Increasing TLB Reach by Exploiting Clustering in Page Translations

Binh Pham Abhishek Bhattacharjee Yasuko Eckert Gabriel H. Loh

Department of Computer Science
Rutgers University
{binhpham, abhib}@cs.rutgers.edu

AMD Research
Advanced Micro Devices, Inc.
{yasuko.eckert, gabriel.loh}@amd.com

Abstract

The steadily increasing sizes of main memory capacities require corresponding increases in the processor's translation lookaside buffer (TLB) resources to avoid performance bottlenecks. Large operating system page sizes can mitigate the bottleneck with a smaller TLB, but most OSs and applications do not fully utilize the large-page support in current hardware. Recent work has shown that, while not guaranteed, some virtual-to-physical page mappings exhibit "contiguous" spatial locality in which consecutive virtual pages map to consecutive physical pages. Such locality provides opportunities to coalesce "adjacent" TLB entries for increased reach. We observe that beyond simple adjacent-entry coalescing, many more translations exhibit "clustered" spatial locality in which a group or cluster of nearby virtual pages map to a similarly clustered set of physical pages. In this work, we provide a detailed characterization of the spatial locality among the virtual-to-physical translations. Based on this characterization, we present a multi-granular TLB organization that significantly increases its effective reach and reduces miss rates substantially while requiring no additional OS support. Our evaluation shows that the multi-granular design outperforms conventional TLBs and the recently proposed coalesced TLBs technique.

1. Introduction

As processor vendors embrace the era of big data, fields like scientific computing, data mining, social networks, and business management depend on processing massive, multi-dimensional data sets. Hardware designers have responded by proposing hardware appropriate for workloads requiring large data reach. These architectures demonstrate a rapid increase in on-chip hardware caches and main memory so that workloads can quickly access large working sets.

In this context, it is critical to re-evaluate virtual memory, ubiquitous across computer systems today. Virtual memory is a powerful abstraction crucial to programmer productivity that automates data transfer between main memory and secondary storage, provides protection benefits, and enables software modularity. At its core, programs operate on virtual addresses that are translated to system-level physical addresses on data requests. Processor vendors accelerate address translation using hardware translation lookaside buffers (TLBs) to cache recently-used virtual-to-physical address translations or page table entries (PTEs). Because misses in these structures are expensive (e.g., x86 systems require the traversal of a four-level page table to find the desired PTE), past work has shown that TLB microarchitecture significantly influences system performance [2, 3, 5–7, 12].

Unfortunately, as software demands more memory, TLB misses impose a bottleneck on memory accesses. Because TLBs struggle to map out the reach of increasing memory and cache sizes, this bottleneck is becoming a critical problem. In response, studies have considered optimizations like better TLB organizations [6, 8], prefetching [7, 11, 14], and speculation [2].

The most recent work has proposed coalesced large-reach TLBs (CoLT) [12] based on the observation that, independent of large pages, operating systems typically exhibit "contiguous" spatial locality (although this is not guaranteed) in which tens of consecutive virtual pages are mapped to consecutive physical pages. This behavior, caused in part by OS buddy allocators and memory compaction [12], generates many instances of contiguous PTE spatial locality, though typically not enough for large page generation. CoLT proposes novel but modest hardware changes to a conventional TLB to exploit contiguous spatial locality.

Our work goes beyond CoLT by observing that many translations exhibit "clustered" spatial locality in which translations are "nearby" in the same address region. We then introduce a multi-granular TLB organization that exploits clustered spatial locality.

This work makes the following contributions. First, we provide a detailed characterization of PTE spatial locality across many workloads. We use both detailed simulations and real-system approaches and show that weakly-clustered spatial locality is more prevalent than contiguous spatial locality. Second, we propose a low-overhead, multi-granular TLB organization that exploits PTE clustering. Our approach uses modest hardware and no OS support, making it robust for applications ranging from the server and desktop to high-performance computing and cloud-computing domains. We consider enhancements to our design (e.g., replacement policies and prefetching) that eliminate 46% of L2 TLB misses on average. Third, our approach largely subsumes the prior CoLT technique by exploiting contiguous spatial locality when it exists. Our approach is thus effective even when OSs have been running for long periods and are fragmented, making contiguous spatial locality scarce.

2. Related Work and Our Approach

2.1. Address Translation Overheads and Enhancements

Despite its programmability benefits, address translation typically degrades performance by 5-15% [2, 6, 7, 12]. Emerging software trends like big data and virtualization further increase these overheads to as much as 40-50% [3, 5]. In response, past studies have considered optimizations such as novel TLB organizations [6, 8], prefetching [7, 11], synergistic TLBs [15], speculation [2], and direct segments [3].

A particularly compelling approach is to encourage the OS to allocate adjacent virtual pages to adjacent physical pages. It then is possible to propose hardware that stores groups of adjacent PTEs in a single TLB entry using a large page to reduce miss rates and increase performance. Modern OSs generate large pages [1, 16] by mapping hundreds of consecutive physical pages to consecutive virtual pages (e.g., x86 requires 512 adjacent PTEs for baseline 4KB pages to realize a 2MB large page). While effective in many cases, large pages must be used carefully because they can suffer overheads from specialized OS code and increased paging traffic [16]. Hence, in practice, large pages are used sparingly (if at all).

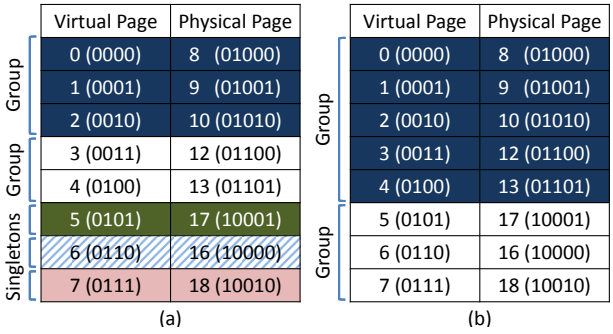

Figure 1. The figure on the left shows the presence of contiguous spatial locality (sequential groups) in a page table. The figure on the right shows that if clustered locality is also observed, the entire page table can be more efficiently covered.

2.2. Spatial Locality in Page Table Entries

Large pages exploit cases in which large swathes of contiguous virtual pages are assigned contiguous physical pages. We refer to groups of adjacent PTEs as *contiguously spatially local*. Large pages require explicit OS intervention to ensure ample contiguous spatial locality; however, it is also possible for operating systems to generate intermediate amounts of spatial locality (in the range of tens to a few hundreds of PTEs). For example, Figure 1(a) shows a page table in which the PTEs for virtual page numbers (VPNs) 0-2 are in a sequential group of physical pages. Similarly, the sequential group of PTEs for VPNs 3-4 are contiguously spatially local. Past work shows that this behavior occurs often even in the absence of large page support because of OS buddy allocators and memory-compaction daemons [12]. Overall, we find that our page table is made up of two sequential groups of PTEs exhibiting contiguous spatial locality and three additional "singleton" PTEs.

This work's key insight is that there exists another form of spatial locality, likely occurring in greater abundance than contiguous spatial locality. Specifically, we find that many PTEs exhibit *clustered spatial locality* in which a cluster of nearby virtual pages map to a similarly clustered set of physical pages. Consider Figure 1(b), assuming that we scan for clusters of up to eight PTEs. In our example, PTEs demonstrate clustered spatial locality if they all share the same VPN divided by 8 *and* the same physical page number (PPN) divided by 8 (i.e., we ignore the bottom 3 VPN and PPN bits). Therefore, the entire page table is covered by two clusters. The first cluster matches the first two sequential group of PTEs from Figure 1(a), and the second cluster comprises PTEs for VPNs 5-7. The goal of our work is to show that this form of clustered locality is abundant, even in fragmented systems, and to design low-overhead hardware to exploit these patterns.

2.3. Past Techniques to Exploit Page Table Spatial Locality

The following three techniques have been proposed in past studies to exploit PTE spatial locality.

Coalesced Large-reach TLBs (CoLT): Past work [12] proposed CoLT to exploit contiguous spatial locality. Figures 2(a) and (b) contrast the structure of a conventional TLB entry with a CoLT entry. While a conventional TLB entry corresponds to a single PTE (in our example, virtual page V1 and physical page P3), a CoLT entry maps a group of contiguous, spatially-local PTEs (in our example, PTEs for virtual pages 1-5). Any arbitrary set of PTEs can be accommodated (e.g., five PTEs in Figure 2) by recording only the base PTE and the number of coalesced PTEs. On look-up, the offset between the base virtual address stored in the tag is used to calculate the offset from the base physical page. There are no alignment restrictions for this approach. CoLT achieves high reach but is entirely reliant on contiguous spatial locality. Unfortunately, contiguous spatial locality becomes rare as systems are fragmented and run for longer periods [12].

Complete sub-blocking: This approach relaxes the need for contiguous spatial locality [16]. Instead, complete sub-blocking looks for clusters of PTEs with contiguous VPNs. For a given sub-block factor N, this approach looks for B *aligned* virtual pages (i.e., all virtual address bits apart from the bottom $\log_2(B)$ bits are the same). It then places all the PTEs corresponding to this group in one complete entry. Figure 2(c) shows an example of this where virtual pages 0-3 all are aligned for a sub-block factor of 4. This means that their PTEs can be stored in one entry if it maintains a field for each PPN (e.g., P1, P6, P3, and P5). Unfortunately, the ability of complete sub-blocking to store any set of PPNs requires expensive hardware (multiple PPN fields). Furthermore, unlike CoLT which accommodates any length of contiguous PTEs, complete sub-blocking stores a PTE count equal to the sub-block factor.

Partial sub-blocking: Talluri and Hill proposed partial sub-blocking as a lower-overhead alternative to complete sub-blocking [16]. Figure 2(d) shows that partial sub-blocking searches for PTEs with an aligned group of virtual pages *and* an aligned group of physical pages. All PTEs that have VPNs and PPNs with the same offset from the start of the aligned package are coalescable into a single entry. In our example, PTEs for VPNs 0, 2, and 3 achieve this. This approach permits "holes" in a group of PTEs when the physical page offset within the aligned packet is different from the virtual page offset (e.g., the PTE for virtual page 1 in our example). Partial sub-blocking achieves high reach using much simpler hardware than complete sub-blocking by imposing alignment and offset restrictions on PPNs. Figure 2(d) shows each entry maintains only a bit vector recording the presence of the physical pages rather than the entire PPN.

Intuitively, partial sub-blocking goes beyond CoLT by exploiting contiguous spatial locality *and* limited forms of clustered locality. However, its PPN alignment and offset requirements cannot capture many instances of clustered spatial locality (e.g., the third cluster in Figure 1 cannot be leveraged because it requires VPNs 5 and 6 to map to PPNs 16 and 17, respectively, to be useful). In practice, we find that most instances of clustered spatial locality in PTEs do not fit the alignment requirements of partial sub-blocking (our measurements show that less than 10% of PTEs fit these alignment requirements naturally). While the original partial sub-blocking approach [16] addresses this problem by adding specialized OS code to generate the right alignment and offset features, our goal is to avoid explicit OS modifications.

2.4. Our Approach: Clustered TLBs

We design a multi-granular TLB architecture that exploits more general forms of spatial locality. We focus on clustered PTE spatial locality that also largely subsumes contiguous spatial locality. We achieve this using a novel clustered TLB architecture.

Figure 2(e) shows a clustered TLB. Like sub-blocked TLBs, a clustered TLB is designed for a maximum cluster factor of N (in this example, N=4). Suppose an aligned group of virtual pages is found; if these virtual pages also point to an aligned group of physical pages, the PTEs can be placed in a single clustered TLB entry. This means that: (1) all VPNs in a cluster share the same bits ignoring the bottom $\log_2(N)$ bits; and (2) all PPNs in a cluster share the same bits ignoring the bottom $\log_2(N)$ bits. Unlike partial sub-

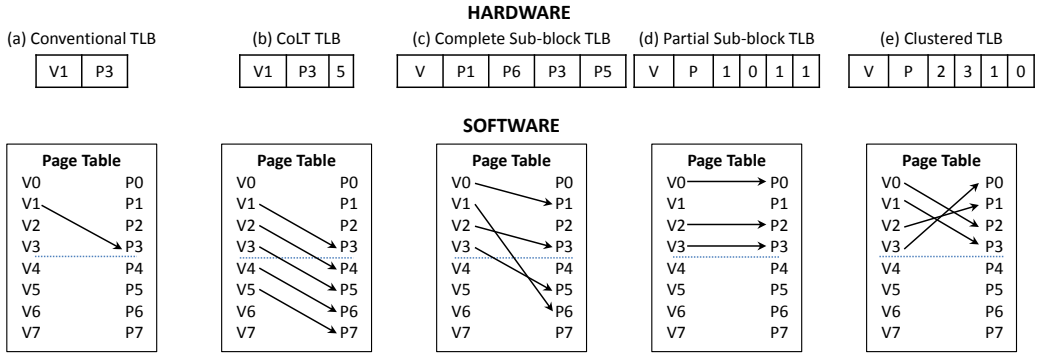

Figure 2. Operation of CoLT, complete sub-blocking, and partial sub-blocking versus clustered TLB. For each approach, we show the structure of a single entry and a page table with the PTEs that can be exploited.

blocking, these PPNs have no additional offset requirements in a cluster (in our example, V0 maps to P2, but V2 to P1) and permit holes. An N-wide vector is maintained per entry to record the offset of the physical pages in the aligned group (only the $\log_2(N)$ bottom-order bits are needed for this). In this way, clustered TLBs can capture *all* the clusters from Figure 1.

Our approach offers key advantages relative to past work. Unlike CoLT, it captures clustered spatial locality, making it robust even in the presence of fragmentation. Second, unlike sub-blocking, it uses much simpler hardware and does not require explicit OS support. Subsequent sections detail our design. We characterize the presence of clustered locality across a variety of workloads and system configurations. We then show how a multi-granular design made up of a clustered TLB coupled with a small conventional TLB effectively captures this clustered spatial locality.

3. Weak Spatial Locality in Page Tables

In this section, we characterize spatial locality in page tables. We analyzed a total of eleven benchmark traces from SPEC workloads (xalancbmk, SPECweb, GemsFDTD, astar, omnetpp, mcf), server workloads (TPC-C, Trade6), and CloudSuite workloads [9] (Graph Analytics, Data Serving, Data Caching)[1]. These traces were correlated with hardware performance counters to ensure that they exhibit similar behaviors. We first characterize the opportunities for previously-proposed CoLT-like approaches, and then we relax the constraints on how PTE entries may be coalesced and examine the impact that has on the potential for coalescing.

3.1. CoLT-like Contiguous Spatial Locality

We measured the fraction of PTEs that exhibit CoLT-styled contiguous spatial locality, in which contiguous virtual pages map to contiguous physical pages. Each graph in Figure 3 corresponds to one application. The x-axis shows the number of PTEs that can be grouped, the y-axis is percentage of all PTEs, and the graphs show cumulative distributions. For example, the bold solid line (labeled "Contiguous" in the legend) for omnetpp shows that 46% of all PTEs have a grouping size of one (i.e., they cannot be coalesced with any adjacent PTEs), about 96% of translations can be coalesced into groups of six or fewer consecutive PTEs, and all PTEs can be coalesced into groups consisting of no more than eight consecutive PTEs. Across the benchmarks, there are some cases in which CoLT-styled sequentially allocated translations provide decent coalescing opportunities (e.g., GemsFDTD, mcf), and others in which sequentially consecutive translations are less prevalent (e.g., xalancbmk, SPECweb-B[2], Data Serving).

[1] See Section 5 for methodology details.

[2] 'B' corresponds to the SPECweb banking workload.

3.2. Clustered Spatial Locality

CoLT requires that consecutive virtual pages map to sequential physical pages. Requiring complete sequentiality for both VPNs and PPNs restricts coalescing opportunities. We instead consider the notion of clustered spatial locality: as long as nearby virtual pages map to nearby physical pages, we consider the corresponding PTEs coalescable. In Figure 3, the notation "Cluster$X$" indicates that translations from within an aligned cluster of 2^X virtual pages that map to an aligned cluster of 2^X physical pages potentially can be combined.

For example, the curves labeled Cluster3 limit the clustering or grouping of PTEs to those that fall within the same aligned set of eight (i.e., 2^3) pages. With the CoLT-style contiguous curves, a value of (for example) 3 on the x-axis indicates that three consecutive PTE entries mapped to three consecutive physical pages. With the Cluster3 curves, the same value of three indicates that three virtual pages (from within a group of eight), map to pages within an aligned group of eight physical pages. The three VPNs need not be consecutive, and the corresponding three PPNs also do not need to be consecutive or even in increasing address order.

For each curve, the point at which the curve meets the y-axis (i.e., the y-intercept) indicates the percentage of PTEs that cannot be coalesced with any other PTEs. For most benchmarks, a modest clustering scope of Cluster2 or Cluster3 can uncover more opportunities for PTE coalescing than when using the more constrained CoLT-like contiguous criteria. For example, in the benchmark xalancbmk, a CoLT-like approach leaves 77% of translations uncoalesced, whereas when considering groups of four (Cluster2) or eight (Cluster3) pages without the sequentiality constraint, the percentage of uncoalescable PTEs drops to 66% and 45%, respectively. Furthermore, the exact curves are heavily dependent on benchmark behavior. For example, Data Caching sees particularly large amounts of contiguous spatial locality (and even clustered spatial locality) because it uses memcached, which in turn allocates large data structures using the slab allocator, which targets contiguous memory allocation.

In almost all cases, relaxed clustering allows significantly more coalescing opportunities than a CoLT-based approach. As the clustering scope increases (i.e., larger X for Cluster$X$), the opportunities for coalescing increase as well (curves move further down and to the right), but in many cases Cluster3 or Cluster4 are sufficient for capturing a significant portion of the opportunity. Mcf is a case when the CoLT-styled approach appears to provide significantly more coalescing opportunity; this arises because the Cluster$X$ criteria limits the coalescing scope to at most 2^X pages, whereas if CoLT gets lucky and there exists a very long run of adjacent translations, CoLT can coalesce all of these.

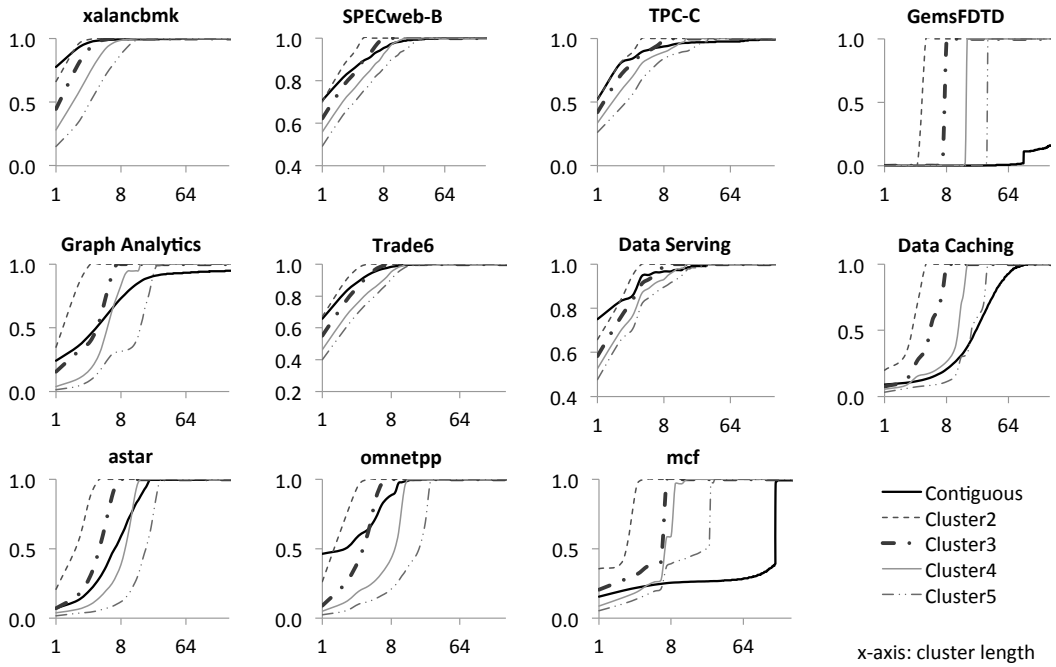

Figure 3. Cumulative distribution functions comparing the opportunity of CoLT-style contiguous spatial locality versus the clustered spatial locality that we target. In general, clustered spatial locality covers a bigger portion of the page table than contiguous spatial locality.

3.3. Impact of Memory System Fragmentation

A potential criticism and limitation of any TLB-coalescing approach is that after a system has been up and running for a long time, the virtual memory system may become fragmented. A highly-fragmented memory system would make it unlikely that nearby virtual pages get mapped to nearby physical pages (let alone to completely sequential physical pages). To quantify the impact of OS memory allocation fragmentation, we ran a subset of the benchmarks on a real server[3] and extracted live snapshots of the applications' page tables. This server is a highly-utilized machine mostly dedicated to running simulations, and the machine had an uptime of about 1.5 months at the time of these experiments. We found that while the exact amounts of coalescing opportunity are not the same as our trace-based analysis, they follow the same trends. We found that despite 1.5 month's worth of fragmentation, clustered spatial locality is more prevalent than contiguous spatial locality *in every single case*. For astar, omnetpp, and mcf, we found that eight clustered entries cover close to the full page table, whereas more than 64 such entries are required if only contiguous spatial locality is leveraged.

4. The Multi-granular TLB

The overall multi-granular TLB consists of a *clustered TLB* that can efficiently store multiple translations for PTEs with clustered spatial locality, a *conventional TLB* for singleton translations without spatial locality, *coalescing logic* for detecting clustered spatial locality and populating entries of the clustered TLB, and logic for performing look-ups, evictions, and other standard TLB operations.

4.1. Clustered TLB

Structure: The basic clustered TLB is a set-associative structure, much like a conventional TLB, but each TLB entry is designed to store multiple clustered page table translations. Figure 4(a) shows a clustered TLB entry. In this example, we assume a clustering reach of eight PTEs (i.e., Cluster3, or C3 for short). The eight VPNs all have identical values except for the lowest-three bits. The VPN's upper bits (i.e., the VPN's bits excluding the lowest three) are called the base VPN. Likewise, any of the coalescable PPNs are identical apart from their respective lowest-three bits, and the common prefix of the PPN is similarly called the base PPN. The key is that because all of the coalescable translations have identical base VPNs and base PPNs, each of these base values needs to be stored only once per clustered-TLB entry. Only the low-order bits of the PPN need to be tracked individually.

The C3 TLB entry potentially can track up to eight PTEs. For each of the individual potential translations, there is one *sub-entry*. Figure 4(a) also shows these eight sub-entries, with a detail of one such sub-entry's contents. Each sub-entry contains a valid bit, a dirty bit, a *referenced bit* (used in replacement policies described later), and the low-order bits of the PPN (e.g., the lowest-three bits in the case of Cluster3).

Look-up: To perform a look-up on the clustered TLB, we start with the VPN. Instead of using the entire VPN to generate a set index, we use only the base VPN (e.g., the lowest-three bits are omitted), as shown in Figure 4(b). If the requested base VPN matches the base VPN stored in the indexed clustered TLB entry[4], then we have a *cluster hit*, but this does not necessarily imply that the requested VPN is tracked by the clustered TLB. Next, we take the low-order bits of the VPN to select one of the eight sub-entries. If the selected sub-entry's valid bit is set, then this indicates an actual hit. The translated PPN then simply is reconstructed by concatenating the base PPN with the low-order PPN bits stored in the selected sub-entry. The sub-entry's referenced bit is set, and if the request corresponded to a write operation, then the sub-entry's dirty bit also is set. If we do not have a cluster hit, or if the selected sub-entry is

[3] 32-thread x86 multiprocessor with 64GB memory, running 64-bit Ubuntu OS v12.04.

[4] For simplicity, only a direct-mapped clustered TLB is shown in the figure, but extension to a set-associative organization parallels that for a conventional TLB or cache.

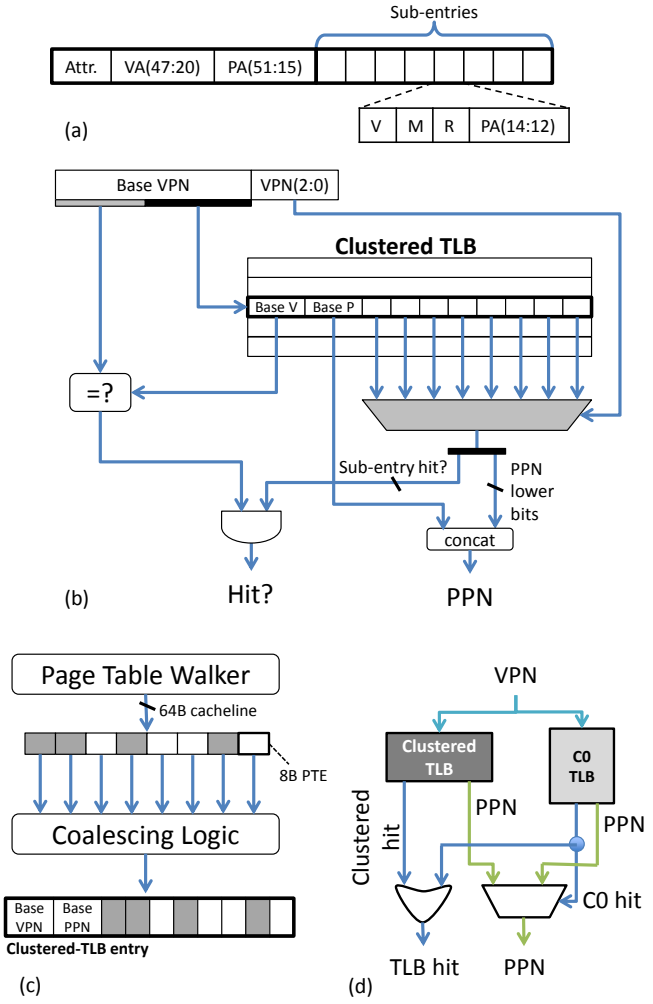

Figure 4. Our multi-granular L2 TLB uses a clustered TLB and a small conventional TLB. Coalescing is performed on TLB fill. (a) Clustered TLB entry, (b) Look-up, (c) Fill, (d) Multi-granular TLB.

not valid, then in either case this results in a cluster TLB miss.

Fill: On a clustered TLB miss, the TLB look-up is forwarded to the hardware page-table walker (PTW). In x86-64 processors, the PTW traverses the four-level page table and returns a cache line containing the requested PTE, as shown in Figure 4(c). In x86-64, the PTE entry size is 8 bytes, which means that a single 64-byte cache line contains a total of eight PTEs (i.e., the requested PTE plus seven others). The clustered TLB's coalescing logic examines the seven non-requested PTEs and checks to see which of these can be coalesced (shown shaded in Figure 4(c)) with the originally requested PTE (i.e., it detects how many PTEs exhibit clustered spatial locality). For the original PTE and each coalescable PTE, the corresponding sub-entry will have the valid bit set, the referenced bit cleared, and the low-order PPN bits stored. All other sub-entries have their valid bits cleared. The common base VPN and base PPN are stored in the overall clustered-TLB entry. All of this logic is off the critical path because the originally requested PTE can be returned as soon as the PTW's page table look-up has completed.

Clustered TLB Eviction: For a set-associative clustered TLB, a clustered TLB entry first must be evicted prior to installing a new set of clustered PTEs. Each clustered TLB entry may contain a different number of valid translations; simply relying on conventional replacement policies such as LRU fails to account for situations when the LRU entry contains many valid translations and other more recently used entries may contain only a few. It is not immediately clear how to trade optimizing for recency against the retention of a larger number of translations.

The sheer number of *valid* translations in a clustered TLB entry might not reflect the utility of those translations. The coalescing logic may prefetch up to seven additional translations (assuming Cluster3), but it is possible that none of these other translations are needed. We propose a simple replacement algorithm that incorporates the referenced bits from each of the sub-entries to estimate the overall *usefulness* of the clustered TLB entry. Usefulness is the number of valid sub-entries with their referenced bits set. We also define a *recency* value, which is the clustered TLB entry's position in the LRU recency stack (lower value = less recently used). Then for each clustered TLB entry in a set, we compute a retention priority:

$$priority = (\alpha * recency) + (\beta * usefulness)$$

The clustered TLB entry with the lowest priority is selected as the victim. This provides a balance between the recency of the clustered TLB entry, and the number of *useful* translations stored by the entry. We found that setting α and β to 1 and 2, respectively, provided good performance while maintaining very simple hardware (e.g., for a four-way set-associative clustered TLB, the recency value is only two bits wide, the usefulness value is four bits wide for Cluster3, and multiplication by 1 and 2 are trivial.

To avoid the pathological situation when a clustered TLB entry that has not been used recently, but still is kept around due to a large number of sub-entries that were useful (i.e., referenced) a long time ago, we periodically decay the referenced bits. We found that exactly how the referenced bits are cleared is not very important; we tried periodic and pseudo-random approaches across a very large range of decay intervals and found that overall performance is largely insensitive if *some* decay occurs every now and then.

4.2. Multi-granular TLB Organization and Operation

The clustered TLB provides efficient storage of multiple PTEs by *not* having to redundantly store multiple copies of the base VPN and base PPN for translations that exhibit clustered spatial locality. However, the locality characterization results from Section 3 showed that there remains a non-trivial percentage of PTEs that cannot be coalesced with other PTEs. Storing such singleton translations in a clustered-TLB entry would be wasteful because only a single sub-entry would be utilized. Figure 4(d) shows the high-level organization of the multi-granular TLB (MG-TLB). The MG-TLB consists of a clustered TLB paired with a conventionally-organized (i.e., not clustered) TLB. For shorthand, we label the conventional TLB "C0".[5] The conventional TLB is primarily utilized to cache singleton translations that cannot be clustered with other PTEs.

Look-up: To perform a look-up, both structures are searched in parallel. A hit in either indicates a TLB hit, and the translation is provided by the hitting structure. A miss in both structures results in an overall TLB miss, and the request is sent to the PTW to retrieve the translation from the page table.

Fill: In the MG-TLB, when the PTW returns the cache line with the requested PTE (along with the seven other neighboring PTEs), the entire set of eight PTEs is delivered to the clustered TLB's coalescing logic. At the end of the coalescing process, the MG-TLB first checks the coalescing degree (i.e., how many PTEs were successfully coalesced). If the coalescing degree is greater than a threshold value θ, then the entire set of coalesced translations is installed into

[5] A conventional TLB can be viewed as a degenerate case of the clustered TLB with a clustering scope of zero (i.e., Cluster0).

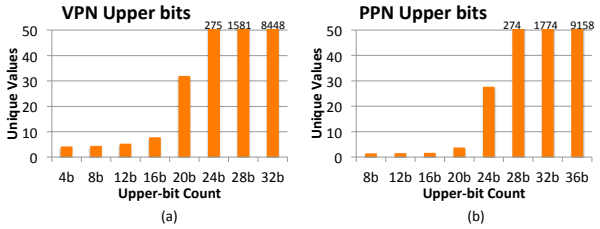

Figure 5. The number of unique values when only considering the x upper-most bits for the VPN (a) and PPN (b), as x is varied. The upper 16 VPN bits and 20 PPN bits change rarely in our experiments.

the clustered TLB. Otherwise, the single originally-requested translation is installed into the conventional C0 TLB. The idea is that the clustered-TLB entries provide greater encoding density (i.e., valid translation per unit area) when the coalescing degree is high. If it is more efficient to store a few (less than θ) translations in C0, then that should be done instead.

Clustered TLB Eviction: When an entry is evicted from the clustered TLB, it is possible that it contains one or more valid translations. One option is simply to drop them all; if any are needed, a subsequent TLB miss will cause them to be re-fetched by the PTW. This could cause an increase in TLB misses. Another approach is to take them all and place each translation in individual entries of the conventional C0 TLB. This approach is unfortunately space-inefficient (which is why we clustered them in the first place). We instead take a middle-of-the-road approach in which only those sub-entries that have their referenced bits set are "saved" and installed into the C0 TLB. The remaining translations are dropped. Apart from preserving useful translations, this is an important optimization because it provides a way to convert a clustered-TLB entry with a high degree of coalescing but low actual usefulness into more efficiently stored conventional TLB entries (i.e., clustered spatial locality does not buy you anything if the coalesced entries are never used, and this policy allows such clustered TLB entries to be "de-coalesced").

4.3. Frequent Value Locality in the Address Bits

Our baseline clustered TLB design exploits the fact that the upper bits of the VPNs and PPNs of nearby PTE entries often contain the same values. We observed that this spatial locality in the addresses' bit patterns also occurs on a global scale.

Consider the layout of typical virtual address spaces. A program's virtual memory space usually is partitioned into a few, large logical regions corresponding to the program's text, heap, stack, etc. These usually are contiguous in the virtual address space, which creates a few frequently used memory regions. When considering only the few most-significant bits of virtual addresses of all valid PTEs, we typically find only a few unique values.

Figure 5(a) quantifies this entropy by showing the number of unique values (y-axis) used by the most significant bits from the VPN (x-axis) on average across our benchmarks. For example, when considering the 12 most-significant bits of the VPN (we use 48-bit virtual addresses), on average we only observe five unique values. This is similar to past work in *frequent value locality* (FVL) that showed that memory locations often store values drawn from a small set of common values (e.g., zero) [13,17]. In this case, we effectively observe that similar FVL exists in the upper bits of the VPNs.

In a similar fashion, Figure 5(b) shows that the most significant PPN bits tend only to use a few unique values. In particular, we find that the 20 most-significant bits of the PPN (we use 48-bit physical addresses) use only one of four unique values on average. This is due partially to the tendency of OSs to cluster physical pages so that transfers to disk make good use of high disk bandwidth.

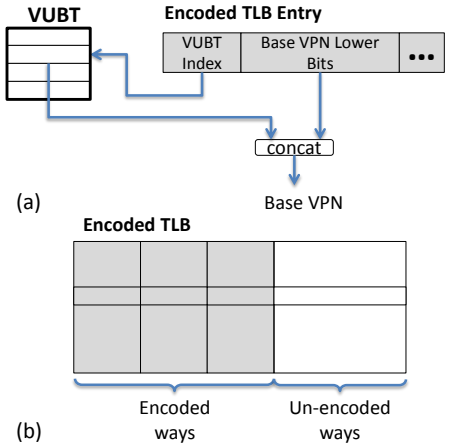

Figure 6. (a) Hardware organization for the Virtual Upper Bits Table (VUBT) and an encoded TLB entry, and (b) a four-way TLB with three encoded ways and one un-encoded way.

We leverage that the most-significant VPN and PPN bits typically are drawn from a limited set of unique values to optimize the MG-TLB further. This also can be used for the baseline TLB and CoLT.

FVL-Support for TLBs: We partition the TLB's base VPN field into the upper bits that tend to come only from a small set of unique values, and the lower bits that exhibit greater value diversity. Figure 6(a) shows an auxiliary structure called the *virtual upper bits table* (VUBT). This stores the commonly-occurring upper bits of the base VPNs. The TLB entry now is modified such that the VPN's upper bits are removed and replaced with an index into the VUBT. To reconstruct the entry's VPN, the VUBT index selects one of the VUBT entries that provides the upper bits of the VPN. The remaining bits of the VPN come from the TLB entry itself. A similar *physical upper bits table* (PUBT) encodes the upper bits of the PPNs. This can be applied to the clustered TLB as well as the conventional TLB. We call such an entry an *encoded* TLB entry.

The VUBT (or PUBT) is limited in size, and so if a VPN's (or PPN's) upper bits do not match any of the entries of the VUBT (PUBT), then the translation cannot be stored in an encoded TLB entry. To support situations when the number of unique VPN upper-bit values exceeds the capacity of the VUBT, we use a hybrid TLB structure in which some ways support the encoded scheme, and other ways use a conventional VPN format. Figure 6(b) shows an example four-way TLB in which the first three ways use encoded TLB entries, and the last way uses un-encoded entries.

Look-up: Look-up proceeds as with a conventional TLB in which the VPN (or base VPN) is used to select a set. For each encoded way, the VUBT index first is used to look up the VPN upper bits from the VUBT. This is concatenated with the stored lower bits to form the overall VPN (or, more accurately, the VPN tag). For the un-encoded ways, the entire VPN (tag) can be read directly from the TLB entries. At this point, each way now has a fully decoded VPN tag, and this can be compared to the requested VPN to determine if there is a TLB hit.

If there is a hit in an encoded entry, the stored PUBT index is used to select the upper PPN bits from the PUBT, which are then concatenated with the lower PPN bits stored in the encoded TLB entry. A hit in an un-encoded way simply uses the PPN already stored in that TLB entry.

Fill: On a TLB miss, the VUBT is searched to see if any existing entries match the upper bits of the VPN (for the translation we are installing into the TLB). At the same time, a similar search is performed on the PUBT for the upper bits of the PPN. If there are matches in *both* the VUBT and the PUBT, then the translation can be installed in an encoded TLB entry. If the upper bits cannot be found in one of the VUBT entries, then a new VUBT entry is allocated for this new upper-bit value. The translation is installed into an encoded TLB entry (assuming the PPN had a match in the PUBT) and the encoded entry stores the index of this newly allocated PUBT entry. A symmetric operation is performed if the PPN's upper bits do not match any existing PUBT entry. If a VUBT or PUBT entry cannot be allocated (i.e., the VUBT or PUBT is full), then the translation is installed into a un-encoded TLB entry.

VUBT and PUBT Management: When an application (process) is first context-switched onto a core, a new page table base pointer (i.e., CR3) is loaded and the TLB is flushed. At the same time, we also flush the VUBT and PUBT. As previously un-encountered VPN and PPN upper-bit values are encountered, they will be allocated new entries in the VUBT and PUBT, respectively. The entries are allocated in order; thus, instead of per-entry valid bits, a single allocation counter per table is needed.

Eventually, one or both of these may fill up, at which point any new VPN or PPN upper-bit values will cause the corresponding translations to be restricted to the un-encoded entries in the TLB. Our characterization (Figure 5) showed that typically there are only a few unique VPN and PPN upper-bit values, and so very small VUBT and PUBT sizes are needed in the vast majority of cases. Based on the characterization results, we set the VUBT size at eight entries and the PUBT size at four. A small VUBT table is desirable because to perform the encoded TLB look-up, each encoded way needs to perform a look-up on the VUBT, and therefore the VUBT needs to be multi-ported (each look-up is a RAM look-up). Similarly, on a fill operation, we need to check if the current VPN upper-bits are already present in *any* of the VUBT entries, which requires a single CAM port. Keeping the VUBT small makes the extra ports not very expensive. The PUBT is slightly simpler because only the hitting way needs to perform a look-up, and so it can be limited to a single RAM port and a single CAM port.

Even if the VUBT or PUBT fills up, translations with the upper-bit values not in these tables will continue to be cached in the TLB's un-encoded ways. The monotonic, write-only allocation of the VUBT/PUBT may seem like a problem, but these will be flushed on every context switch. If a single program runs for a very long time without ever being context-switched out by the OS, it would be trivial to have the processor periodically (but fairly infrequently) flush the TLB and VUBT/PUBT. The performance impact is minimal if this flush interval is sufficiently long. For a simultaneous multi-threaded (SMT) processor, we could have one set of UBTs per hardware thread. This avoids flushing all UBTs when only a single thread is context-switched. Given the small size of the UBTs, the space overhead is minimal (typical SMTs are only two-threaded).

4.4. Hardware Cost

4.4.1. Basic Multi-granular TLB Hardware Cost Table 1 compares area cost and reach for conventional TLB, CoLT-SA (the set-associative version of the CoLT TLB proposed by Pham et al. [12]), and the MG-TLB.

Conventional TLB: 512 entries, four ways. Each entry has 29 bits for the tag, 40 bits for the PPN, and 5 bits for the attribute. In total, we have 75 bits per entry, which adds up to 37,888 bits (4.625KB). The other designs are configured to target a similar bit-storage budget.

Table 1. Comparison of Hardware Cost

	Baseline L2TLB	CoLT-SA	Cluster-C3	Cluster-C0
Entries	512	512	128	320
Assoc.	4	4	4	4
Max Reach	512	2,048	1,024	320
Min Reach	512	512	128	320
Attr. Bits	5	5	5	5
Tag Bits	29	27	28	30
Data Bits	40	48	85	40
Entry Bits	74	80	118	75
Total Bits	37,888	40,960	15,104	24,000

CoLT-SA: 512 entries, four ways. Each CoLT entry has only 27 bits for the tag because we left-shift the VPN by 2 bits to compute the set index; 40 bits for the base PPN; 8 bits for 4 sub-entries, each sub-entry in the contiguous range has 1 valid bit and 1 dirty bit, and 5 bits for the attribute. As a result, each CoLT entry has 80 bits, which gives us 40,960 bits total, or 8% area overhead compared to the baseline.

Multi-granular TLB: We allocate about one-third of the storage budget to C3 and two-thirds of the budget to C0. This results in 128 C3 entries, and 320 C0 entries[6]. Each C3 entry has 5 bits for the attribute; 28 bits for the tag because we ignore the bottom 3 bits of the VPN; 37 bits for the base PPN because we also ignore the bottom 3 bits of the PPN; and, eight sub-entries, with each sub-entry having 6 bits (valid, modified, referenced, and 3 bottom bits of the corresponding PPN). Hence, 128 C3 entries requires 15,104 bits. Each C0 entry has 5 bits for the attribute, 30 bits for the tag, and 40 bits for the PPN. Therefore, 320 C0 entries require 24,000 bits. This MG-TLB configuration requires 39,104 bits which is close to the baseline (3%) and less than CoLT.

Given these configurations, the conventional TLB always has a reach of 512 pages. CoLT-SA can have a coverage of up to 2,048 pages (i.e., if each of the 512 entries fully coalesces four PTEs), while the MG-TLB has maximum reach of 1,024 in the C3 table (i.e., if each of the 128 clustered-TLB entries is fully populated with eight translations) plus the 320 entries in the C0 TLB for a total of 1,344 possible translations. In the worst case, CoLT-SA has reach of 512 pages, while the MG-TLB has a reach of 448. At first glance, CoLT-SA may appear to be better in terms of reach than the MG-TLB, but this is true only if CoLT-SA can find enough contiguous spatial locality. We will show that the MG-TLB's effective reach is superior to CoLT-SA's because, in practice, clustered spatial locality is easier to find than strict contiguous locality.

4.4.2. Enhanced Multi-granular TLB Hardware Cost Table 2 shows the configuration of the MG-TLB when we exploit the FVL in the upper bits of VPNs and PPNs. We use a similar relative area allocation between C3 and C0 as before; however, for each encoded TLB entry, we need to keep additional bits for the VUBT and PUBT indexes. We assume a VUBT with eight entries, and a PUBT with four entries; therefore, the respective indexes are 3 and 2 bits each. In addition, a small part of the area budget is used to implement the VUBT and PUBT (along with one small allocation counter for each). Each VUBT or PUBT entry is 16 bits and the VUBT and PUBT counters are 4 and 3 bits, respectively, so in total we need 132 bits for the VUBT and 67 bits for the PUBT. Overall, the total area cost is 39,075 bits, or **3%** of area overhead compared to the baseline, which also is much less than CoLT.

Despite those additional bits, by replacing the 16 upper bits of the VPN or the PPN with a 3-bit VUBT or 2-bit PUBT index, re-

[6] In a real implementation, the number of C0 entries (or at least the number of sets) would be a power-of-two. For the purposes of maintaining similar storage budgets across each type of TLB for fair comparisons, we used a non-power-of-two size.

Table 2. Enhanced MG-TLB Hardware Cost

	C3's Full Len Way	C3's Encoded Way	C0's Full Len Way	C0's Encoded Way
Entries	38	114	109	327
Max Reach	304	912	109	327
Min Reach	38	114	109	327
Attr. Bits	5	5	5	5
Tag Bits	29	12	31	14
Data Bits	85	69	40	24
VEncode Bits	3	3	3	3
PEncode Bits	2	2	2	2
Entry Bits	119	91	76	48
VUBT Bits (Shared)	132	132	132	132
PUBT Bits (Shared)	67	67	67	67
Total Bits	4,522	10,374	8,284	15,696

Table 3. Summary of benchmarks used in our studies

Benchmarks	Suite	Page Walk Overhead
xalancbmk	SPEC CPU2006	9.4%
SPECweb-B	SPECweb2005	9.5%
TPC-C	TPC	8.6%
GemsFDTD	SPEC CPU2006	9.2%
Graph Analytics	CloudSuite	17.7%
Trade6	IBM WebSphere	11.1%
Data Serving	CloudSuite	8.8%
Data Caching	CloudSuite	20.0%
astar	SPEC CPU2006	19.5%
omnetpp	SPEC CPU2006	26.4%
mcf	SPEC CPU2006	33.8%

spectively, we can save quite a bit of space, which allows us to add more entries to both the C3 and C0 TLBs. While maintaining approximately the same bit-budget as the un-encoded MG-TLB, C3 and C0 TLBs that each use three encoded ways allow us to have 152 C3 entries and 436 C0 entries. This increases the maximum reach of the MG-TLB to 1,652 (from 1,344 in the basic design) and the minimum reach from 448 to 588. We will show that this design performs the best out of all of our evaluated configurations.

5. Experimental Methodology

This section describes the infrastructure used to evaluate our multi-granular TLB and two comparison points: a baseline conventional TLB and the recently proposed CoLT design.

5.1. Workloads

Table 3 shows the workloads evaluated in our study. We consider a wide range of applications, from scientific workloads to server and cloud workloads, and select benchmarks with non-negligible TLB miss overheads. We also evaluated eight benchmarks with low TLB sensitivity (from SPEC CPU2006, SPECjbb2005, web browsing, and gaming; results for these benchmarks are omitted for space), and their results are consistent with the observations we show in this work. We collect traces of at least 50 million instructions per benchmark using AMD's SimNow^{TM} [4] full-system simulator software. The traces capture issued user and system instruction and data references and record the virtual and physical page address pairs. We also correlate the traces against hardware performance counters to ensure that they capture a representative execution phase of the benchmarks.

5.2. Simulation Infrastructure

5.2.1. Functional Simulator For fast design-space exploration of our multi-granular TLB in comparison to the baseline TLB and

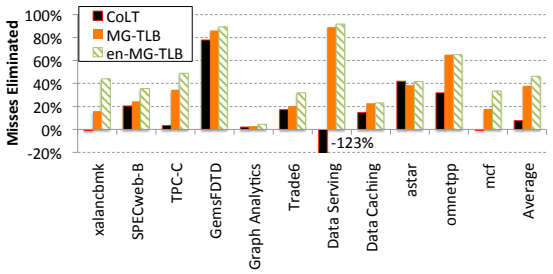

Figure 7. L2 TLB misses eliminated by the baseline multi-granular TLB (MG-TLB), enhanced MG-TLBs with structures to exploit redundant most significant VPN and PPN bits (en-MG-TLB) and CoLT. MG-TLB and en-MG-TLB comprehensively eliminate more misses than CoLT.

CoLT, we use a functional simulator that models a two-level TLB with 64-entry L1 instruction and data TLBs. We assume a baseline L2 TLB of 512 entries, similar to current products [12]. Because our multi-granular TLB targets the L2 level, we compare this against the benefits of CoLT on just the L2 TLB. All TLBs have four-way associativity.

5.2.2. Performance Evaluation We use an in-house trace-driven timing simulator derived from the MacSim simulator [10], using a two-wide in-order core. It models three-level cache hierarchies, two-level TLBs, a hardware page-walk unit complete with a page-walk cache, and a detailed DRAM model. The TLBs have associated miss status holding registers (MSHRs) to model pipelined accesses. We also calibrated the page-walk overheads of our timing simulator against hardware measurements on a real machine. As shown in Table 3, overheads for our benchmarks range from a problematic 9% for Data Serving to a severe 34% for mcf.

6. Multi-granular TLB Evaluations

Our multi-granular TLB (MG-TLB) enjoys a number of design options. We evaluate the performance implications of various options in this section.

6.1. Understanding Changes in Hit Rates

Figure 7 compares the percentage of L2 TLB misses eliminated (compared to a standard baseline four-way, 512-entry TLB) when using CoLT (512-entry); MG-TLBs without exploiting the frequent value locality in the VPN and PPN's upper bits (128-entry C3 TLBs with a 320-entry conventional C0 TLB); and the encoded MG-TLBs (en-MG-TLB) that leverage the upper-bit frequent value locality (152-entry C3 TLBs with 436-entry standard C0 TLB).

MG-TLBs eliminate more TLB misses than CoLT, averaging 38% miss eliminations (30% more than CoLT). FVL-based encoding (en-MG-TLB) only boosts this difference, eliminating on average 46% of the TLB misses. More specifically, we note the following three observations:

First, benchmarks that exhibit more clustered spatial locality than contiguous spatial locality (e.g., Data Serving, TPC-C) also eliminate more misses with MG-TLB than with CoLT. In fact, CoLT actually provides a negative result for Data Serving, mostly because the change in set-indexing scheme outweighs its ability to exploit contiguous spatial locality. Fortunately, exploiting clustered spatial locality overcomes this issue, eliminating the vast majority (more than 80%) of the TLB misses. Enhancements provide additional benefits.

Second, benchmarks like mcf or Data Caching, which show more contiguous spatial locality, *still benefit more from MG-TLB than CoLT*. We find that this occurs because changes to CoLT's set-

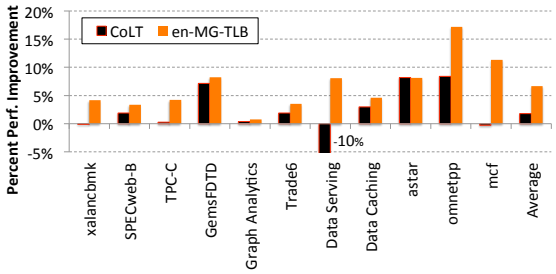

Figure 8. Performance improvements when using CoLT and en-MG-TLB. Our approach outperforms CoLT *in every single case*.

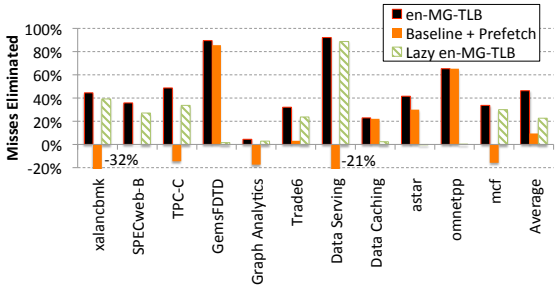

Figure 9. Separating the prefetch and capacity benefits of MG-TLBs.

indexing scheme undo the benefits of exploiting contiguity. Instead, our dual approach of leveraging clustered spatial contiguity and allowing a small conventional L2 TLB for singleton PTEs is more beneficial. As a result, MG-TLB eliminates 20% more of mcf's TLB misses than CoLT.

Third, en-MG-TLB improvements relative to MG-TLB are non-trivial for xalancbmk, TPC-C, Trade6, and mcf. In these benchmarks, the FVL in the upper bits makes even relatively small VUBT and PUBT tables highly effective.

6.2. Overall Performance Improvements

Figure 8 compares the performance improvement of en-MG-TLB with CoLT. Our approach outperforms CoLT in every case except astar. In some cases, the performance difference is significant (close to 18% for Data Serving, 12% for mcf, and 10% for omnetpp). On average, we outperform CoLT by about 5%, but we purposefully included benchmarks in which neither has much benefit (e.g., Graph Analytics) to demonstrate that the MG-TLB approach does not *hurt* performance when the spatial locality is sufficient, as well as benchmarks in which CoLT truly does well (e.g., astar, in which CoLT performs slightly better than MG-TLB). As TLB overheads continue to rise [3, 5], the expected benefit of TLB coalescing techniques such as MG-TLB would be expected to increase.

6.3. Prefetching versus Capacity Improvements

Our multi-granular TLB eliminates misses in two main ways. First, on a TLB miss, it speculates that PTEs near the one that is requested may be useful. This benefit is similar to prefetching because it is not known whether clustered PTEs will be useful in the future. However, unlike classical prefetching, which must evict an existing entry to make room for a new one, clustered TLBs use the same entry for the entire clustered packet. Second, because each clustered entry provides a higher reach, there is a capacity improvement relative to the standard approach, for the same total area.

Figure 9 teases apart the relative benefits of these two factors by plotting (1) TLB miss-elimination rates for MG-TLBs; (2) TLB miss elimination when the same clustered PTEs are prefetched into a standard 512-entry L2 TLB; and, (3) a lazy MG-TLB approach, in which only the desired PTE is inserted into the clustered TLB on demand, but which then integrates the other PTEs in the cluster if they are demanded in the future. Overall, this comparison informs us whether most of en-MG-TLB's benefits arise from prefetching effects or its superior reach.

Figure 9 shows that the relative benefits vary per benchmark. Some benchmarks (e.g., GemsFDTD, Data Caching, omnetpp) gain from prefetching. In fact, for some of these (e.g., omnetpp), the capacity benefit is almost negligible. The other benchmarks however, en-MG-TLB and the Lazy (no prefetch) version perform similarly, making clear that capacity is the main benefit. For some benchmarks, prefetching alone is negative (xalancbmk, Data Serving, etc.) because capacity improvement is the key to overall performance boosts.

Figure 10. TLB miss-elimination rates assuming that the clustered TLB is C2, C3 (our default assumption so far), and C4

7. Sensitivity Studies

MG-TLB has a number of parameters crucial to its overall performance. We investigate these parameters in this section.

Cluster Size: We have thus far assumed that the MG-TLB uses a C3 clustered TLB. Figure 10 shows how TLB miss-elimination rates change when C2 and C4 TLBs are used instead. Larger clustering potentially exploits more clustered spatial locality. At the same time, each entry's size increases, decreasing the total number of entries. Moreover, the selected index bits are further left-shifted, increasing the possibility of conflict misses when clustered spatial locality is insufficient.

Figure 10 shows that C3 tends to perform best on most benchmarks. In some cases like data Caching, which is known to generate large clustered spatial locality due to slab allocator use, C4 outperforms C3. However, benchmarks like Data Serving and xalancbmk are degraded at C4.

Coalescing Thresholds: Our MG-TLB designs have assumed that at least $\theta = 2$ PTEs must be clustered for insertion into the clustered TLB. Figure 11 shows how this assumption affects miss rates by varying θ from 1 to 4 for en-MG-TLB with a C3 clustered TLB. We see that a value of 2 is generally the best (and is markedly better for some benchmarks like Data Serving and xalancbmk). Intuitively, this makes sense because a single C3 clustered TLB consumes less space than two conventional L2 TLB entries. Therefore, if we coalesce two PTEs and place them in a single C3 entry, we expend fewer bits in storing them compared to the small conventional L2 TLB for singleton PTEs. When θ goes beyond 2, these two PTEs are stored in two separate C0 TLB entries, wasting space.

Sizing MG-TLB Components: We now consider how the relative sizes of the MG-TLB's conventional small L2 TLB and clustered

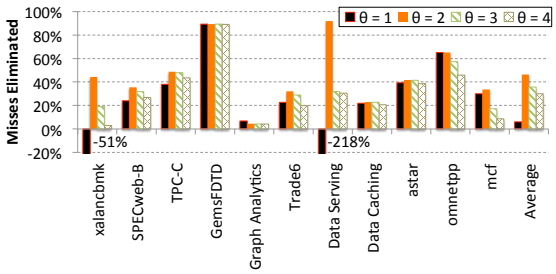

Figure 11. TLB miss-elimination rates for en-MG-TLB as the cluster threshold is changed for insertion into the clustered TLB.

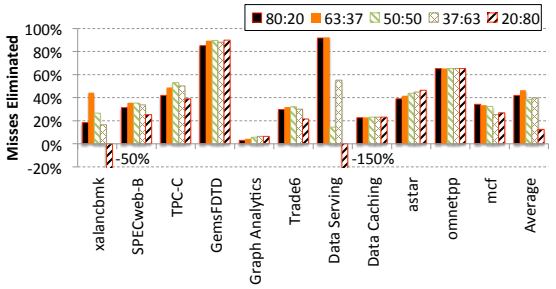

Figure 12. TLB miss eliminations for different relative sizes of the small singleton PTE's TLB and the clustered TLB in MG-TLB. The legend shows the ratio of the MG-TLB area for the small conventional TLB to the area for the clustered TLB.

L2 TLB affect TLB miss rates. Our default scheme devotes about one-third of MG-TLB area for the clustered TLB and two-thirds for the conventional L2 TLB. Figure 12 shows how TLB miss eliminations vary when these values are changed. The x:y ratio tells us what portion of the MG-TLB area goes to the conventional L2 TLB (x) and the clustered TLB (y). Generally, we find that TLB misses are best for our default configuration (2:1), though other configurations see similar gains. However, MG-TLB effectiveness perceptibly diminishes when the conventional L2 TLB becomes much smaller in comparison (e.g., 1:4), indicating that we must adequately cache the singleton PTEs that do not experience clustered spatial locality.

Sensitivity to VUBT and PUBT Size: Our default en-MG-TLB uses 8-entry PUBTs and 4-entry VUBTs. We have varied these sizes to study their impact on TLB miss rates. In general, PUBTs rarely require additional entries, whereas VUBTs require the use of the dedicated full-length way in rare instances. Overall, even 128-entry VUBTs and PUBTs provide negligible performance improvements to our approach.

MG-TLB Effectiveness for Different Sizes: Our evaluations compare MG-TLB to a baseline 512-entry L2 TLB. Therefore, all our designs are sized to meet this total area. However, we also have studied cases in which we have half and double the total area to play with (i.e., our baseline L2 TLB becomes 256 or 1,024 entries). We find that MG-TLB (and its encoded counterparts) consistently outperforms both the baseline L2 TLB and CoLT for these sizes, and that its performance benefits increase when more area is available for some benchmarks (e.g., mcf, Data Serving). This bodes well for future designs which will likely have more resources available for address translation.

8. Conclusions

This paper is the first to observe that significant amounts of clustered spatial locality exists in applications. This form of spatial locality is present across a variety of system use cases and configurations (e.g., even in fragmented systems) and largely subsumes previously-observed contiguous spatial locality. In response, we propose a multi-granular TLB that identifies PTEs in which groups of nearby virtual pages are mapped to groups of nearby physical pages. By coupling a clustered TLB for these types of PTEs with a small conventional L2 TLB, we consistently outperform past work on coalesced TLBs despite using modest hardware and requiring no dedicated software support.

Our best-performing design point eliminates 46% of L2 TLB misses, resulting in a 7% CPU cycle reduction for a wide range of applications. Our proposed TLB organization substantially increases the effective TLB reach with only modest hardware changes while requiring no OS support, providing a promising solution for emerging big data applications.

Acknowledgments

We thank the anonymous reviewers for their feedback on this work. Part of this work was supported by the National Science Foundation under Grant No. 1253700.

References

[1] A. Arcangeli. Transparent Hugepage Support. *KVM Forum*, 2010.
[2] T. Barr, A. Cox, and S. Rixner. SpecTLB: A Mechanism for Speculative Address Translation. *ISCA*, 2011.
[3] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift. Efficient Virtual Memory for Big Memory Servers. *ISCA*, 2013.
[4] R. Bedicheck. SimNow: Fast Platform Simulation Purely In Software. *HOTCHIPS*, 2004.
[5] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. *ASPLOS*, 2008.
[6] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. *HPCA*, 2010.
[7] A. Bhattacharjee and M. Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. *ASPLOS*, 2010.
[8] J. B. Chen, A. Borg, and N. Jouppi. A Simulation Based Study of TLB Performance. *ISCA*, 1992.
[9] 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.
[10] GaTech. Macsim. *http://code.google.com/p/macsim/*.
[11] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *ISCA*, 2002.
[12] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. *MICRO*, 2012.
[13] T. Sato and I. Arita. Low-Cost Value Predictors Using Frequent Value Locality. *ISHPC*, 2002.
[14] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-Based TLB Preloading. *ISCA*, 2000.
[15] S. Srikantaiah and M. Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. *MICRO*, 2010.
[16] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. *ASPLOS*, 1994.
[17] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-Centric Data Cache Design. *ASPLOS*, 2000.