Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations

Chang Hyun Park Taekyung Heo Jungi Jeong Jaehyuk Huh
School of Computing, KAIST
{changhyunpark,tkheo,jgjeong,jhuh}@calab.kaist.ac.kr

ABSTRACT

To mitigate excessive TLB misses in large memory applications, techniques such as large pages, variable length segments, and HW coalescing, increase the coverage of limited hardware translation entries by exploiting the contiguous memory allocation. However, recent studies show that in non-uniform memory systems, using large pages often leads to performance degradation, or allocating large chunks of memory becomes more difficult due to memory fragmentation. Although each of the prior techniques favors its own best chunk size, diverse contiguity of memory allocation in real systems cannot always provide the optimal chunk of each technique.

Under such fragmented and diverse memory allocations, this paper proposes a novel HW-SW hybrid translation architecture, which can adapt to different memory mappings efficiently. In the proposed *hybrid coalescing* technique, the operating system encodes memory contiguity information in a subset of page table entries, called *anchor entries*. During address translation through TLBs, an anchor entry provides translation for contiguous pages following the anchor entry. As a smaller number of anchor entries can cover a large portion of virtual address space, the efficiency of TLB can be significantly improved. The most important benefit of hybrid coalescing is its ability to change the coverage of the anchor entry dynamically, reflecting the current allocation contiguity status. By using the contiguity information directly set by the operating system, the technique can provide scalable translation coverage improvements with minor hardware changes, while allowing the flexibility of memory allocation. Our experimental results show that across diverse allocation scenarios with different distributions of contiguous memory chunks, the proposed scheme can effectively reap the potential translation coverage improvement from the existing contiguity.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Virtual memory**; Allocation / deallocation strategies;

KEYWORDS

Virtual memory, address translation, TLB coalescing

ACM Reference format:
Chang Hyun Park Taekyung Heo Jungi Jeong Jaehyuk Huh School of Computing, KAIST . 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080217

1 INTRODUCTION

With ever increasing memory capacity demands for large memory applications, address translation for virtual memory support has become a critical performance bottleneck of the applications. To improve the address translation efficiency, there have been two different approaches. The first approach, *coverage improvement*, expands the translation coverage of the translation looakaside buffer (TLB) within a given area and power budget [5, 13, 21, 22, 30, 31, 33, 34, 40]. The second approach, *TLB miss penalty reduction*, decreases miss handling latencies after memory requests miss in the TLB [3, 4, 8, 19, 29, 36]. Although both approaches are important for translation performance, enhancing translation coverage can provide direct performance improvements, as the latency can be completely hidden for TLB hits.

To improve the translation coverage, there have been a spectrum of techniques employed in commercial systems or proposed in recent studies. One of the most common approaches is to increase the page size. In commercial x86 architectures, 2MB and 1GB page sizes are supported in addition to the traditional 4KB page size. Increasing the page sizes dramatically improves the coverage of a TLB entry by 512 times with a 2MB large page. However, to fully exploit the improved coverage, operating systems (OS) must be able to allocate a 2MB chunk of physical memory for each page. A more radical way of improving the translation coverage is to use variable-sized HW segment translation instead of page-based translation [21]. Its effectiveness also relies on whether the OS can allocate a very large contiguous memory chunk for each segment. An alternative to large pages and segments is HW-based coalescing techniques [33, 34]. *CoLT* and *cluster TLB* coalesce multiple page translations into a single TLB entry as long as their physical locations are in a contiguous region. Unlike segments or large pages, the pure HW-oriented techniques opportunistically find the contiguous pages and coalesce them to a single TLB entry. The operating system can improve the coalescing chances by allocating contiguous pages, but it does not need to guarantee certain chunk sizes.

However, these approaches have trade-offs in two aspects: *allocation flexibility* and *scalability of translation coverage*. Supporting large pages allows only limited numbers of page sizes, which restricts the scalability of coverage. Segments have the highest scalability of coverage, as they can support variable-length translation with virtually no limit. However, memory allocation requires a much stricter discipline, applicable only when a limited number of very large chunks of physical memory can cover the entire memory footprint. On the other hand, HW coalescing allows flexible allocation,

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

but coverage scalability is limited to four to eight pages, as coalescing must be purely done by the HW components.

A common requirement of the prior techniques is that the OS must be able to consistently provide contiguous memory chunks suitable for each technique. However, recent studies show that such contiguous chunk allocation is not always possible or can even degrade the performance of multi-socket NUMA systems [17, 24]. In addition to the common NUMA architectures, the emerging new memory architectures, such as 3D stacked DRAMs, network-connected hybrid memory cube (HMC), and non-volatile memory (NVM) , can further increase the non-uniformity in memory [14, 20, 26, 28, 32, 35]. Such memory heterogeneity requires fine-grained memory mapping to place frequently accessed pages on fast near memory, complicating the allocation of large contiguous memory chunks.

Due to the non-uniformity in memory architectures, the OS cannot always provide the best allocation tuned for different coverage improvement techniques. Even for the same application, the actual memory allocation status changes drastically depending on the system state, which incurs severe performance variation [24]. One technique may work well with a certain memory allocation scenario, but may not work well, if the OS cannot provide the optimal allocation for the technique. Therefore, an ideal coverage improvement technique needs to be able to adapt to diverse memory allocation states. In this paper, we propose a hybrid address translation technique adaptable for diverse memory allocation scenarios, while improving the translation coverage whenever possible. The hybrid technique utilizes the high-level mapping information available to the operating system, and requires minor changes in translation architectures, using mostly the same TLB structures and page tables.

The new translation architecture, called *hybrid coalescing*, encodes the contiguous allocation information in a subset of page table entries, called *anchor entries* designated at every *N* page table entries. The anchor entry must contain how many following pages are allocated in contiguous physical memory. For an L1 TLB miss, if the requested page address is not in the L2 TLB, the nearest anchor entry for the page number is searched in the L2 TLB. If the requested page is part of the contiguously allocated memory region, the anchor entry can provide the translation by simply adding the virtual address difference between the anchor and requested pages, to the physical page address of the anchor entry. The rationale behind the anchor-based translation is that if the majority of application memory is allocated in some distributions of contiguous chunks, most of the TLB entries will be filled with anchor entries. Each anchor entry can cover a large portion of memory translation.

The proposed translation techniques can adapt to various memory allocation scenarios, since the OS can change the density of anchor entries in page tables. If memory allocation can provide mostly contiguous chunks of memory, the anchor entries are sparsely located. If the memory allocation is fragmented to small chunks, the anchor entries are densely populated. Using the flexible anchor density, hybrid coalescing can exploit the memory allocation contiguity as much as possible, even if the contiguity states are diverse. As the OS encodes the contiguity information in anchor entries, the proposed scheme can eliminate the HW overheads of finding and coalescing contiguous pages in HW-based coalescing techniques. Furthermore, without the HW restriction, the proposed technique can vastly increase the translation coverage of each anchor entry.

This paper is the first study to propose a HW-SW hybrid TLB coalescing, providing both scalable coverage and allocation flexibility. The proposed technique has the following strengths over the prior approaches. First, it can dynamically change different chunk sizes for translation coalescing to adapt to the currently available contiguity in memory allocation. Second, unlike fixed large pages, the OS does not need to provide a strict fixed chunk allocation. Hybrid coalescing can extract the available contiguity as much as possible, even if a certain fixed contiguity is not provided. Third, the proposed scheme can support a highly scalable coverage improvement, as the contiguity is encoded in the page table. Finally, the changes to the current TLB and page table structures are minor.

In the experimental results, the paper shows that under various memory allocation scenarios, the proposed scheme can provide the best performance consistently. The proposed scheme outperforms or performs similar with the best prior scheme for each mapping scenario, achieving the best average performance across diverse scenarios.

The rest of the paper is organized as follows. Section 2 discusses challenges posed by address translation and memory heterogeneity. Section 3 describes the proposed hybrid translation techniques, and Section 4 discusses how to find the best distance for anchor placements. Section 5 presents the experimental results. Section 6 discusses the related work and Section 7 concludes the paper.

2 MOTIVATION

2.1 Translation Coverage Improvement

The first approach to improve the translation coverage is to use multiple page sizes. In the current x86 architectures, 2MB and 1GB page sizes are supported in TLBs. Applications may explicitly request for large pages during memory allocation, or may make use of the transparent huge page (THP) support, the operating system can assign 2MB pages, if 2MB chunks are available. Using a couple of different page sizes does not incur significant complexity in the TLB designs, and thus, the latest architecture can support both 4KB and 2MB pages in the L2 TLBs without requiring separate TLBs for each page size, although the 1GB pages use a separate and smaller 1GB page L2 TLB. However, a disadvantage of large page sizes is that its coverage is still limited with only a few possible page sizes, and the scalability of its coverage will be eventually limited. Furthermore, the OS must always assign a fixed large chunk to benefit from the translation coverage improvement.

To drastically increase the translation coverage, the second approach uses variable-sized segments. Direct segments and Redundant Memory Mapping (RMM) support one or multiple segment regions of variable length [5, 21]. For each segment region, the operating system must allocate a contiguous chunk of memory. As long as such contiguous memory allocation is possible, the translation coverage of a single segment can scale to a very large region of virtual address space, practically eliminating much of the address translation costs. However, as the number of HW segment translation entries is much smaller than the current TLB size due to the fully associative range search required for segment translation, each process can

	THP	Cluster/CoLT	RMM	Our Approach
Scalability	Mod.	Mod.	Good	Good
Flexibility	Mod.	Flex.	Restr.	Flex.

Table 1: Comparison of scalability and allocation flexibility (Mod.: moderate, Flex.: flexible, Restr.: restricted)

only use a limited number of segments at a time. RMM supports 32 segment translation entries (*range TLB*) to match the latency of the L2 TLB [21]. Furthermore, for its effectiveness, segment-based translation relies on a very strict huge chunk allocation.

The third approach is the HW-based coalescing technique. As proposed by CoLT and clusterTLB, there are some levels of contiguity in memory allocation as the operating system uses a buddy algorithm to reduce memory fragmentation [33, 34]. In the HW coalescing techniques, the HW TLB controller searches the page table entries and finds contiguously allocated pages. Since a cacheline contains multiple page table entries, the logic can efficiently search through multiple page table entries without issuing separate memory accesses. Although this approach does not rely on the strict allocation of contiguous pages, as the HW controller exploits the contiguity opportunistically, the scalability of translation coverage is quite limited. To allow efficient lookups of coalesced entries in TLBs, they support only a limited coalescing capability of 4-8 pages in a TLB entry. CoLT additionally provides a fully associative mode that supports a much larger number of coalesced contiguous pages. However, it requires a fully associative lookup, which in turn restricts the number of entries available.

The three approaches have different trade-offs in their HW cost, allocation flexibility, and scalability of coverage. Large page supports have the smallest extra HW cost, but the allocation flexibility and coverage scalability are moderate. The segment-based translation has the highest coverage scalability, but requires a strict memory allocation with some extra HW for the direct segment registers. RMM requires a fully associative range search, which severely limits the size of the range TLB. The limited range TLB restricts the OS to allocate only a very limited number of huge contiguous memory regions for each process. Finally, HW coalescing allows flexible memory allocation with fine-grained memory mapping, but the coverage scalability can be the most limited among the three approaches. Table 1 presents the comparison of the prior techniques.

2.2 Increasing Non-Uniformity in Memory

Although improving translation coverage requires allocation of contiguous memory chunks, increasing non-uniformity in memory architectures within a system can pose a challenge for using large pages. The majority of datacenter systems use multi-socket NUMA nodes for better system density. In the NUMA systems, thread scheduling can often mismatch the memory locations the threads are accessing. Gaud et al. showed that that using 2MB large pages often degrades the performance of multi-threaded applications, and thus large pages must be selectively used to prevent causing unnecessary remote memory accesses [17]. Kwon et al. further investigated the unfair allocation of large pages [24]. In real systems, the availability of large page allocation can fluctuate significantly, and processes often receive large pages inconsistently, causing performance variations.

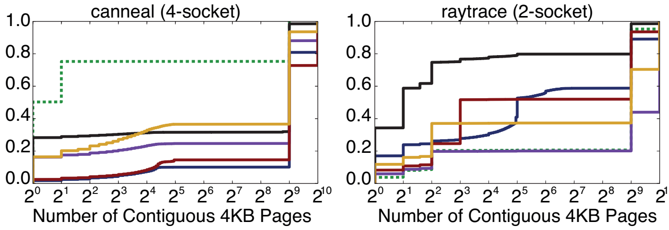


Figure 1: Cumulative distributions of chunk sizes in `canneal` **and** `raytrace`

Both studies show that allocating even moderate sized 2MB pages is not trivial in real systems, when memory non-uniformity exists.

Furthermore, such memory non-uniformity is expected to increase in future, with the advent of 3D stacked DRAM, network-based hybrid memory cube (HMC) [23, 26, 28, 32], and non-volatile memory (NVM) [14, 20, 35]. With the emerging memory technologies, the memory hierarchy can change to multiple levels of memory with different latency and bandwidth characteristics.

The recent studies show that the capacity of stacked memory is large enough to be part of the main memory, and thus the physical address space consists of fast-near memory and far-slow memory regions [28]. In a generic system model for such hierarchical main memory, the operating system uses the virtual-to-physical mapping using page tables to assign either near or far memory pages to the virtual memory space of applications [28]. The HMC architecture provides multiple memory modules connected by on-chip networks [23]. In such networked memory systems, latencies for accessing different modules vary, increasing the non-uniformity of memory access times.

Emerging non-volatile memory also accelerates the heterogeneity of memory. NVM is projected to be slower (in terms of latency and bandwidth) compared to DRAM [14, 20, 35]. However, they are also expected to provide a higher density along with the non-volatility characteristic. Coupling these characteristics, DRAM can be used as a SW cache for hot pages, and NVM as a large backing memory for cold pages. Agarwal and Wenisch proposed an application-transparent page management that makes use of the non-uniformity and different benefits of DRAM and NVM [1]. Such multi-level memory requires fine-grained page mappings to fully exploit its potential benefits. For example, within a 2MB virtual page, only part of the page can be accessed frequently. Storing the entire 2MB page in the near memory can waste the precious near memory space.

2.3 The Effect of Memory Allocation Diversity

This section shows the variance in memory allocation with real systems using the PARSEC benchmark suite. We recorded the mapping contiguity on two different x86 machines, two and four socket NUMA machines, each running Linux 3.16.0 and 3.19.0, respectively. To change the memory mapping status for each run, we executed the workload of interest alone or with randomly executing background jobs chosen from PARSEC. The number of concurrent random background jobs was controlled to pressure the system memory while preventing any memory swapping. We periodically

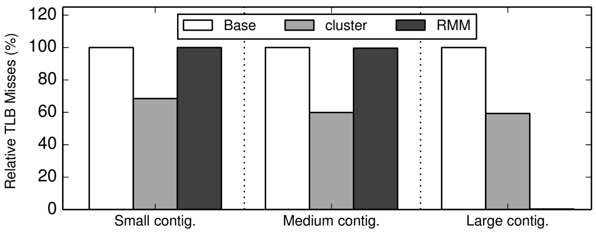


Figure 2: Relative TLB misses of prior techniques with three different mapping scenarios

took memory map snapshots, and for each execution, analyzed the memory map at its largest allocated state.

Figure 1 shows the CDF of two different workloads running on the two and four socket systems. The x-axis is the number of contiguous 4KB pages. The dotted line is the memory mapping when the workload was running alone, while the other lines are the multiple executions with other random background jobs. The figures show a wide variation of memory contiguity when the memory allocation states are varied by the different co-runners or by the system configurations. Without clear patterns, the results confirm that the allocation contiguity varies somewhat randomly based on the initial state and how memory requests are generated from multiple processes.

As the contiguity diversity shows, even the same application running on the same server may receive different memory mappings. Thus, designing a system that works on a specific mapping may not perform as expected when the mapping distribution changes. The same phenomenon, of significant contiguity difference, was also observed by Cox et al., where they found that running an application with `memhog` [34] of differing intensity affected the memory allocation contiguity of an application [10].

Figure 2 presents our evaluation of two prior schemes at different contiguity distributions. The details of the contiguity distributions and configurations are explained in Section 5.1. Cluster TLB (`cluster`) effectively reduces TLB misses for the small chunk configuration, but `RMM` is not effective as the small number of range TLBs cannot cover many small chunks. However, for the large chunks, the benefit of `cluster` is almost similar to that with the small chunk configuration, not getting much more improvements from increased contiguity. On the other hand, `RMM` can almost eliminate TLB misses with the large contiguity configuration.

As the memory mapping of the process is subject to variability, it is necessary to design a translation scheme that performs well in different mapping situations. In this work, we propose a hybrid TLB coalescing mechanism that aims to provide an efficient translation for different types of memory mappings.

3 HYBRID TLB COALESCING

3.1 Anchored Page Table

Large pages, segments, and HW coalescing can expand the coverage of address translation from the limited HW resources. However, as discussed in Section 2.1, the three approaches impose different restrictions on the memory allocation flexibility on the operating systems, while providing different levels of coverage scalability. In this section, we re-balance the role of the operating system and architectural components of the prior approaches to support flexible memory allocation, while supporting better coverage scalability than HW-coalescing.

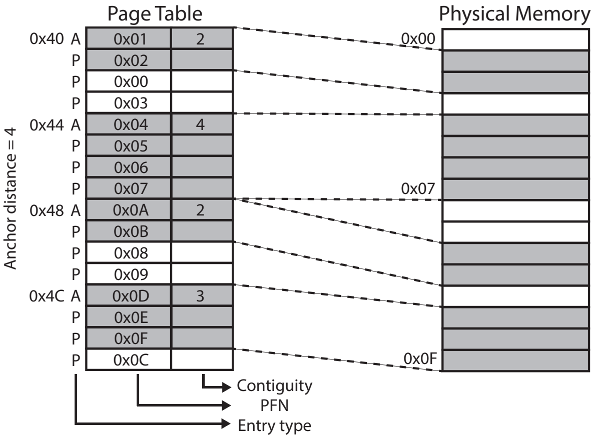


Figure 3: Anchor entries marked in a page table (anchor distance = 4). Each anchor entry maintains mapping contiguity starting from the anchored address

In our approach, instead of relying on the HW logic to identify contiguous pages, the operating system uses its own memory allocation information to record the contiguity status to part of page table entries. The required HW changes to the TLB lookup logic is minimized by using mostly existing components in current MMUs. To record the contiguous chunk information, every N page table entries are designated as *anchor* entries. The anchors are placed on entries aligned by N. N is the distance between two adjacent anchor entries. Each anchor entry contains how many following pages are contiguously allocated, starting from the anchor entry. The anchor entry functions as a regular page table entry as well as the anchor point.

Figure 3 presents an anchored page table. In the example, the anchor distance N is 4, and thus every 4th entry is designated as an anchor entry (`A` entry type in the figure). The anchor page table entry uses unused bits to record how many following pages are contiguously mapped, as shown in Figure 4. The first anchor entry at the virtual page number 0x40 in Figure 3 has two pages that have been consecutively allocated (contiguity = 2). The contiguity counts are maintained by the operating system. If memory pages are newly allocated, relocated, or deallocated, the operating system must update the contiguity information in the corresponding anchor entry, in addition to updating the entry for the page. The anchored page table uses the same page table organization as the conventional ones, and only encodes the extra contiguity information using the unused bits in a subset of page table entries.

The anchor distance N should be determined to reflect the memory contiguity status of the process. For example, if all memory pages are allocated in 64KB chunks, the optimal anchor distance is 16 ($64KB4KB$). However, real memory allocations consist of various different chunk sizes, depending on the memory allocation behavior of the application, the OS allocation scheme, and the system memory configuration along with its current fragmentation status. We will

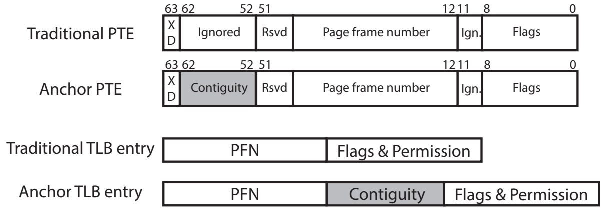


Figure 4: Page table and TLB entries for anchor TLB compared to traditional ones

discuss an OS algorithm to find the best anchor distance in Section 4. The anchor distance must be added to the context information of each process, along with the page table pointer (CR3 in x86). For every context switch, the anchor distance should be restored to the anchor distance register.

Figure 4 shows the details of page table entry used for normal and anchor entries. The anchor entry is also a regular page table entry, but it uses unused bits to store the contiguity information. To store the contiguity field in the TLB, each TLB entry may need to be increased slightly compared to the conventional ones, which do not need to store the unused bits of page table entry in the TLB.

However, in order to make anchor TLB future proof, and to provide sufficient scalability, we propose distributing the contiguity among multiple page table entries. Page table entries are always stored in groups of 8 entries per cache block of 64B. For an anchor distance of 8, the first page table entry of the cache block is the anchor of the seven other entries in the cache block. Thus 3 bits[1] are sufficient to represent an anchor distance of 8, which will fit into a single page table entry.

If we need to represent a larger anchor distance, there are still 8 bits that are usable in the page table entry. Furthermore, we can use the unused bits of multiple page table entries of the same cache block, starting from the actual anchor entry. As any anchor for anchor distances larger than 8 will always be aligned to the first page table entry of the cache block, the anchor contiguity reading logic can always read from the first entry of the cache block when reading contiguity bits distributed across page table entries in the same cache block. This allows us sufficient contiguity of up to $2^{11\times8}$ if the current physical address maximum of 2^{52}B is maintained. If the physical address maximum is bumped up to 2^{57}B (to match the virtual address space of 5-level paging), $2^{6\times8}$ contiguity is still available. This amount of contiguity is more than enough. It is worth mentioning that page table entries are fetched from the main memory in units of cache blocks, and will not result in any additional memory access to read the contiguity bits from different page table entries residing in the same cache block.

For the evaluation of this paper, we use 16 bits (maximum contiguity of 2^{16}) for the contiguity field to represent the number of contiguous 4KB pages from the anchor entry.

3.2 Translation with the Anchored Page Table

To translate with the anchored page table, only minor changes are required to the MMU. The TLB structure does not need to be modified except for a few additional bits per entry for storing the contiguity field, as both anchor and normal page table entries share the same TLB. As the L1 TLB is tightly integrated with the core, and the performance is sensitive to its access latency, the support for anchor TLB is added to the L2 TLB.

Regular Entry	Anchor Entry	Contiguity Match	Operations
Hit	.	.	Translation done
Miss	Hit	Yes	
Miss	Hit	No	Fetch the page table entry and fill in the TLB
Miss	Miss	Yes	Fetch regular entry and anchor entry. Fill only anchor entry in the TLB.
Miss	Miss	No	Fetch regular entry and anchor entry. Fill only regular entry in the TLB.

Table 2: L2 TLB operations

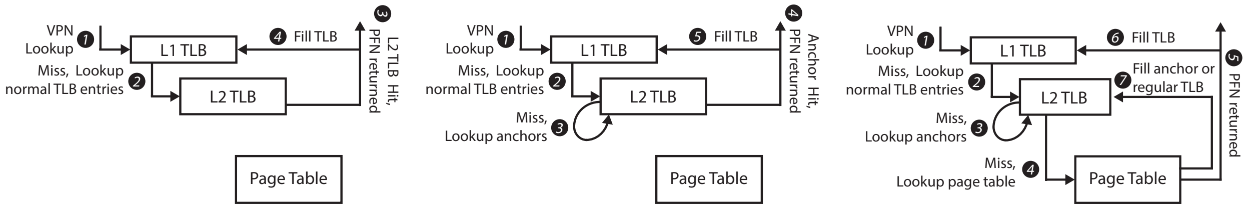
Figure 5 illustrates the TLB and page table lookup for L1 TLB misses. On an L1 TLB miss, the L2 TLB is looked up and if it is a hit, as shown in Figure 5a, the translation is completed by using the physical page number stored in the TLB entry. For an L2 TLB miss, instead of starting the conventional page table walk, the corresponding anchor entry for the VPN is looked up in the L2 TLB, and if a matching anchor entry is found, the translation is completed, as shown in Figure 5b. If an anchor entry misses or the contiguity misses, as shown in Figure 5c, a page walk is triggered. Note that the L2 TLB holds *both* regular and anchor TLB entries.

Table 2 summarizes the flow of L2 TLB operations for all cases. The first two rows represent regular TLB hit and anchor hit, respectively. The third row represents a case where the anchor lookup succeeds, but the corresponding VPN does not belong to the anchor's contiguous block, resulting in a miss. In this case, the translation information for the VPN only exists in the corresponding page table entry, requiring a page table walk followed by a regular TLB fill.

Next, the final two rows show the case where both regular TLB and anchor lookups miss, causing a page table walk. However, it is unknown whether the VPN is part of an anchor block or not until the page table walk finishes, and thus both regular TLB entry and anchor TLB entry need to be fetched. Due to the urgency of execution, the regular TLB entry is fetched first and passed onto the core ❺ and the L1 TLB ❻ as shown in Figure 5c. The anchor page table entry is then fetched and the VPN is checked to see if it belongs in the anchor block ❼. This fetch and checking step is no longer in the critical path of core execution. If the VPN belongs in the anchor block (confirmed by the `contiguity match`), the anchor TLB entry is inserted into the L2 TLB. If the VPN does not belong to the anchor block (denied by the `contiguity match`), the regular TLB entry is inserted into the L2 TLB.

Now we will discuss how anchor entries are looked up. For a given VPN to translate, the anchor virtual page number (AVPN) of the incoming VPN needs to be located. The AVPN is located at every alignment boundary of the anchor distance. For example, if the anchor distance is 4, AVPNs are located at frames 0, 4, 8, etc. Thus

[1] 3 bits will represent 0 - 7. If we make the embedded contiguity exclude the anchor entry itself, we can use the embedded value 7 to represent an anchor entry of contiguity 8.



(a) L2 TLB hit

(b) L2 TLB miss, anchor hit

(c) L2 TLB miss, anchor miss

Figure 5: L2 TLB lookup flows of regular and anchor entry lookups

locating the AVPN of an incoming VPN is calculated by aligning the VPN by the anchor distance. This calculation is executed by clearing out the $log_2 anchor\ distance$ LSB bits of the VPN which results in the AVPN. Henceforth, we will use the symbol d to denote the $log_2 anchor\ distance$, which is also denoted in Figure 6.

The indexing scheme of TLB requires a modification to store anchor entries effectively. Figure 6 represents the relation between the virtual address and VPN, and how VPN is indexed in the TLB. AVPNs have bits with zero values in `[12:d+12)` bits, as the AVPNs are aligned to 2^d pages. To ensure all consecutive AVPNs are mapped to different sets of the TLB, and thus to use all the sets of the TLB for anchor entries, `[d+12:d+12+N)` bits of the virtual address are used as the index bits, where N denotes the $log_2\# of\ sets\ in\ the\ L2\ TLB$. The bits `[12:d+12)` hold the distance between the VPN and the corresponding AVPN. This distance is compared against the contiguity value of the anchor TLB entry to decide whether the VPN is part of the anchor subblock or not. To finalize the translation on an anchor hit, the physical page number is calculated directly by adding the distance between the VPN and its anchor, $VPN - AVPN$, to the physical page number stored in the anchor entry (APPN). The final physical page number is $APPN + VPN - AVPN$.

As more pages are allocated contiguously by the anchor entry, translation requests to those pages only need to use the anchor entry, without needing to add their regular entries to the TLB. If the translation is completed successfully with the anchor entry, the actual page table entry for VPN is not fetched from the page table, preventing page walks and preventing unnecessary pollution of the TLB. In an ideal scenario, the TLB will be populated only with anchor entries, each of which can provide translation for multiple pages within its contiguity count. The translation coverage of each anchor entry is limited by the process-wide anchor distance, which is set by the operating system according to the contiguity of memory chunks allocated to the process. Therefore, anchor TLB is adaptable and scalable.

3.3 OS Implication

To support hybrid coalescing, the extra HW components are limited to the additional register to hold the anchor distance, and adders to produce the physical address from the anchor physical address. To maintain the contiguity information in anchor entries, the operating system requires modest changes.

Updating Memory Mapping: A physical memory frame can be allocated, relocated, or deallocated for a process by the operating system. Whenever the OS updates the memory allocation for a process, including the initial memory allocation for the process creation, in addition to updating the page table entries of changed pages, the related anchor entries may need to be updated, if the continuity bits in the anchor entries are affected by memory allocation changes. After updating the page table entries and anchor entries, a conventional TLB shootdown is invoked, and it invalidates both the page table entries and anchor entries from the TLBs of all cores.

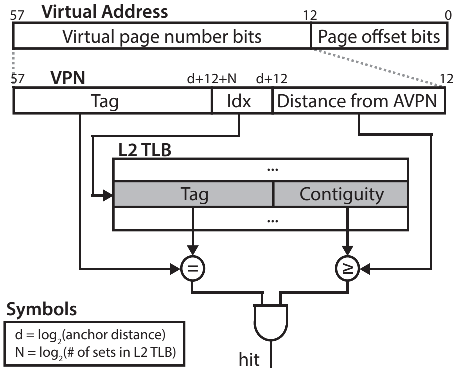


Figure 6: Anchor lookups require two comparison units to check the equality of the tag and to check whether the incoming address belongs within the anchor contiguity

Anchor Distance Change: The second change is to decide the optimal anchor distance for each process, based on its memory allocation status. The OS infrequently checks the fragmentation in allocated memory of a process, and decides the best anchor distance. Section 4 will elaborate the selection algorithm.

When the OS changes the anchor distance, those changes must be propagated through the entire page table. Changing of the anchor distance requires two costs: updating the page table, and synchronizing the TLB. The anchor entries for the new distance are updated to store proper contiguity information. During the updating process, only the page table entries that lie on the anchor distance alignment need to be changed, as the page table walker will only look at those entries when generating an anchor TLB entry. For example, anchor entries at 0, 4, 8, etc. for the anchor distance of four will be updated. Thus, when changing the distance to a larger distance, smaller number of anchor entries need to be updated, as the distance between anchors are increased. On the other hand, when the distance is changed to a smaller distance, more anchor entries need to be updated. We conducted an experiment to collect the overhead of changing the anchor distance. The cost of sweeping through th entire page table when the process uses 30GB of memory is 452ms, 71.7ms, 1.7ms for changing the anchor distances to 8, 64, and 512, respectively. When a process is created, the default anchor distance can be set to any number chosen by the OS, and it changes to the selected anchor distance once sufficient amount of memory is allocated. If the memory mapping of an application frequently changes during the execution and results in different optimal anchor distances, the OS can make a decision based on the cost and benefit of changing the anchor distance.

The second cost is synchronizing the TLB after updating the page table. In our work, we will update the entire page table, and then we will invalidate the entire TLB. Considering the fact that the native Linux kernel for x86 flushes the TLB on context switches, the cost of invalidating the TLB can be relatively minor. Also, in this work we assume the memory map change is checked in periodic epochs of one billion instructions. Even with the periodic checks, the actual anchor distance change is rare, as different epochs still use the same level of memory chunk allocation.

We have found that the anchor distance change to be rarely executed. For specific applications or situations where the memory mapping changes dramatically, the OS can set the limit on how often it can be invoked. In the next section, we will show how the distance is selected.

Permission and Page Sharing: Even if the address mapping is contiguous, pages may have different r/w/x permissions. Hybrid coalescing can support any fine-grained permission, by simply treating a page with a different permission as the non-contiguous page. The translation for the page must not be handled by the anchor entry. Although such fine-grained permission differences reduce the effectiveness of any TLB coalescing techniques, the prior work has shown that permissions are commonly homogeneous in much larger granularities[5], and the actual performance impact is expected to be minor in common applications. For page sharing across processes, the contiguity is set in the page table of each process. Therefore, the contiguity in shared regions can also be exploited, although its effectiveness may vary by the anchor distance selected for each process.

4 DYNAMIC ANCHOR DISTANCE

In this section, we propose a dynamic anchor distance selection algorithm to find the best density of anchor entries in the page table. The per-process anchor distance is determined based on the distribution of contiguous memory chunks allocated to the process. At the beginning of a process execution, only a small amount of memory is allocated, but many processes allocate the majority of memory they use for the rest of execution in the early phase of execution, as shown by Basu et al. [5]. Once the initial memory allocation phase is stabilized, the anchor distance can be selected from the chunk distribution.

However, memory mappings can change even during the execution of a process, as the process itself may dynamically allocate and deallocate memory causing changes in the memory mappings. Furthermore, the operating system may reorganize the memory mappings to optimize the performance. The Linux kernel may try compacting memory as an effort to create more large pages for the process [24, 34]. Operating systems may also promote pages into a super page when sufficient reserved pages have been touched [29]. Large pages may be demoted by the operating system when pages of the large pages are unmapped, or even initiated by tools that optimize the system for NUMA-ness [17].

Algorithm 1 Dynamic distance selection algorithm

```
// contiguity_histogram is a list of (contiguity, frequency) pairs

// List of available anchor distances in the system
Distances ← [2, 4, 8, ... $2^{16}$]

// Costs for different anchor distances
$\forall d \in Distances$    $cost_d$ ← 0

for each d in Distances do
    // Calculate distance cost for each contiguity of histogram
    for each (cont,freq) in contiguity_histogram do
        anchors ← cont/ anch_dist × freq
        large_pgs ← remainder / 512 × freq
        pages ← remainder × freq

        // Weigh down costs of entries with larger coverage
        $cost_d$ + = anchors /anch_dist
        $cost_d$ + = large_pgs/512
        $cost_d$ + = pages
    end for
end for

// Pick anchor distance with min cost
min_dist ← $\min_{\forall d \in Distances}(cost_d)$
setProcAnchorDistance(min_dist) // Set distance of the process
```

As the memory mapping of any running application may change dynamically, and each execution of the same application can result in different mappings, the operating system needs to provide a means of setting the appropriate anchor distance based on the information available in the OS. In the proposed scheme, the operating system periodically checks the chunk distribution of each process, and recalculates the optimal anchor distance. If the new anchor distance is sufficiently different from the current one, the anchor distance is updated, incurring costly operations of anchor distance change.

4.1 Selection Process Overview

The main aim of the selection algorithm is to minimize the number of TLB entries (anchor, large page, and 4KB page entries) required to provide coverage for the active pages mapped to a process. To assess the memory contiguity status, the OS maintains a histogram of contiguity distribution. The contiguity histogram holds how many contiguous memory chunks of varying contiguity are allocated to the process.

Using the contiguity histogram, the capacity cost of storing the page mapping in TLBs is estimated by a heuristic approximation. An anchor distance, which minimizes the capacity cost, will be selected as the best anchor distance. Note that in this method, the frequency of page accesses is not considered, as accurately collecting such information is costly in the current systems. The anchor distance selection process only uses the static memory mapping information summarized as the contiguity histogram. Our results show that this static estimation can provide a reasonable accuracy for finding the best anchor distance. The algorithm is shown in Algorithm 1 and described in the following paragraphs.

Heuristic Selection: The OS maintains the contiguity histogram to reflect the contiguity of memory pages of a process. Each entry of the contiguity histogram will hold two values: `contiguity` and `frequency`. Contiguity represents the size of chunk, and frequency represents how many chunks of the corresponding contiguity have been allocated. Using the contiguity histogram, the OS estimates how many TLB entries are required to cover the entire memory footprint of the process. For every possible anchor distance value, the OS goes through the contiguity histogram and assembles a *cost* for each anchor distance.

The high level description of the Algorithm 1 is as follows. To calculate the cost per anchor distance, the number of required hypothetical TLB entries is approximated. Once the costs have been calculated for all anchor distances, the distance with the smallest cost is selected.

The counts of required TLB entries are separately counted for 4KB pages, 2MB large pages, and anchor regions. For example, an anchor distance of 16 allows a single anchor entry to cover the size of 64KB (16 × 4KB). In such a case, a contiguous chunk of 64KB memory will require a single anchor TLB entry, while a 128KB chunk of memory requires two anchor TLB entries. If there are any remaining pages in the chunk after the coverage of anchor entries is deducted from the chunk, the number of required 2MB pages is calculated. The remaining pages after 2MB page deduction must be covered with 4KB pages. Once the required numbers of entries for different types of pages have been calculated, the total cost is obtained from the weighted sum of different types. The weight is the inverse of the coverage of each type.

Distance Stability: Stability of the distance selection algorithm is important, as changing the anchor distance is a costly task, as described in Section 3.3. The distance selection algorithm is executed periodically to adapt to any significant memory mapping distribution changes. Although our dynamic selection mechanism periodically checks the best anchor distance, the best anchor distance for a process does rarely change, as once a large amount of memory pages allocated to a process, their chunk distribution does not change significantly for the rest of execution.

In our simulations, we executed the distance selection algorithm every one billion instructions. From the execution based on real machine traces, the distance selection algorithm did not make any changes after making the initial selection decision. Our proposed simple algorithm provides stability of anchor distance selection, preventing any frequent distance changes that may cause significant overheads.

4.2 Discussion & Future work

The dynamic distance selection scheme that we have introduced implicitly assumes that the entire address space of a running process has a single clusterable distance. However, an address space has different semantic memory regions: code, data, shared libs., heap and stack. Different regions may have different contiguity. Also, even within the same semantic region, the contiguity distribution may be different, as the OS may have a different memory condition during the execution of the process.

Schemes	TLB Configuration
Common L1	4KB: 64 entry, 4 way 2MB: 32 entry, 4 way
Baseline/THP	4KB/2MB(shared) 1024 entry, 8 way
Cluster	Regular TLB: 768 entry, 6 way Cluster-8: 320 entry, 5 way
RMM	Baseline L2 TLB RMM: 32 entry, fully associative
Anchor	4KB/2MB/Anchor(shared): 1024 entry, 8 way
Latencies	7 cycle L2 hit latency [18] 8 cycle clust./RMM/anch. hit lat. 50 cycle page table walk lat. [22]

Table 3: TLB configuration used for evaluation

Scenario	Contiguity
`low contiguity`	1 - 16 pages (4KB - 64KB)
`medium contiguity`	1 - 512 pages (4KB - 2MB)
`high contiguity`	512 - 65,536 pages (2MB - 256MB)
`max contiguity`	maximum

Table 4: Synthetic mapping scenarios

Thus, to further improve the performance of the anchor TLB, *region* may be introduced. A region is part of virtual address space with a separate anchor distance optimized for the region. An address translation for a given region uses the region-specific anchor distance. To support such a multi-region anchor TLB, an additional hardware must hold multiple region definitions consisting of the starting VPN, ending VPN, and anchor distance for each region. The additional HW component is similar to the range TLB structure in RMM. The *region table* will be looked up in parallel with the 4KB/2MB TLB lookup. Since all the regions must be searched in parallel for fast accesses, the number of regions is limited. If the TLB lookup misses, the anchor distance from the matching region is used to lookup the anchor entry in the L2 TLB. The dynamic distance selection can be extended to partition the memory into different regions if there are contiguity variances. We believe that such approach would further improve the anchor efficiency, and we leave this as future work.

5 EVALUATION

5.1 Methodology

We simulated our work on a trace-based simulator that models the cache and TLB structures. The configuration of the HW components are shown in the Table 3. We executed benchmarks from the `SPECCPU2006`, `biobench` suites, and additionally `graph500` and `gups`. The working set sizes of `graph500` and `gups` are set to 8GB. To generate the trace of each application, we used the Pin binary instrumentation tool [27] and generated a memory access trace of 12 billion instructions. At the same time, at every billionth instruction boundary, we periodically captured the virtual to physical memory address mapping on the real machine, using the `pagemap` [11] interface provided by Linux.

For the evaluation, we show the performance of our system using a total of 6 mapping scenarios: two mappings were captured from

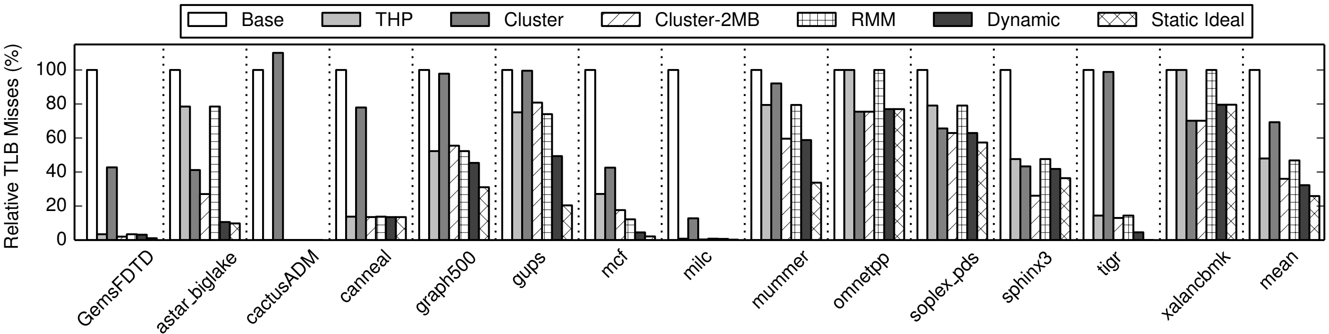


Figure 7: Relative TLB misses for `demand paging` **mapping**

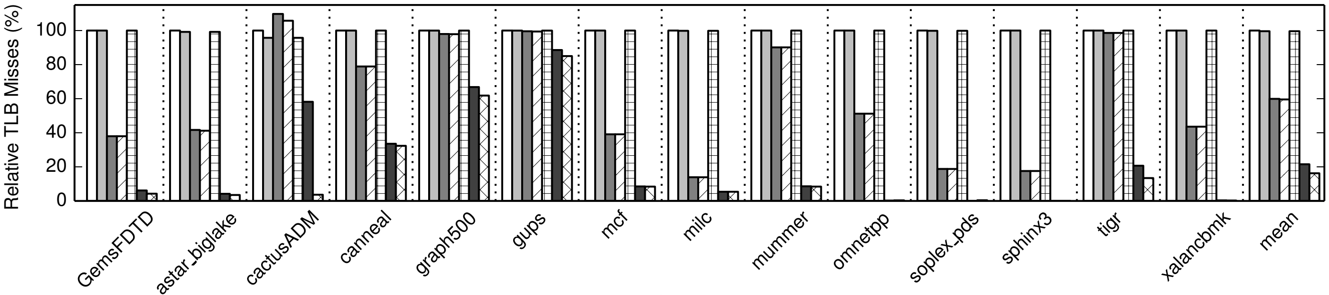


Figure 8: Relative TLB misses for `medium contiguity` **mapping**

actual system executions, and four mappings were synthetically generated.

Real Mappings: we generated a trace on a vanilla Linux 3.18.29 machine, which uses `demand paging`. With demand paging, memory pages are not allocated until actual accesses (via page fault) occur. This approach is the default behavior currently used. It minimizes unused pages, but the contiguity of the page mappings can be potentially reduced. We also generated a trace when the system uses `eager paging`, which allocates memory pages immediately upon a memory allocation request from the process. We modified the Linux kernel to map pages for the eager allocation. However, our eager paging implementation does not directly allocate large chunks of contiguous space, but rather requests pages through the buddy allocator system sequentially. The use of eager paging increases the mapping contiguity, compared to the contiguity of demand paging. Linux transparent huge page support was enabled when generating traces of both real mappings.

Synthetic Mappings: we generated four synthetic mappings: `low contiguity`, `medium contiguity`, `high contiguity`, and `maximum contiguity`. Table 4 shows the contiguity distributions for the mappings. For each mapping, chunk sizes are selected from the given range with a uniform random distribution. The `low contiguity` and `medium contiguity` represent the scenarios which cannot provide large chunks. `maximum contiguity` is an extreme contiguous mapping where every contiguous virtual address region is mapped to the same amount of contiguous physical region. This is an ideal mapping for segment-based translations such as RMM. The synthetic mappings are used to exercise the proposed scheme and prior ones with various allocation scenarios, revealing the advantages and disadvantages of the approaches under different allocation situations.

Comparison: We compare our work to `THP` (Transparent Huge Pages [12]), an implementation of large page in Linux, cluster TLB (`cluster`) [33], and `RMM` [21]. Cluster TLB requires a partitioned TLB with regular and cluster entries. However, our baseline TLB is larger than that used by Pham et al. [33]. To scale the TLB, we increased both of the numbers of cluster and regular entries proportionally, while keeping the set-way settings reasonable. The original cluster TLB does not use the 2MB large page, and for fair comparison, we also evaluate `cluster-2MB` which can use the large page in the regular TLB entries in addition to clustering at 4KB. `RMM` requires an additional fully associative range TLB. We employed a 32 entry range TLB as used by Karakostas et al. [21], in addition to the 1024 entry baseline TLB. The 1024 baseline TLB with `RMM` supports both 4KB and 2MB pages.

We show the performance of our method in two schemes: `dynamic` and `static ideal`. In `dynamic`, the proposed dynamic distance selection algorithm is used to pick the anchor distance. In `static ideal`, we show the results with one optimal distance which performs best for each application and mapping, by exhaustive evaluation of all possible distances. The `static ideal` scheme is used to illustrate the near maximum TLB miss reductions achievable by the anchor selection algorithm.

TLB Parameters: Table 3 describes the TLB parameters for the prior schemes and hybrid coalescing. The L1 TLB configuration is the same for all schemes. The L2 TLB capacity is set to 1024 entries. However, `cluster` partitions it to the regular and cluster entries. In addition to the L2 TLB, `RMM` has the 32-entry fully associative range TLB. The latency parameters are derived from those used by Karakostas et al. [22].

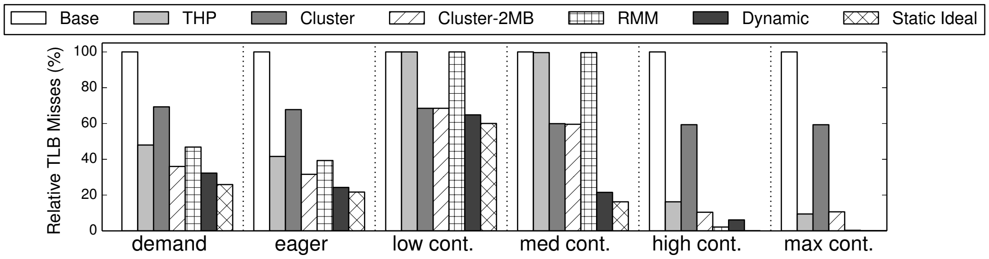


Figure 9: Average TLB misses of each translation scheme for all mapping scenarios

5.2 Results

We present the evaluation results of all benchmarks for demand paging in the real mapping and medium contiguity in the synthetic mapping. For the rest of mapping scenarios, we present the mean TLB reductions due to the space constraint.

5.2.1 TLB Misses for Demand Paging and Medium Contiguity. Figure 7 shows the relative TLB misses with demand paging, normalized to those with the baseline configuration. With the transparent huge page support turned on, the mapping of demand paging contains many 2MB contiguous chunks. Therefore, all the techniques other than baseline and cluster which do not use the large page, benefit from the large page allocation. Across all the applications except for omnetpp and xalancbmk, THP alone can reduce the TLB misses effectively by average 60% for all the other applications. Although the original cluster reduces TLB misses effectively for mcf and milc significantly, their reductions are limited for some applications (canneal, gups, mummer, and tigr) without support of the large page. However, cluster-2MB can benefit both from HW coalescing and large page, achieving 64% reduction on average. RMM is also effective for the majority of applications, and their reductions are on average 53.2% which are similar to THP. However, both THP and RMM are not very effective for omnetpp and xalancbmk, as those applications do not exhibit large chunk contiguity. For the two applications, HW coalescing of cluster and cluster-2MB effectively reduces TLB misses among the prior schemes.

Unlike the prior schemes, hybrid coalescing can consistently reduce TLB misses, by exploiting both large page contiguity and fine-grained coalescing. With demand paging, the proposed dynamic hybrid coalescing can provide the reduction of 67.3% on average, which is better than the best prior scheme (cluster) for the mapping.

Figure 8 plots the evaluation on the medium contiguity mapping. Unlike demand paging, this mapping has contiguity mostly less than 2MB page, and thus THP is ineffective. RMM also shows similar results to THP, due to the lack of high contiguity. For this fine-grained contiguity existing in the mapping, cluster and cluster-2MB reduce misses effectively for many applications, but they are not very effective for graph500, gups, and tigr. For cactusADM, the misses get worse due to the statically partitioned TLB between regular and clustered entries. In cactusADM, the clustered TLB entries are underutilized, while the regular entries are full.

The proposed hybrid coalescing can effectively exploit the available contiguity, excelling the other schemes. It can effectively utilize the medium contiguity of less than 2MB, which cannot be exploited by THP and RMM. At the same time, hybrid coalescing has higher coverage than cluster and cluster-2MB. Even for the worst case gups application, it can reduce TLB misses by 11.4%. The difference between dynamic and static-ideal is relatively small. However, for cactusADM, the dynamic algorithm does not find the best optimal distance, although dynamic still reduces TLB misses significantly. It is due to the static nature of the dynamic algorithm which finds the distance based on the allocation snapshot, without knowing access frequency.

	demand			medium-contiguity		
	R.hit	A.hit	L2 miss	R.hit	A.hit	L2 miss
astar	43 %	49 %	6 %	52 %	46 %	2 %
cactus	49 %	51 %	0 %	11 %	44 %	45 %
canne.	33 %	55 %	12 %	25 %	59 %	16 %
GemsF.	91 %	8 %	1 %	13 %	85 %	2 %
mcf	91 %	8 %	1 %	66 %	32 %	2 %
milc	74 %	25 %	1 %	3 %	92 %	5 %
omnet.	48 %	29 %	23 %	62 %	38 %	0 %
soplex	75 %	12 %	13 %	57 %	43 %	0 %
sphinx	87 %	3 %	10 %	53 %	47 %	0 %
xalan.	18 %	16 %	66 %	66 %	34 %	0 %
mummer	39 %	5 %	56 %	70 %	22 %	8 %
tigr	61 %	34 %	5 %	61 %	22 %	17 %
gups	27 %	20 %	53 %	11 %	1 %	88 %
graph.	49 %	5 %	46 %	29 %	5 %	66 %

Table 5: L2 TLB hit/miss statistics. The regular L2 TLB hit rate (R.hit), anchor TLB hit rate (A.hit), and L2 TLB miss rate are shown.

Table 5 shows the access breakdowns at the L2 TLB for the demand paging and medium contiguity mappings. As discussed with Figure 7, the 2MB large page is very effective for demand paging, and thus many L2 TLB accesses hit on the regular L2 TLB entries containing large pages. For applications with relative low regular hit rates, such as canneal, xalancbmk, and gups, anchor entries additionally absorb 16-55% of TLB accesses. In medium contiguity, the regular entry hit rates are much lower than those in demand paging, due to the lack of large page accesses. In such cases, anchor entries can be very effective in reducing TLB misses.

5.2.2 TLB Miss Summary of All Mapping Scenarios. Figure 9 summarizes the TLB miss reductions for all six mappings. Demand paging and eager paging provide both the 2MB large page and fine-grained coalescing chances depending on applications, with a

	demand	eager	low	medium	high	max
astar_biglake	16	256	4	16	128	256
cactusADM	4K	8K	4	32	256	512
canneal	1K	512	4	8	256	1K
GemsFDTD	8K	8K	4	32	256	1K
mcf	64K	64K	4	32	512	64K
milc	16K	8K	4	32	256	256
omnetpp	4	4	4	16	128	256
soplex_pds	2	2	4	16	64	64
sphinx3	4	4	4	32	32	32
xalancbmk	4	4	4	32	128	128
mummer	2K	32K	4	32	128	256
tigr	2K	512	4	32	256	512
gups	32K	32K	4	32	1K	64K
graph500	64K	16K	4	32	1K	64K

Table 6: Anchor distances in pages, selected by the dynamic distance selection algorithm.

higher overall benefit from the 2MB large page. Effectively utilizing the two opportunities, cluster-2MB can provide the best reduction among the prior schemes, reducing TLB misses by 64% and 68.4% on average for demand and eager mappings, respectively. The proposed hybrid coalescing further reduces TLB misses benefiting from a higher coalescing capability than cluster-2MB, achieving 67.7% and 75.7% reduction.

The low and medium continuity mappings provide few 2MB or larger chunks, and thus only the prior HW clustering and the proposed hybrid coalescing can reduce TLB misses. However, with better coalescing scalability, the hybrid coalescing can reduce TLB misses by 35.2% and 78.5%, compared to 31.5% and 40.4% of cluster-2MB in low and medium mappings. For these mappings, THP and RMM are nearly ineffective. The high and maximum contiguity mappings provide large contiguous chunks which help both THP and RMM. RMM, with better coverage scalability, almost eliminates TLB misses. Even in these cases, the proposed hybrid coalescing almost matches the miss reductions of RMM.

From the results, we conclude that our scheme outperforms or performs similar to the best prior scheme for each mapping scenario, achieving the best average performance across diverse scenarios. It can adjust the coalescing capability dynamically to match various allocation contiguity distributions. Such dynamic adjustment allows it to extract whatever available contiguity as efficiently as possible.

5.2.3 Anchor Distance Selection. Table 6 shows the anchor distances selected by the dynamic selection algorithm for different mapping scenarios. The values shown are the number of consecutive pages. For demand paging and eager paging, the contiguity distributions are highly skewed for many applications with a small number of very large contiguous chunks in memory allocation. As these chunks tend to dominate the memory footprint, the selection algorithm chose large distances. Note that 64K is the largest anchor distance used for the evaluation. However, for a few applications, such as omnetpp and xalancbmk, a small distance of 4 pages are selected. As shown in Figure 7, the two applications have a fine-grained contiguity which can only be coalesced by cluster, cluster-2MB, and the proposed scheme.

However, for synthetic mappings, the chunk distributions are uniform. For the chunk distribution of each configuration, the heuristic algorithm finds a reasonably good distance for each range, with four in low contiguity to 32-1K in high contiguity.

5.2.4 Translation CPI. Figures 10 and 11 show the breakdown of the cycles spent per instruction in the address translation. The L1 TLB will be accessed in parallel with the cache access and so the L1 TLB access latency is hidden [22]. We estimate the CPIs based on the latencies in Table 3. The L2 hit portion denotes CPIs spent for regular L2 TLB hits. The next CPI portions are specific to each technique: cycles spent for anchor hits, cluster TLB hits, range TLB hits for hybrid coalescing, cluster, and RMM, respectively. Firstly, the CPI results are consistent with the TLB miss analysis in Figures 7 and 8, although the actual CPI changes are affected by TLB misses per instruction. The proposed hybrid coalescing is better than or almost match the best prior scheme for each mapping scenario. For applications with high TLB miss rates in the baseline configuration, the performance improvements of the proposed scheme are significant, with the CPI reduction of 0.85, 2.7, and 5.82 for gups, tigr and graph500, respectively, for the demand paging mappings. In the case of graph500 executing on the medium contiguity mapping, up to 3.51 CPI reduction is expected.

6 RELATED WORK

There have been many prior work to improve different aspects of address translation to support virtual memory.

Improving TLB Coverage: Improving the TLB coverage has been a topic of research over the years [5, 13, 21, 22, 30, 31, 33, 34, 40]. Talluri and Hill proposed subblocking of TLB entries that allows a single TLB entry to represent multiple page mappings [38] within a subblock of pages. Subblocking has influenced this work and other work, including CoLT [34] and Cluster TLB [33]. In multi-core systems, data sharing across multiple cores is exploited to reduce TLB misses by collaboratively using shared translation entries [9, 25, 37]. Papadopoulou et al. proposed a prediction-based indexing approach to support multiple page size with a single lookup when predictions hit [30]. Recently, Cox et al. proposed Mix TLB supporting multiple page sizes using a single indexing scheme [10]. With a single indexing for 4KB page, it intentionally allows multiple TLB entries of a large page to exist in the TLB. However, based on the observation that even superpages tend to be allocated consecutively by the OS, HW coalescing of superpage entries offsets the capacity waste caused by multiple TLB entries of the same large page.

Reducing TLB Miss Penalty and Prefetching: There are many parts of the page table walker that can be optimized to minimize the TLB miss penalty. The first set of studies reduces the page table walk overhead by improving the translation cache [3, 8], which minimizes the number of memory accesses to fetch intermediate page table nodes. Speculation can also be used to speed up the page table walking procedure [4]. By using the reservation scheme [29], it is possible to accurately interpolate the missing address by using the data that is available. Finally, prefetching can be used to proactively insert pages that may be used in the near future into the TLB [19, 36].

A different approach to reduce address translation is virtual caching. Virtual caching allows to defer address translation after cache misses. Several prior work on virtual caching save TLB

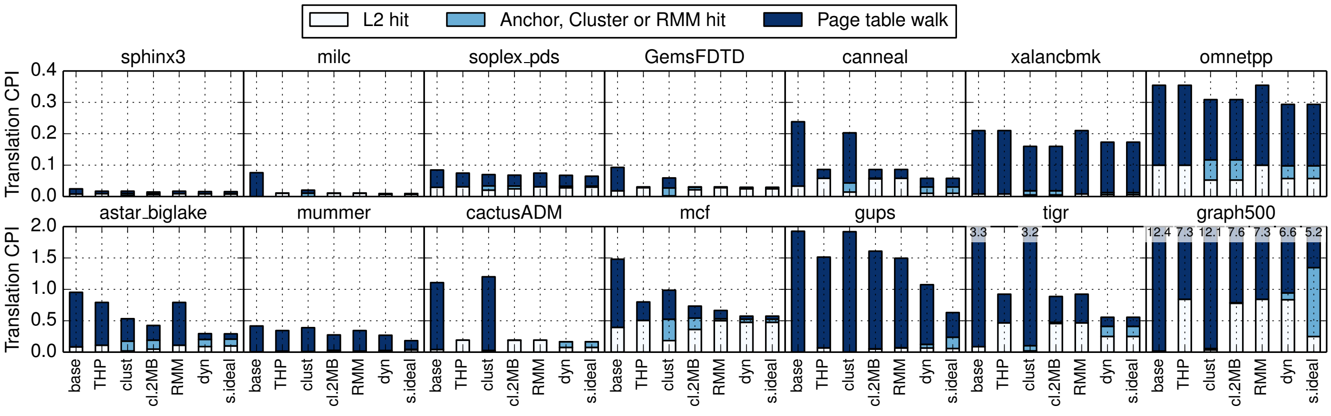


Figure 10: CPI breakdown of translation overhead for `demand paging`

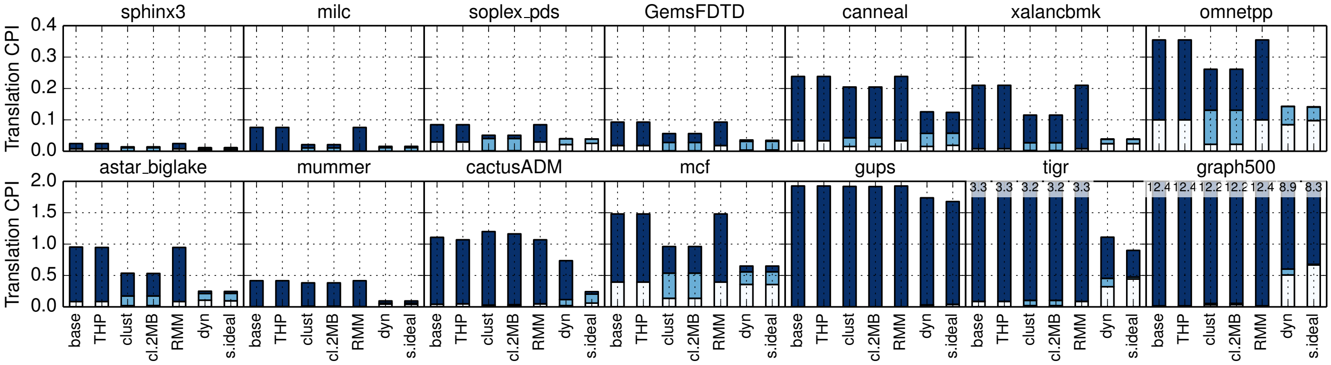


Figure 11: CPI breakdown of translation overhead for `medium contiguity`

power consumption by reducing TLB accesses significantly, or reduce page table walks as large on-chip caches can contain data which could have missed in TLBs of the conventional physical caching [6, 31, 39, 40].

Virtualized Systems: As virtual machines add an additional layer of address translation, TLB miss latencies are amplified, and thus the virtualized systems exhibit more severe performance drops by TLB misses, compared to native ones. Various prior work have tackled the translation challenge of virtualization. Gandhi et al. extended the segment-based translation to support nested translation of virtualized systems [15]. On the other axis, there were studies that improve the TLB miss latency handling by improving the translation cache [7] or improving the organization of the nested page tables [2, 16].

7 CONCLUSION

This paper proposed a novel HW-SW hybrid TLB coalescing technique. By encoding allocation contiguity information in page tables, highly coalesced address translation was possible, only with minor changes to the existing MMU designs. The hybrid approach not only provides high coverage scalability by using the OS support, but also allows adaptability to diverse memory fragmentation status. With various memory mapping scenarios, our experimental evaluation showed that the proposed scheme can provide translation performance better than or matching the best prior scheme for each scenario.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF-2016R1A2B4013352) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants were funded by the Ministry of Science, ICT and future Planning (MSIP), Korea.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[2] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 476–487. https://doi.org/10.1109/ISCA.2012.6237041

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[6] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th*

Annual International Symposium on Computer Architecture (ISCA '12). IEEE Computer Society, Washington, DC, USA, 297–308. https://doi.org/10.1145/2366231.2337194

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[8] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[10] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[11] Linux Kernel Documentation. 2016. pagemap, from the userspace perspective. (28 May 2016). Retrieved April 26, 2017 from https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[12] Linux Kernel Documentation. 2017. Transparent Hugepage Support. (05 Mar 2017). Retrieved April 26, 2017 from https://www.kernel.org/doc/Documentation/vm/transhuge.txt

[13] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. 2015. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. IEEE, 223–234. https://doi.org/10.1109/HPCA.2015.7056035

[14] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[15] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 178–189. https://doi.org/10.1109/MICRO.2014.37

[16] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 707–718. https://doi.org/10.1109/ISCA.2016.67

[17] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242.

[18] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[19] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. IEEE Computer Society, Washington, DC, USA, 195–206. https://doi.org/10.1109/ISCA.2002.1003578

[20] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 13, 16 pages. https://doi.org/10.1145/2901318.2901325

[21] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[22] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*. IEEE Press, 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[23] Gwangsun Kim, John Kim, Jung H. Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with Hybrid Memory Cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 145–155. https://doi.org/10.1109/PACT.2013.6618812

[24] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721.

[25] Yang Li, Rami Melhem, and Alex K. Jones. 2012. Leveraging Sharing in Second Level Translation-Lookaside Buffers for Chip Multiprocessors. *IEEE Computer Architecture Letters* 11, 2 (2012), 49–52. https://doi.org/10.1109/L-CA.2011.35

[26] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *2008 International Symposium on Computer Architecture*. 453–464. https://doi.org/10.1109/ISCA.2008.15

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[28] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[29] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36, SI, 89–104. https://doi.org/10.1145/844128.844138

[30] Misel M. Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. 210–222. https://doi.org/10.1109/HPCA.2015.7056034

[31] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2016. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 217–229. https://doi.org/10.1109/ISCA.2016.28

[32] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[33] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[34] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[35] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[36] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/339647.339666

[37] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*. 313–324. https://doi.org/10.1109/MICRO.2010.26

[38] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[39] Hongil Yoon and Gurindar S. Sohi. 2016. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*. 212–224. https://doi.org/10.1109/HPCA.2016.7446066

[40] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1810085.1810109