB. Note that the L2 TLB is inclusive of just the set-associative L1 TLB and not the superpage TLB.

There are three natural coalescing mechanisms for this hierarchy. First, we coalesce in just the set-associative L1 and L2 TLBs. Second, we coalesce in the superpage TLB only. Third, we use a combined approach that routes some coalesced entries to the set-associative TLBs and others to the superpage TLBs. We now describe each of these schemes.

4.1. CoLT-SA Design and Implementation

CoLT-Set Associative (CoLT-SA) coalesces multiple virtual-to-physical page translations in the set-associative L1 and L2 TLBs. We first detail its high-level operation and then focus on specific design challenges.

4.1.1. Overall operation. The bottom half of Figure 4 shows a high-level view of CoLT-SA. In step 1, the set-associative L1 TLB and superpage TLB are looked up in parallel. Assuming L1 and L2 TLB misses (step 2), a page table walk brings in the desired translation entry into the LLC (step 3). We assume, like most x86 systems with dedicated MMU page table caches [5], that the LLC is the highest cache level for page table entries.

After the LLC fill, two parallel events occur. First, the requested translation is returned to the processor pipeline. In parallel, the *Coalescing Logic* studies the translations around the requested entry for contiguity. It coalesces as many of these translations as possible, as long as they map to the same set. This coalesced entry is inserted into the L1 and L2 TLBs (step 4). However, conventional set-associative TLBs map consecutive virtual addresses (and hence contiguous translations) to consecutive sets, precluding coalescing. We therefore modify the virtual page bits used for set-selection so that translations for groups of consecutive virtual page numbers do map to the same set. Furthermore, since we provide the requested translation to the pipeline in parallel with the *Coalescing Logic's* operation, the latter is off the critical path and does not affect TLB miss handling times.

4.1.2. TLB set selection. To understand how we modify TLB set selection to permit coalescing, consider the following example. An 8-set TLB would require three bits, bits 2 to 0 of the virtual page number for set selection (VPN[2-0]). Naturally, this would map consecutive translations to consecutive sets, preventing coalescing. However, if we left-shift the index bits by $log_2(N)$ bits, we may place N consecutive translations in the same set (permitting a maximum of N contiguous translations to be coalesced into a single entry). Therefore, to ensure that translations with four consecutive virtual pages map to the same set, we use VPN[4-2] as the new indexing bits.

To coalesce more entries, the indexing bits are further left-shifted (for example, to coalesce up to eight entries, VPN[5-3] must be used). However, using higher order bits for set indexing increases conflict misses since more consecutive entries are mapped to the same set. This is a fundamental tradeoff for CoLT-SA designs – in choosing the correct index bits, we must balance opportunities for coalescing with potentially higher conflict misses. We will show that allowing for coalescing of four contiguous translations generally performs best.

4.1.3. Lookup operation. The top half of Figure 4 illustrates CoLT-SA lookups. Each coalesced TLB entry maintains tag bits, the higher order bits left of the index bits used for set selection. For example, if up to four contiguous translations can be coalesced in a TLB with eight sets, VPN[4-2] is used for set selection and VPN[63-5] is the tag. In step (a), this tag is checked against the requested virtual page number. In step (b), the non-index lower-order virtual page bits (VPN[1-0] in our example) are used to select among multiple valid bits. There is one valid bit for every possible translation in a coalesced entry. These valid bits indicate the presence of a translation in the coalesced entry. If on step (b), a valid bit is set, there is a TLB hit. At this point, extra logic calculates the physical page number. CoLT entries store the base physical page number for each coalesced entry. This number corresponds to the virtual page represented by the first set valid bit. To reconstruct the physical page number, combinational logic (*PPN Generation Logic*) calculates the number of valid bits away this entry is from the first set valid bit. This number is added to the stored base physical page number to yield the desired physical page.

4.1.4. Practical coalescing restrictions. Ideally, after the page table is walked to handle a TLB miss, coalescing logic finds as many contiguous translations around the requested translation as possible. Practically, however, coalescing is restricted by two constraints. First, as we have already discussed, the choice of index bits for set selection places a limit on coalescing opportunity. A second limit arises from our desire to minimize the overhead associated with

searching for contiguous translations. On a TLB miss, a page table walk finds the desired translation. We aim to prevent any additional page walks when checking for contiguous entries adjacent to the requested translation. Since the page table walk accesses the last-level cache (LLC) and brings data in 64-byte cache lines, seven additional translations are fetched. These translations are brought without additional memory references; thus we check just them for contiguity. In practice, this approach restricts coalescing to a maximum of eight translations. Despite this restriction, CoLT eliminates a high number of TLB misses.

4.1.5. Replacement, invalidations, and attribute changes. CoLT-SA assumes standard LRU replacement policies. While there may be benefits in prioritizing entries with different coalescing amounts differently, we leave this for future work. We also assume a single set of attribute bits for all the coalesced entries, restricting coalescing opportunity. More sophisticated schemes supporting separate attribute bits per translation in a coalesced entry will improve our results. Furthermore on TLB invalidations, we flush out entire coalesced entries, losing information for pages that would be unaffected in standard TLBs. Gracefully uncoalescing TLB entries and only invalidating victim translations will perform even better. This too is the subject of future work.

4.1.6. Discussion of hardware overheads. To accommodate CoLT, the TLBs experience some key hardware changes. We argue, however, that these changes are modest. First, we believe that CoLT lookup remains low-overhead and does not impact TLB access cycle times. The initial tag match and check of valid bits is simple. The PPN generation logic addition is also low-overhead as the amount of coalescing is bounded (in our example, at best, an addition of four will be required). As such, readily-implementable combinational logic, similar to logic used to calculate prefetching strides and addresses or update branch predictor state, can calculate the physical page number. This is lower-overhead than prior prefetching schemes requiring dedicated adders [19].

Second, coalescing logic occurs on the TLB fill path rather than lookup, allowing subsequent TLB reads to proceed unimpaired. It is possible for subsequent reads to request translations that are part of the requested coalesced entries. These reads must wait until coalescing completes but we find these instances occur rarely. Furthermore, one might expect CoLT to require additional TLB ports to fill entries without conflicting with subsequent TLB reads. In our results, we assume no additional ports, finding that there is no significant performance degradation from this.

4.2. CoLT-FA Design and Implementation

Rather than supporting coalescing in set-associative TLBs and changing indexing schemes, we can instead coalesce into the fully-associative TLB (this structure is usually used exclusively for superpages). We refer to this as CoLT-Fully Associative (CoLT-FA).

4.2.1. Overall operation. The bottom half of Figure 5 delineates CoLT-FA operation. Assuming misses in all the TLBs (steps 1 and 2), a page walk is conducted in step 3. At this point, a cache line provides up to eight translations that can be checked for contiguity. Up to eight translations are now coalesced in step 4. If coalescable, the entry is loaded into the fully-associative TLB. If no coalescing is possible, it is loaded into the set-associative L1 and L2 TLBs.

On insertion into the fully-associative TLB, further coalescing is possible. Since contiguity may exist between the newly coalesced entry and a resident entry, the fully-associative TLB is scanned to seek further opportunities for coalescing. This scan is conducted while the requested TLB entry is returned to the processor. Further coalescing from the scan is done in step 5.

Empirically, we have found that due to the small size of the superpage-TLB, useful entries are frequently evicted. Therefore, for performance reasons, when bringing a coalesced entry into the fully-associative structure, we still bring just the requested entry (and not its coalesced neighbors) into the L2 TLB. While this does create some redundancy in terms of stored entries, we will show that performance is improved. Note that we leave the L1 TLB unaffected due to its much smaller capacity. Note also that CoLT-FA shares both superpage entries and coalesced entries in a single structure. One initial concern may be that if coalesced entries far outnumber superpage entries, the latter will be evicted from the fully-associative TLB. In practice, we find that this is not a problem for two reasons. First, superpages are used sparingly, requiring a very small number of entries in the buffer. Second, when used, these superpages are frequently accessed, meaning that they remain at the head of the LRU list, preventing their eviction.

4.2.2. Lookup operation. The top half of Figure 5 details CoLT-FA lookup. Each coalesced entry maintains a base virtual page number as the tag and a field that logs the number of entries coalesced. Unlike CoLT-SA, there are no coalescing restrictions due to indexing schemes. We find that using five bits for the coalescing length field suffices as this captures a contiguity of 32 pages. Each entry also stores the base physical page and attributes of all contiguous translations.

In step (a), *Range Checking Logic* compares the requested virtual page number against the range of translations stored by each entry of the fully-associative TLB. As shown, comparator and adder logic is required for the range check. If the virtual page is detected in the range, there is a TLB hit. At this point (step (b)), the *PPN Generation Logic* subtracts the tag base virtual page number from the requested virtual page number. This value is then added to the stored base physical page number to find the desired physical page.

4.2.3. Replacement, invalidations, and attribute changes. We assume standard LRU for the fully-associative structure. Due to its smaller size, we suspect though that smarter replacement policies will be even more effective. Furthermore, we share the same attribute bits for all coalesced entries and invalidate entire entries, but for larger amounts of coalescing. Despite this, we will show that CoLT-FA performs effectively.

4.2.4. Discussion of hardware overheads. Generally, CoLT-FA hardware for range checks and physical page number generation is more complex than CoLT-SA. To account for this, we reduce the size of the fully-associative TLB in CoLT-FA as compared to the baseline case without coalescing. Commercial systems tend to implement 16 to 24-entry fully-associative TLBs for superpages [17]. To ensure that the added lookup complexity does not bias our results, we assume only 8-entry fully-associative TLBs with coalescing. While a detailed circuit-level analysis of the lookup overhead is beyond the scope of this work, our decision to apply CoLT to a half-sized fully-associative TLB attempts to maintain the same access times.

An additional overhead arises from the secondary scan performed between existing fully-associative TLB entries and the coalesced entry being filled. While one might initially assume that we may need to increase port counts to ensure that subsequent lookups are not delayed, our implementation retains just the single port. Instead, we assume that the initial lookup of the fully-associative TLB iden-

tifies those resident entries likely to be coalescible with the filled entry. Once the coalescing logic merges translations from a single LLC cache line, it then checks whether those resident entries can be further coalesced. In this way, a second TLB scan can actually be avoided, minimizing coalescing overheads.

4.3. CoLT-All Design and Implementation

Finally, CoLT-All coalesces into both set-associative L1/L2 TLBs and the superpage TLB. Its primary benefit over CoLT-SA and CoLT-FA is that it provides potentially the largest reach, at the expense of modifying both the set-associative and superpage TLB.

4.3.1. Overall operation. Figure 6 illustrates CoLT-All's operation when all the TLBs experience a miss. In step 1, the page walk has occurred and the coalescing hardware has determined the amount of contiguity present in the cache line. In then checks this contiguity to see how it compares to a threshold. If it is lower than a threshold (step 2), this means that the contiguity can be accommodated by the indexing scheme of the set-associative TLBs. For example, suppose the contiguity is three pages and we use an 8-set TLB with VPN[4-2] for indexing (allowing coalescing of up to four translations). In this case, the coalesced entry is allocated into the set-associative L1 and L2 TLBs. However the contiguity may be higher than the threshold and the amount that the set-associative TLBs can accommodate. In our example, the contiguity may be five. In this case, the entry is coalesced and brought into the superpage-TLB. At the same time, because the superpage-TLB is small, useful coalesced entries may be frequently evicted. Therefore, like CoLT-FA, we allocate an entry at this point into the L2 TLB as well. Unlike CoLT-FA however, our set-associative L2 TLB can now also handle coalesced entries (albeit with smaller levels of coalescing permissible by its choice of index bits). Therefore, CoLT-All brings in as much of this coalesced entry as possible into the L2 TLB, unlike CoLT-FA which brings just the requested translation. Finally, in step 4, the new allocated superpage entry may be coalesced with already-resident entries.

4.3.2. Lookup, replacement, invalidation, and attributes. While lookups operate similarly to CoLT-SA and CoLT-FA, the only difference is that it is possible for an entry to be resident in both the set-associative and fully-associative TLBs. While this occurs only rarely in practice, there are no correctness issues associated with this. Furthermore, there are no changes in replacement, invalidation, and attribute policies.

5. Methodology

We now detail the infrastructure and workloads used to quantify real-system contiguity and CoLT's effectiveness at leveraging this contiguity to eliminate TLB misses. Our analysis focuses on data pages since data references cause far more misses than instruction references [10, 27].

5.1. Real-System Characterizations of Page Allocation

5.1.1. Experimental platform and methodology. We use a system with a 64-bit Intel i7 processor, 64-entry L1 TLBs, and a 512-entry L2 TLB, a 32KB L1 cache, a 256KB L2 cache, a 4MB last-level cache (LLC), and 3GB of main memory. Furthermore, we run Fedora 15 (Linux 2.6.38).

To measure contiguity, we modify the kernel to scan the page table looking for instances of contiguous address translations. We walk the page table every five seconds, capturing contiguity changes through the benchmark run. Our original definition of contiguity is based only on page numbers; however, we now additionally require that contiguous translations must share the same page attributes and flags. While this eases the hardware implementation of CoLT by allowing for the same set of attribute bits per coalesced entry, contiguity would be even higher if this constraint were relaxed.

Benchmark	Suite	THS on L1/L2 MPMI	THS off L1/L2 MPMI
Mcf	Spec	56550/28600	95600/49230
Tigr	BioB.	19000/18150	26950/18860
Mummer	BioB.	12910/11450	14760/12970
CactusADM	Spec	6610/8140	8420/6930
Astar	Spec	8480/4660	17390/11240
Omnetpp	Spec	8410/2730	34040/8080
Xalancbmk	Spec	2670/2150	14120/2100
Povray	Spec	7010/630	7310/630
GemsFDTD	Spec	1300/620	8030/3620
Gobmk	Spec	710/410	1550/510
FastaProt	BioB.	460/300	610/300
Sjeng	Spec	1840/200	3860/440
Bzip2	Spec	4070/150	7120/270
Milc	Spec	120/90	3780/1820

Table 1: Summary of benchmarks used in our studies.

To study the effect of memory compaction, we use the Linux `defrag` flag. Enabling this flag triggers the memory compaction daemon both on page faults and as system background activity. Disabling this flag greatly reduces the number of times the memory compaction daemon runs. In tandem, we enable and disable THS to study the impact of superpaging. We also ensure that our system is realistically fragmented by using a machine that has already run a number of applications (eg. web browsers, network clients, office utilities) for two months. To further load the system, we run `memhog`, a memory fragmentation utility [12], with our workloads. We study scenarios where `memhog` fragments 25% and 50% (a highly fragmented system when combined with the other background activities) of the memory. In all, we study twelve system configurations. Due to space constraints, this paper focuses on:

1. THS on, normal memory compaction, no memhog: this is the current default setting for Linux.

2. THS off, normal memory compaction, no memhog: this shows contiguity *without* superpaging.

3. THS off, low memory compaction, no memhog: conservative case for contiguity because neither THS no memory compaction occur. The buddy allocator struggles to find contiguous physical blocks.

4. THS on, normal memory compaction, memhog: we test the effect of system load on the default Linux setting by assigning 25% and 50% of system memory to memhog.

5. THS off, normal memory compaction, memhog: shows the impact of fragmentation without superpaging.

5.1.2. Evaluation workloads. We study system contiguity on the Spec 2006 benchmarks [1] and bioninformatics workloads from Biobench [2] in Table 1. We run each of the workloads with their maximum data sets (for Spec, this corresponds to *Ref*) to completion. From the real-system runs, we use on-chip performance counters to track L1 and L2 TLB misses per million instructions (MPMI) when THS support is enabled and disabled. The benchmarks are ordered from highest to lowest THS on L2 TLB MPMIs. `Mcf`, `Tigr`, `Mummer`, `CactusADM`, and `Astar` see particularly high TLB MPMIs. While

enabling superpaging does reduce TLB misses for some workloads, it alone is insufficient. For example, `Mcf` still has an L2 TLB MPMI of 57K with THS on, while `Mummer` is unchanged.

5.2. Simulation-Based CoLT Evaluations

5.2.1. Simulated system. Past work on TLBs [5, 6, 9, 27] focuses on miss rates rather than performance because it is infeasible to run memory-intensive applications for long enough durations to provide performance numbers. We also study miss rates, but consider performance too. We use a two-step evaluation to quantify changes in hit rate and to then offer performance numbers feasible for simulations.

Like the bulk of recent work on TLB analysis, we first use a trace-based approach to analyze miss rates [5, 6, 9, 10]. We extract detailed memory traces by simulating an x86 processor on Simics [30]. These highly detailed traces maintain logs of both data and instruction references at the micro-op level. Our traces also capture full-system effects by running benchmarks on a Linux 2.6.38 kernel. We hack the simulated kernel to provide full page table walk details for every single memory reference (this includes the virtual page, the physical page, and all attribute bits). We set the kernel to its default configuration of using THS and normal memory compaction. As we will show, since contiguity is present across all kernel configurations, CoLT will be effective across the range of superpaging and memory compaction settings.

We run the traces through a highly-detailed custom memory simulator. We need to stress our TLBs using simulated workloads in a manner that matches real-system stress; therefore, we use 32-entry and 128-entry L1 and L2 4-way set-associative TLBs. These sizes are chosen as they produce simulated load within 10% of the load experienced by a real system. Our baseline system also assumes a 16-entry fully-associative superpage TLB. As previously detailed, CoLT-FA and CoLT-All reduce this size to 8 entries in order to provide conservative performance improvement data and negate the impact of slightly more complex lookups. Furthermore, unlike past work [9, 11], we model a more realistic TLB hierarchy with 22-entry MMU caches, accessed on TLB misses to accelerate page table walks [5]. Finally, we assume a three-level cache hierarchy similar to the Intel Core i7 (32KB L1 cache, 256KB L2 cache, 4MB LLC).

Having assessed miss rates, we now go beyond prior work and the study the performance implications of our approach. We use the Pin-based [20] CMP$im [18] simulation framework to model a 4-way out-of-order processor with a 128-entry reorder buffer. The processor's TLB and cache parameters match those of our custom trace module. Unfortunately, the simulation speeds of this detailed microarchitectural framework are slow; hence we cannot use it to run full Linux distributions with the memory allocation behavior necessary to study CoLT on sufficiently long-running, large-data applications. However, while this simulator does not maintain virtual-to-physical address translations, it does observe the performance effects of TLB misses by tracking the allocated virtual pages. In tandem with the miss rate eliminations extracted from our trace-based approach, this allows us to interpolate CoLT's actual performance gains. This interpolation strategy is valid for two reasons. First, TLB miss penalties (page walks) are serialized as only one page walk can typically be handled at a time [9, 11]. Hence, TLB misses lie on the execution's critical path. Second, our interpolation approach is actually conservative as it does not account for the instruction replays that would likely occur on TLB misses. Therefore, our projected performance benefits would likely *increase* on a real system.

5.2.2. Evaluation workloads. We use the workloads from Table 1. However, due to slow simulation speeds, we use Simpoints [24] that total to one billion instructions per workload. These simpoints include operating system effects captured by Simics and assume realistic inputs (for Spec, this corresponds to *Ref*).

6. Characterizations of Page Allocation Contiguity

We now quantify how the buddy allocator, memory compaction, THS, and system load affect application contiguity on a real system. We show that page allocation contiguity *always* exists regardless of the kernel configuration.

We begin by discussing the cumulative density functions (CDFs) from Figures 7 to 15. These graph the distribution of contiguities experienced by non-superpage pages. Note that contiguity (the x-axis) is presented as a log scale.

6.1. Superpaging, Memory Compaction

Figures 7, 8, and 9, ordering the benchmarks from highest to lowest TLB MPMI, show contiguity assuming default Linux kernel settings (superpaging and normal memory compaction). The legend provides average contiguity numbers.

Figures 7, 8, and 9 show that there is heavy contiguity across the workloads that cannot be exploited by superpages. On average, pages are in 41-contiguity groupings. Furthermore, there can be large instances of contiguity above the average. For example, most CDFs see many 64 to 256-contiguity instances.

Interestingly, there exist many cases of 512 and 1024-page contiguity. Since THS is enabled, one might initially expect that these should be treated as superpages. However, this contiguity does not translate to superpages for two reasons. First, these memory chunks are not superpage-aligned. Second, THS currently supports superpaging for only *anonymous* pages created through `malloc` calls; as such, a number of file-backed pages created from are not superpage candidates. Overall, we find that 15% of non-superpage pages actually have over 512-page contiguity.

Fortunately, Figures 7, 8, and 9 enjoy particularly high contiguity for TLB-stressing benchmarks. `Mcf`, `Tigr`, and `CactusADM` see tens to hundreds of contiguous pages, indicating their amenability to TLB coalescing. For a number of these benchmarks, such as `Mcf`, high contiguity arises because `malloc` and `mmap` calls are made at the beginning of the execution to allocate large hash-based data structures. These structures span megabytes of space, which the buddy allocator ensures maps to contiguous physical pages.

6.2. No Superpaging, Memory Compaction

Figures 10 to 12 show how contiguity changes when superpaging support is disabled. Average contiguity drops compared to THS on from 41 to 18, for two reasons. First, THS optimistically creates as many 2MB page as possible. While these 2MB pages eventually get broken into 4KB pages due to system load, they do leave large amounts of smaller, residual contiguity. Without THS, contiguity is not generated this way. Second, disabling THS drastically reduces memory compaction daemon invocations. Nevertheless, sufficient exploitable intermediate contiguity remains (in the tens of pages, around 18). Furthermore, heavy TLB-pressure benchmarks like `Mcf` and `Mummer` see very high contiguity.

Surprisingly, some benchmarks like `Omnetpp` and `Sjeng` actually see *higher* contiguity without THS. This occurs because the lack of THS reduces superpages allocated to *other* running processes. As a result, the pages allocated to our workloads remain unfragmented and contiguous.

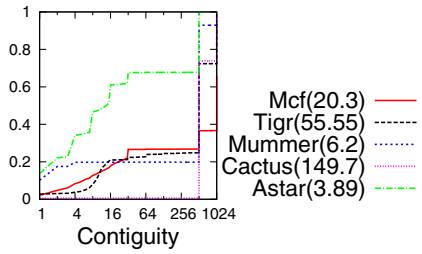

Figure 7: THS on, normal memory compaction contiguity CDF.

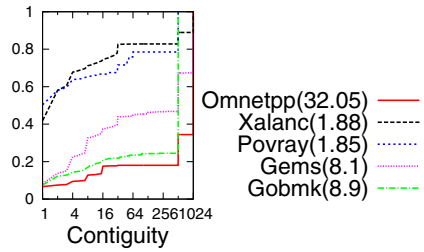

Figure 8: THS on, normal memory compaction contiguity CDF.

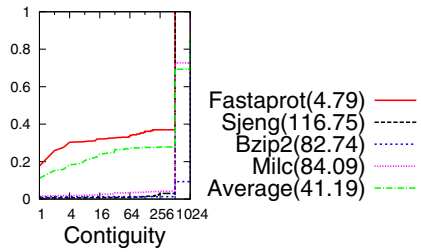

Figure 9: THS on, normal memory compaction contiguity CDF.

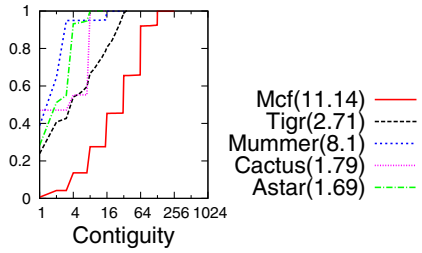

Figure 10: THS off, normal memory compaction contiguity CDF.

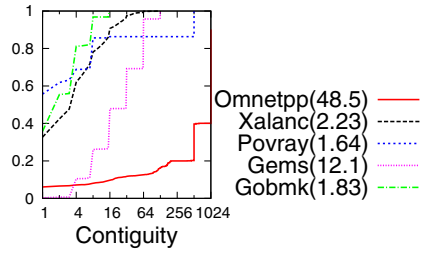

Figure 11: THS off, normal memory compaction contiguity CDF.

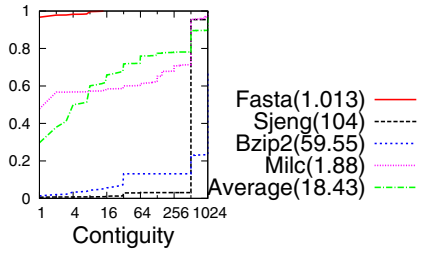

Figure 12: THS off, normal memory compaction contiguity CDF.

6.3. Superpaging, Low Memory Compaction

Figures 13, 14, and 15 present a worst-case scenario setting for Linux, where THS is turned off and memory compaction is greatly reduced via disabling the `defrag` kernel flag. While *no* kernel uses or recommends this setting, we study it to ensure that sufficient contiguity exists, even when there are almost no mechanisms to explicitly generate it. In fact, our results show that on average, contiguity drops only marginally compared to the THS off, normal memory compaction case to 15 pages on average. While important benchmarks like `Mcf`, `Mummer`, and `Omnetpp` do lose compared to the prior settings, they retain sufficiently high intermediate contiguity. For example, even though `Mummer's` average contiguity is now 1.3, roughly 50% of its 4KB pages enjoy 4-page contiguity. Correctly exploiting this gives our TLBs a 4× reach.

6.4. Superpaging, Memory Compaction, Memhog

We now focus on the impact of system load on fragmentation and contiguity. Figure 16 shows how contiguity is affected when `memhog` runs with each benchmark and fragments 25% and 50% of system memory. Our studies have shown that combined with the other running system processes, `memhog` with 50% heavily fragments almost all memory and causes page fault rates to greatly increase. We assume default Linux settings (THS enabled, normal memory compaction).

One might initially expect higher load to lower contiguity. Surprisingly, we find the *opposite* trend to hold when using `memhog(25%)`, with contiguity rising from 41 to roughly 43 pages on average. For some benchmarks, the gain is markedly high; for example, `Mcf` and `GemsFDTFD` contiguities are boosted by an order of magnitude. The primary reason for this is that higher load invokes the memory compaction daemon more often. This in turn provides the buddy allocator more contiguous physical blocks.

Greatly fragmenting the system with `memhog(50%)` however, does reduce contiguity. However, even this intermediate contiguity is relatively high, averaging close to 10 pages. For heavy TLB-pressure benchmarks like `Mcf` and `Mummer`, this configuration still achieves higher contiguity than without system load. As such, the buddy allocator, in tandem with memory compaction, manages to actually leverage the additional load to increase contiguity.

6.5. No Superpaging, Memory Compaction, Memhog

This represents the scenario where THS is turned off despite high system load. While kernel settings would not typically allow this, we use this setting to stress-test our measurements. We find that even under the pessimistic setting of no THS and `memhog(50%)`, the average contiguity is above 5. TLB coalescing can thus potentially provide a 5× reach.

6.6. Summary of Results

Three primary conclusions can be drawn from our real-system characterizations. First, under *every* single configuration, even those that are unrealistically severe, the buddy allocator, compaction daemon, and THS support succeed in inadvertently generating great intermediate contiguity. Second, system load can have surprising implications on contiguity, often increasing it. For some benchmarks that suffer from high TLB misses, such as `Mcf`, this is a promising observation. Third, superpages are ill-equipped to handle this contiguity. Therefore, coalescing techniques to harness this intermediate contiguity are warranted.

7. CoLT Evaluations

We now evaluate CoLT's benefits, focusing on per-application miss rate reductions and performance gains.

7.1. TLB Miss Rate Analysis

7.1.1. CoLT TLB miss rates. Figure 18 quantifies CoLT's TLB miss reductions. Benchmarks are ordered from highest to lowest TLB miss rates. We first capture the number of L1 and L2 TLB misses for a baseline configuration with 32-entry and 128-entry L1 and L2 TLBs (4-way) and a 16-entry superpage TLB. Note that we count misses for both the set-associative L1 TLB and the superpage TLB as L1 TLB misses since they are checked in parallel and have

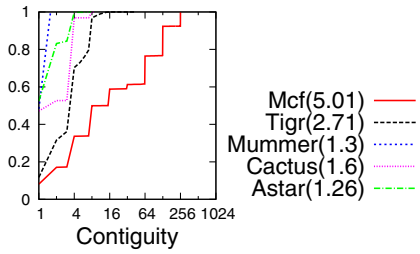

Figure 13: THS off, low memory compaction contiguity CDF.

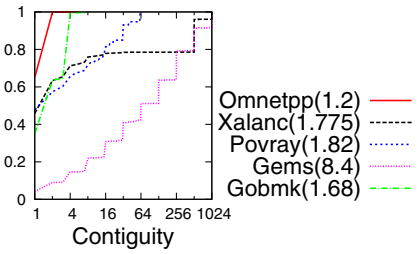

Figure 14: THS off, low memory compaction contiguity CDF.

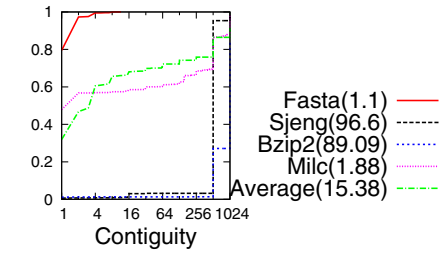

Figure 15: THS off, low memory compaction contiguity CDF.

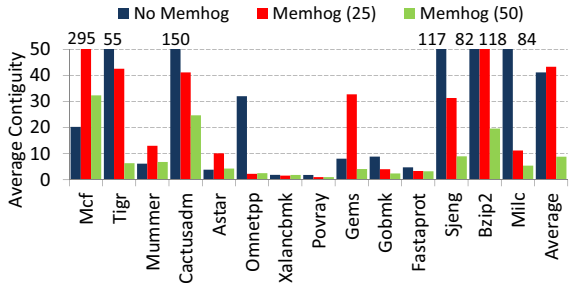

Figure 16: Average contiguity for THS on, normal memory compaction with varying `Memhog`.

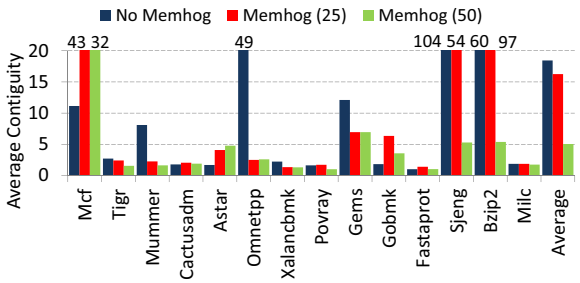

Figure 17: Average contiguity for THS off, normal memory compaction with varying `Memhog`.

the same hit time. After recording these misses, we then run the same benchmarks on configurations with CoLT-SA, CoLT-FA, and CoLT-All, tracking the new TLB miss rates. We assume that CoLT-SA uses VPN[4-2] and VPN[6-2] for L1 and L2 set selection, meaning that up to four translations can be coalesced per entry (we will later show the effect of using more aggressive indexing). We also conservatively assume 8-entry fully-associative TLBs when using CoLT-FA and CoLT-All.

First and foremost, Figure 18 shows that all three CoLT schemes improve *every* single benchmark by eliminating large chunks of the baseline misses. On average, CoLT-SA eliminates 40% of both L1 and L2 TLBs misses, while CoLT-FA and CoLT-All do even better, eliminating around 55% of both L1 and L2 misses. Second, Figure 18 shows that many of the benchmarks experiencing TLB pressure gain particularly from CoLT. For example, `Mcf`, `CactusADM`, and `Astar` all eliminate vast amounts of their TLB misses. In fact, `Astar` almost achieves perfect TLBs with no misses with CoLT-FA and CoLT-All.

Third, there is a correlation between system contiguity and effectiveness of CoLT. For example, `Mcf`, `Bzip2`, `Milc`, and `CactusADM`, which all see more instances of 20-page contiguity on average, can coalesce large amounts of translations, increasing TLB reach substantially. However, contiguity alone does not guarantee coalescing success; for coalescing to be effective, contiguous entries must actually be used close together in time. Without this temporal proximity, a coalesced entry will be evicted from the TLB before multiple member translations are used. This explains the lower benefits of `Tigr`, which sees 10% TLB miss elimination rates despite a contiguity of over 50 pages on average.

Fourth, Figure 18 shows that leveraging the superpage TLB in CoLT-FA and CoLT-All provides 10-15% gains over CoLT-SA on average. Benchmarks like `Mcf` and `Fastaprot` benefit particularly from this. These gains are achieved *despite* dropping from a 16-entry to an 8-entry structure. We find that the primary reason for this is that even with THS on, superpages are used sparingly. Therefore, a surprisingly high number of entries remain wasted in the fully-associative TLB in the baseline case. Instead, CoLT-FA and CoLT-All use these entries and can even perform unrestricted coalescing on them, unlike the set-associative TLBs.

The difference between CoLT-FA and CoLT-All remains more nuanced. We find generally that both schemes eliminate roughly 55% of TLB misses on average. Generally on the more TLB-intensive benchmarks (eg. `Mcf`, `Tigr`, `Mummer`, `CactusADM`), CoLT-All outperforms CoLT-FA slightly. However, in many benchmarks, CoLT-All falls surprisingly short of CoLT-FA. This occurs because CoLT-FA is better able to coalesce translations that reside across multiple LLC cache lines. Essentially, in these benchmarks only a few translations in a single cache line are coalescible with translations from another cache line. In CoLT-FA, since all the translations are brought into the fully-associative structure, these entries are coalesced. In CoLT-All however, if the cache line that maintains only a few coalescible translations falls below the pre-defined threshold, they are inserted into the set-associative TLB and can therefore never be merged with the second cache line's translations (which sits in the fully-associative TLB). This reduces CoLT-All's hit rates compared to CoLT-FA.

Overall, all CoLT designs eliminate a large fraction of TLB misses. We now focus on implementation details of the various CoLT designs to lend greater insight on our gains.

7.1.2. Impact of CoLT-SA's indexing scheme on TLB miss rates. Our initial CoLT-SA results assume that we use VPN[4-2] and VPN[6-2] for L1 and L2 set selection. This limits the amount of coalescing to four translations per entry. While additional contiguity could be coalesced by further left-shifting the index bits, this also increases conflict misses. Figure 19 studies these opposing forces on the 4-way associative TLBs by left-shifting the traditional index bits by one bit (VPN[3-1] and VPN[5-1] for L1 and L2 TLBs), two bits, and three bits (VPN[5-3] and VPN[7-3] for L1 and L2 TLBs). These correspond to maximum allowable coalescing of two, four and eight translations.

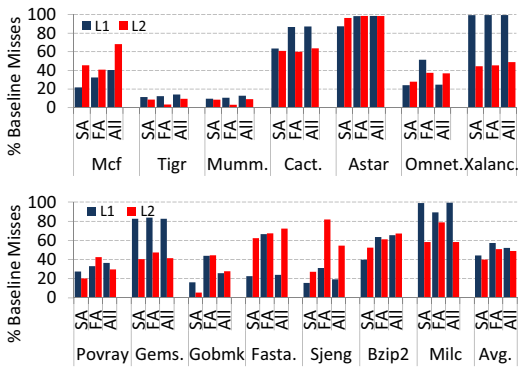

Figure 18: Percentage of baseline TLB misses eliminated using CoLT-SA, CoLT-FA, and CoLT-All.

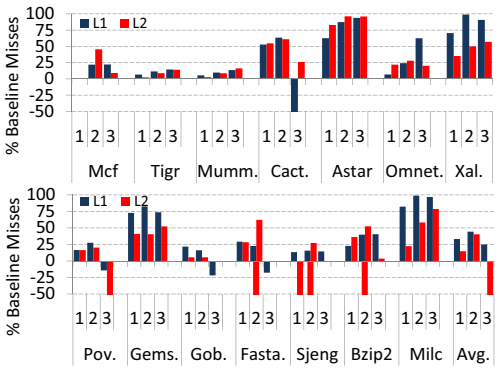

Figure 19: Percentage of baseline TLB misses eliminated by CoLT-SA when left-shifting the index by 1, 2, and 3 bits.

Figure 19 clearly shows that left-shifting the index bits by two provides the best balance between coalescing opportunity and conflict misses. Below this (left-shifting by one bit), we can only coalesce two entries, restricting our TLB miss elimination rates. However, left-shifting by three bits actually *increases* TLB misses in many cases due to the additional conflict misses. In general, unless there is very high contiguity, like for `Mummer`, `Tigr`, and `Milc`, left-shifting the index bits by three is overly-aggressive. Henceforth, we assume a left-shit of two bits for our indexing scheme.

7.1.3. Impact of bringing missing entries into L2 TLB for CoLT-FA and CoLT-All. As previously detailed, while CoLT-FA and CoLT-All bring coalesced entries into the fully-associative, superpage TLB, they also leverage the L2 TLB. For CoLT-FA, when a coalesced entry is brought into the superpage TLB, just the requested entry is also brought into the L2 TLB; for CoLT-All, a coalesced entry (where the coalescing amount is restricted by the index scheme) is brought into the L2 TLB. As we previously noted, this is useful since the superpage TLB is small (8-entry); as a result, only entries with high levels of coalescing are maintained there. As such, intermediate-level coalesced entries are often evicted. Bringing these entries into the L2 TLB as well increases the chance that these they remain available if necessary.

For CoLT-FA, we have run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and just the translation triggering the coalescing is also brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We have found that on average, (1) outperforms (2) by an additional miss elimination of 10-15% for both L1 and L2 miss counts. We see particularly high gains with our approach on workloads with relatively lower contiguity such as `Povray`, since the small superpage TLB cannot coalesce a high enough number of entries to prevent eviction.

For CoLT-All, we have similarly run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and its smaller coalesced version (a maximum of coalescing of four translations in our design) is brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We see again that our approach, (1) outperforms (2) by an average of 10-20% TLB miss eliminations.

7.1.4. Studying CoLT's effectiveness at higher associativities. We now consider CoLT effectiveness as TLB associativity is varied. A number of past studies have quantified how effectively increasing TLB associativity eliminates misses [13]. Generally, these studies have concluded that the slightly higher TLB hit rates are offset by huge power dissipation problems [7]. These observations are largely responsible for the relatively low associativity (typically 2-way or 4-way) supported on current TLBs.

CoLT, however, increases the benefits of higher set-associativity since the indexing scheme of the TLB can be more aggressively changed without as significant an increase in conflict misses. Overall, this allows higher levels of coalescing. Figure 20 compares how many L2 TLB misses in a 4-way 128-entry L2 TLB can be eliminated by CoLT-SA (4-way, CoLT-SA), by varying the associativity to 8-way but not allowing coalescing (8-way, No CoLT), and by allowing CoLT on the 8-way TLB (8-way, CoLT-SA). Note that all configurations use a fixed TLB size despite associativity changes.

First, Figure 20 shows that merely increasing the associativity to 8-way only eliminates 10% of the baseline L2 misses. In fact, even 4-way L2 TLBs with low-overhead CoLT-SA far exceed the benefits of higher associativity, eliminating 40% of baseline misses on average.

Figure 20 shows however, that the 8-way configuration augmented with CoLT-SA does provide significant benefits. CoLT-SA now eliminates 60% of the baseline misses on average, a substantial improvement over the other two scenarios. While a detailed power analysis is beyond the scope of this work, the performance, power ratio may therefore become more amenable with CoLT.

7.2. Performance Analysis

Up to this point, we have evaluated the benefits of CoLT in terms of miss rate eliminations. While this does indicate CoLT's effectiveness, we now focus on performance numbers which track how much faster each application runs with coalescing. Figure 21 details, for every benchmark, performance improvements from CoLT-SA, CoLT-FA, and CoLT-All. It also provides data on performance improvements that would occur with absolutely perfect, 100%-hit rate TLBs. The latter serves as a comparison point to determine how effectively CoLT performs. Once again, the baseline is a system with 4-way 32-entry and 128-entry L1 and L2 TLBs, and a 16-entry superpage TLB. CoLT-FA and CoLT-All conservatively reduce the superpage TLBs to 8 entries. Moreover, as previously detailed, we simulate a 4-way out-of-order processor.

Figure 21 shows that perfect TLBs would improve most benchmark runtimes by over 10% (eg. all except `Gobmk` and `Sjeng` in our benchmarks). In fact, `Xalancbmk` sees a huge 115% improvement in performance from TLBs that achieve 100% hit rate. These numbers indicate that TLB miss handling does significantly slow down bench-

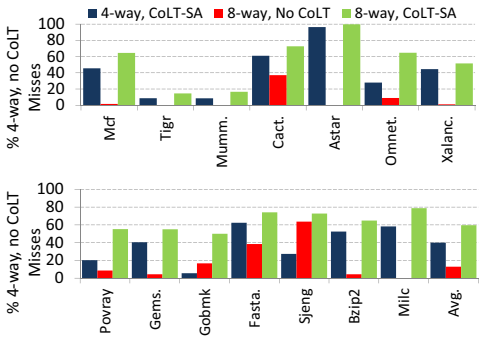

Figure 20: Percentage of baseline misses eliminated by CoLT-SA when increasing associativity.

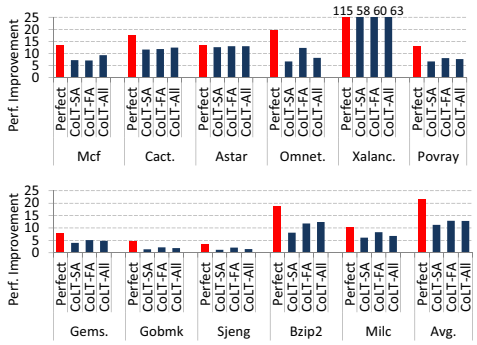

Figure 21: CoLT-SA, CoLT-FA, and CoLT-All performance improvements compared to perfect TLBs.

marks. This also implies that CoLT strategies have the potential to significantly improve performance.

Fortunately, Figure 21 shows that all of the CoLT approaches do indeed boost application performance significantly. On average, CoLT-SA achieves a 12% performance improvement, while CoLT-FA and CoLT-All achieve 14% improvements. On benchmarks like `Xalancbmk`, the performance improvements hover around 60% of runtime. Across other workloads like `Mcf`, `CactusADM`, `Astar`, `Omnetpp`, and `Bzip2`, at least one of the CoLT configurations improves performance over 10%. We anticipate that as applications with even larger working sets or virtualization are considered, these performance improvements will be even higher.

Figure 21 indicates that CoLT-SA, which has the simplest implementation, performs comparably to CoLT-FA and CoLT-All. Nevertheless, benchmarks like `Omnetpp` and `Bzip2` do see boosts from CoLT-FA/CoLT-All. Because we assume smaller 8-entry fully-associative TLBs, we expect CoLT-FA and CoLT-All results to be even higher with more realistically-sized superpage TLBs.

8. Conclusion

This paper proposes and designs Coalesced Large-Reach TLBs capable of exploiting address translation contiguity to achieve high reach. Due to a variety of OS memory management techniques involving buddy allocators, memory compaction, and superpaging, large amounts of translation contiguity are generated, even under heavy system load. While this contiguity typically cannot be exploited to generate superpages, CoLT provides lightweight hardware support to detect this behavior. As a result, large TLB miss eliminations are possible (on average, 40% to 58%), translating to performance improvements of 14% on average.

This work has a number of interesting implications for architects, system designers, and OS designers. We showcase TLB optimization techniques that architects can readily incorporate in existing processors. System designers and OS designers can tune their software systems to generate contiguity suitable for CoLT. For example, while superpages can overwhelm systems performance due to their overheads, CoLT provides alternate mechanisms to boost performance. We believe that CoLT will become even more critical as applications have increasingly large memory requirements and trends like virtualization become prevalent.

9. Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank William Katsak for his help in modifying the Linux kernel for our studies. Finally, we thank Viji Srinivasan for her suggestions on improving the final version of the paper. The authors acknowledge the support of Rutgers University's Office of the Vice President for Research and Economic Development. This work was supported in part by their Faculty Research Grant award.

References

[1] "The Standard Performance Evaluation Corporation. SPEC CPU2006 Results," *http://www.spec.org/cpu2006*.
[2] K. Albayraktaroglu *et al.*, "BioBench: A Benchmark Suite of Bioinformatics Applications," *ISPASS*, 2005.
[3] AMD Corporation, "AMD Programmer's Manual," vol. 2, 2007.
[4] Andrea Arcangeli, "Transparent Hugepage Support," *KVM Forum*, 2010.
[5] T. Barr, A. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *ISCA*, 2010.
[6] T. Barr, A. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," *ISCA*, 2011.
[7] A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *ISCA*, 2012.
[8] R. Bhargava *et al.*, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *ASPLOS*, 2008.
[9] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2010.
[10] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," *PACT*, 2009.
[11] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," *ASPLOS*, 2010.
[12] D. Bovet and M. Cesati, "Understanding the Linux Kernel," 2005.
[13] J. B. Chen, A. Borg, and N. Jouppi, "A Simulation Based Study of TLB Performance," *ISCA*, 1992.
[14] 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.
[15] Z. Fang *et al.*, "Online Superpage Promotion with Hardware Support," *HPCA*, 2001.
[16] N. Ganapathy and C. Schimmel, "General-Purpose Operating System Support for Multiple Page Sizes," *USENIX*, 1998.
[17] Intel Corporation, "TLBs, Paging-Structure Caches and their Invalidation," *Intel Technical Report*, 2008.
[18] A. Jaleel *et al.*, "CMP$im: A Pin-based On-the-fly Multi-core Simulator," *4th Workshop on Modeling, Benchmarking, and Simulation*, 2008.
[19] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," *ISCA*, 2002.
[20] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *PLDI*, 2005.
[21] C. McCurdy, A. Cox, and J. Vetter, "Investigating the TLB Behavior of High-End Scientific Appplications on Commodity Multiprocessors," *ISPASS*, 2008.
[22] D. Nagle *et al.*, "Design Tradeoffs for Software-Managed TLBs," *ISCA*, 1993.

[23] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.
[24] E. Perelman *et al.*, "Using SimPoint for Accurate and Efficient Simulation," *SIGMETRICS*, 2003.
[25] T. Romer *et al.*, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," *ISCA*, 1995.
[26] M. Rosenblum *et al.*, "The Impact of Architectural Trends on Operating System Performance," *SOSP*, 1995.
[27] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," *ISCA*, 2000.
[28] M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *ASPLOS*, 1994.
[29] M. Talluri *et al.*, "Tradeoffs in Supporting Two Page Sizes," *ISCA*, 1992.
[30] Virtutech, "Simics for Multicore Software," 2007.