CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–35
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Survey of Techniques for Architecting TLBs

Sparsh Mittal*

Indian Institute of Technology, Hyderabad

SUMMARY

"Translation lookaside buffer" (TLB) caches virtual to physical address translation information and is used in systems ranging from embedded devices to high-end servers. Since TLB is accessed very frequently and a TLB miss is extremely costly, prudent management of TLB is important for improving performance and energy efficiency of processors. In this paper, we present a survey of techniques for architecting and managing TLBs. We characterize the techniques across several dimensions to highlight their similarities and distinctions. We believe that this paper will be useful for chip designers, computer architects and system engineers. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Review; classification; TLB; superpage; prefetching; power management; virtual cache; workload characterization

†

1. INTRODUCTION

Virtual to physical address translation is a crucial operation in computing systems since it is performed on every memory operation in processors with physical caches and on every cache miss in processors with virtual caches. To leverage the memory access locality, processors with page-based virtual memory use TLB to cache the recent translations. Due to this, TLB management becomes crucial for improving the efficiency of processors. TLB miss requires costly page walks, e.g., in a 4-level page table design, a TLB miss may need up to 4 memory accesses [1]. Due to this, applications may spend a large fraction (e.g., up to 50% [2, 3]) of time in TLB miss handling.

To minimize TLB miss rate and latency, a careful choice of design parameters is required, such as TLB size, associativity (direct-mapped, SA† or FA) and number of levels (one or two), however, these are known to present strict tradeoffs. These challenges are compounded by other design options, e.g., use of a single base page, superpage or multiple page size(s), private or shared TLB(s) in multicore processors, prefetching, etc. Although the performance impact of TLB can be reduced by using virtual caches [4], they present challenges of their own. It is clear that intelligent techniques

*Correspondence to: Sparsh Mittal, IIT Hyderabad, Kandi, Sangareddy 502285, Telangana, India. email: sparsh0mittal@gmail.com.
†This is author's version of the article accepted in "Concurrency and Computation: Practice and Experience" journal. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Self-Archiving.
†Following acronyms are used frequently in this paper: "address space ID" (ASID), "data TLB" (DTLB), "first-in first-out" (FIFO), "fully-associative" (FA), "input/output" (IO), "instruction set architecture" (ISA), "instruction TLB" (ITLB), "java virtual machine" (JVM), "just-in-time" (JIT), "least recently used" (LRU), "memory management unit" (MMU), "page table entry" (PTE), "page table walker" (PTW), "physical address" (PA), "physical page number" (PPN), "program counter" (PC), "set-associative" (SA), "spin transfer torque RAM" (STT-RAM), "virtual address" (VA), "virtual memory" (VM), "virtual page number" (VPN), "virtual to physical address" (V2PA), "working set size" (WSS). In this paper, unless specifically mentioned as 'L1 cache', 'L1 data cache' or 'L1$' etc., L1 and L2 refer to L1 TLB and L2 TLB, respectively.

are required to optimize the design and management of TLBs. Recently, several techniques have been proposed for addressing these issues.

Contributions: This paper presents a survey of techniques for designing and managing TLBs. Figure 1 presents the organization of the paper. We first discuss the tradeoffs and roadblocks in managing TLBs and provide a classification and overview of research works on TLBs (§2). We then discuss studies on TLB architecture exploration and workload characterization (§3). Further, we present techniques for improving TLB coverage and performance, e.g., superpaging, prefetching, etc. (§4). We also discuss techniques for saving TLB energy (§5), including virtual cache designs (§6). Furthermore, we review TLB design proposals for GPUs and CPU-GPU systems (§7). We conclude the paper with a brief discussion of future challenges (§8).

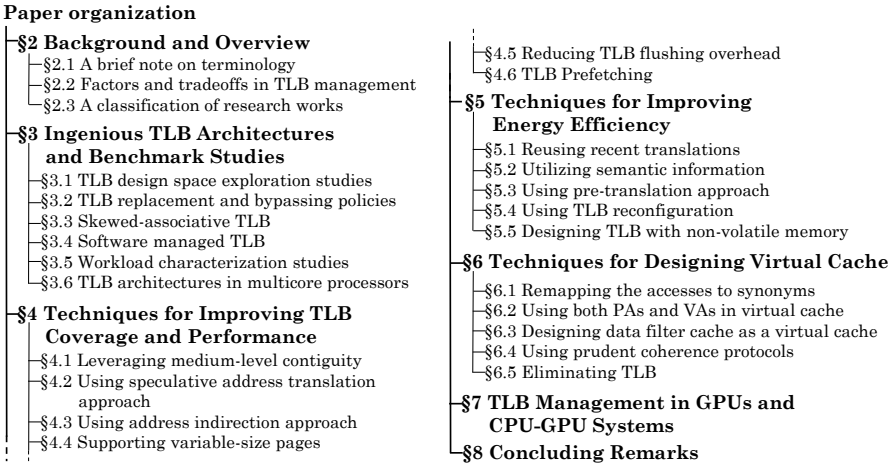


Figure 1. Organization of the paper

Scope: TLB management has a significant overlap with related areas, e.g., virtual memory, page table management, cache management; however for sake of conciseness, this paper only includes works focused on TLB improvement. We discuss techniques proposed for CPUs, GPUs and CPU-GPU heterogeneous systems. We include architecture and system-level techniques, and not circuit or application-level techniques. We focus on the key insights of the papers and only include selected quantitative results to show scope of improvement. We believe that this paper will be valuable for researchers and computer architects.

2. BACKGROUND AND OVERVIEW

We first discuss terms and concepts that will be useful throughout this paper (§2.1). We then discuss the challenges involved in design and optimization of TLBs (§2.2). We finally categorize research works along several dimensions to provide an overview of the field (§2.3). For details on TLB configurations in commercial processors, we refer the reader to previous work [2, 5–12]. Note that "address translation cache" and "directory lookaside table" are other infrequently used terms for referring to TLB [2].

2.1. Background and terminology

Coverage: Coverage (or reach or mapping size [13]) of TLB shows the sum of the memory mapped by all its entries. Clearly, the coverage needs to be high for applications with large working set. For example, Peng et al. [3] note that only few (e.g., 4) PCs are responsible for a large (e.g., >70%) fraction of misses and these misses can be lowered by enhancing TLB coverage.

Superpage: A superpage refers to a VM page with size and alignment which is power of two times the system page size (e.g., 4KB or 8KB) and maps to contiguous physical pages. Superpage

DOI: 10.1002/cpe

uses a single TLB entry for multiple adjacent virtual pages which improves TLB coverage and reduces misses. Thus, superpages are useful for mapping large objects, e.g., big arrays, kernel data, etc. and are especially useful for coping with increasing working sets of modern applications.

TLB shootdown: On events such as page swaps, privilege alterations, context switches and page remapping, etc., V2PA mapping changes. This necessitates invalidating the TLB entries whose translation mapping has changed and this is referred to as TLB shootdown. In multicore processors, invalidation message needs to be sent to all per-core TLBs that hold changed entries and hence, shootdown leads to scalability bottleneck [14].

HW- and SW-managed TLBs: Since TLB hit/miss determination happens on critical path, it is always implemented in hardware. By contrast, handling of a TLB miss may be implemented in hardware or software.

In a *HW-managed TLB*, on a TLB miss, a HW state machine performs page table walk to find a valid PTE for the given VA. If the PTE exists, it is inserted into TLB and the application proceeds normally by using this PTE. If a valid PTE is not found, the hardware raises a page fault exception which is handled by the OS. The OS brings the requested data into memory, sets up a PTE to map the requested VA to the correct PA and resumes the application. Thus, in a HW-managed TLB, the TLB entry structure is transparent to the SW which allows using different processors while still maintaining software compatibility. HW-managed TLBs are used in x86 architecture [15].

In a *SW-managed TLB*, on a TLB miss, the hardware raises a 'TLB miss' exception and jumps to a trap handler. The trap handler is a code in the OS which handles TLB miss by initiating a page table walk and performing translation in SW. A page fault, if any, is also handled by the OS, similar to the case in HW-managed TLB. Processors with SW-managed TLBs have specific instructions in their ISA that allow loading PTEs in any TLB slot. The TLB entry format is specified by the ISA. SW-management of TLB simplifies HW design and allows OS to use any data structure to implement page table without requiring a change in HW. However, SW-management also incurs larger penalty, e.g., due to pipeline flushing. MIPS and SPARC architectures generally use SW-managed TLBs [15].

Cache addressing options: For L1 cache lookup, index and tag can be obtained from VA or PA and hence, there are four possible addressing options [4, 9, 16], namely physical/virtual indexed, physical/virtual tagged (PI/VI-PT/VT) which are illustrated in Figure 2. Notice that PI-PT, VI-PT and PI-VT designs require a TLB access for all memory operations. We now discuss these addressing options.

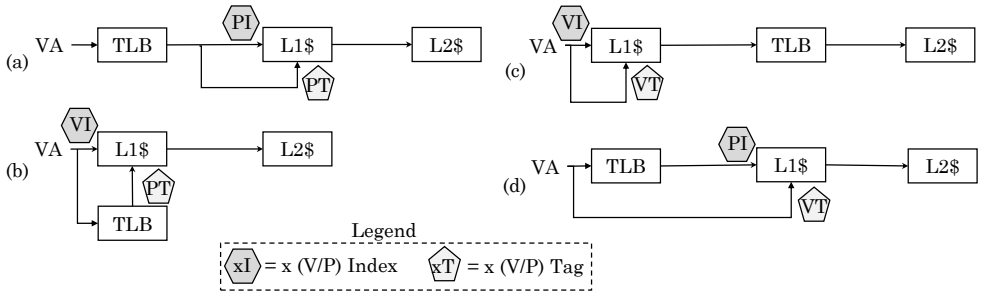


Figure 2. Four cache addressing options: (a) PI-PT, (b) VI-PT, (c) VI-VT and (d) PI-VT.

(a) PI-PT: Before L1 cache lookup, both index and tag are obtained from TLB and thus, TLB lies in critical path.

(b) VI-PT: L1 cache index is derived from VA and TLB lookup is performed in parallel. The tag is obtained from the PA provided by TLB and this is used for tag comparison. Compared to PI-PT design, VI-PT design partially hides the latency of TLB access, and hence, it is commonly used. To perform indexing with VA, virtual index bits should be same as physical index bits and this happens when index is taken from page offset [17], as shown in Figure 3(b). For this, cache associativity should be greater than or equal to cache size divided by page size.

DOI: 10.1002/cpe

For example, with 32KB L1 cache and 4KB page, associativity should be at least 8. This, however, increases cache energy consumption but may not provide corresponding miss rate reduction.

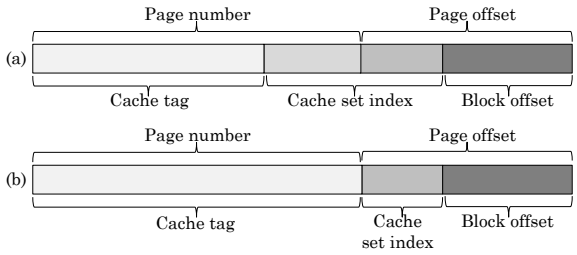


Figure 3. (a) Computation of different values from bits in physical address (general case) (b) Special case where cache index is computed entirely from page offset

(c) VI-VT: Both index and tag are obtained from VA. TLB is accessed only on an L1 cache miss (assuming L2 cache is PI-PT). This design is loosely referred to as virtual cache. Since L1 cache hit rate is high, TLB accesses are greatly reduced, although TLB access now lies on critical path of L1 cache miss. Also, V2PA translation needs to be performed for a block evicted from virtual L1 cache to perform writeback to physical L2 cache. Other challenges of VI-VT caches are discussed in Section 2.2.

(d) PI-VT: Index is computed from the PA provided by TLB. Since index is required before tag, TLB is in critical path and TLB access is required on each access. Clearly, this design does not offer any advantage and hence, it is rarely used [16].

Synonyms and homonyms: Different VAs in the same process mapping to same PA are referred to synonyms. When a same VA exists in different processes, each mapped to a distinct PA, it is referred to as a homonym.

2.2. Factors and tradeoffs in TLB management

Choice of architectural parameters: The timing constraints on TLB access (e.g., before L1 cache tag check but after computation of VA in VI-PT designs) impose strict requirements on TLB design. Parameters such as TLB size, associativity and number of levels present well-known design tradeoffs. For example, using an L0 TLB can reduce access latency, however, the requirement of small size of L0 TLB leads to high miss-rate which necessitates squashing and reissuing many instructions [18]. Similarly, in multicore processors, private per-core TLBs reduce access latency and are more scalable to large number of cores than shared TLB. However, shared TLB provides capacity benefits, which can reduce TLB misses. While some of these TLB design tradeoffs are similar to those of caches, their distinct functions and optimization targets lead to crucial differences [19] and hence, a separate study of TLB is required for addressing these issues.

Challenges in using superpage: The choice of page size is crucial to effectiveness of TLB. While large pages reduce TLB misses, they lead to wastage of physical memory when only a small portion of the large page is used. Further, since the OS tracks memory usage on page granularity, a change in even one byte requires writing back the whole page to secondary storage on modification to a memory mapped file, which leads to huge IO traffic. Also, large pages preclude use of page coloring [20], which is a widely used approach for enabling OS-control of cache. Furthermore, large pages necessitate non-trivial changes in OS operation and memory management [11, 21]. By comparison, small pages enable fine-grain memory management and energy efficient operation.

Challenges in using variable page size: To adapt to the memory access patterns of different applications, multiple page sizes may be allowed. For example, x86-64 supports 4KB, 2MB and 1GB (where 1GB size is not enabled in all systems) page sizes [5]. Similarly, Sparc, Power and Itanium processors allow more than one page sizes. However, leveraging those page sizes leads to

DOI: 10.1002/cpe

higher complexity and they lead to fragmentation when the page size is larger than an adjacent zone of available memory. Further, since page size of an address is unknown during TLB access, the set-index in an SA TLB cannot be determined. To address this, either an FA TLB or one-TLB per page size need to be used which incur additional overheads and lead to sub-optimal use of TLB space. Also, the miss overhead of a TLB allowing multiple page sizes is higher than that of a TLB allowing a single page size [11].

Minimizing TLB miss overhead: Given the large overhead of a TLB miss, intelligent strategies are required for minimizing this. For example, some processors (e.g., Sparc) use a SW-managed, direct-mapped VI-VT cache, termed "translation storage buffer" (TSB) [22]. A TSB entry holds the translation (PPN) for a given VPN. On an ITLB or DTLB miss, TSB is lookedup with the VPN of the missing translation. A hit leads to loading of entry from TSB to TLB whereas a miss requires searching the hash table. Similarly, improving PTW design can reduce miss overhead [1, 23–27]. However, these design options bring their own challenges and tradeoffs.

Minimizing TLB flushing overhead: In multitasking systems and virtualized platforms with consolidated workloads, context switch between processes/tasks requires TLB flushing. To avoid this, VM and/or process-specific tags can be used with each TLB entry which allows preserving them across context switches. However, this tagged TLB approach makes TLB a shared resource which leads to contention.

Improving energy efficiency: TLB power consumption can be nearly 16% of the chip power in some processors [8, 16]. Also, since TLB is accessed frequently, it is also a thermal hotspot. For example, Kadayif et al. [28] show that the power density of ITLB can be 7.8 nW/mm^2, whereas that of DL1 and IL1 are 0.67 and 0.98 nW/mm^2, respectively. Due to this, TLB power management is crucially important.

Challenges in virtual caches: In virtual caches, synonyms may be stored in different cache sets under different VAs and modification of one block renders other blocks stale whereas invalidating synonyms destroys cache locality. Similarly, homonyms may cause wrong data to be accessed. Also, page permission needs to be stored with each block and on a change in permissions of a page, permission information for all blocks from that page need to be updated. Similarly, on a change in page mapping, VA for the block also needs to be changed. Other challenges in virtual caches include maintaining compatibility with conventional processor architectures and OSes [17] and ensuring cache coherency [29]. Due to these challenges, virtual caches are rarely used in commercial systems.

TLB design in GPUs: Due to their throughput oriented architecture, memory access intensity in GPUs is much higher than that in CPUs and due to lockstep execution model of GPU, a single miss leads to stalling of the entire warp [24]. For these reasons, TLB management is even more important in GPU. The requirement of maintaining compatibility with CPU MMU presents further restrictions on GPU MMU design [26].

2.3. A classification of research works

Table I classifies the works based on computing unit, evaluation platform and optimization metric, whereas Table II classifies the works based on their main idea and approach. Since real-systems and simulators offer complimentary insights [30], they are both indispensable tools for evaluating TLBs. For this reason, Table I also summarizes the evaluation platform used by different works.

We now comment on Tables I and II and also summarize other salient points to provide further insights.

1. TLB dynamic energy consumption can be reduced by lowering the number of accesses, e.g., based on spatial and temporal locality of memory accesses, previous translations can be recorded and reused (Table II). To enhance the effectiveness of these techniques, compiler-based schemes have been proposed that seek to reduce page transitions and enhance the locality [16, 48, 49, 56]. Further, among the works that reuse translations (Table II), some works perform reuse both in the *same cycle* and in *consecutive cycles* [54, 68], whereas other works perform reuse in *consecutive cycles* only.

DOI: 10.1002/cpe

Table I. A classification based on processing unit, evaluation platform and optimization target

Category	References
Processing Unit	
GPUs and CPU-GPU systems	[24–26, 31]
CPUs	nearly all others
Evaluation Platform	
Real system	[12, 24, 32, 33]
Both simulator and real-system	[3, 15, 17, 21, 23, 34–39]
Simulator	Nearly all others
Optimization target	
Performance	Nearly all
Energy	[5, 6, 8, 10, 16–18, 26, 28, 29, 31, 40–60]

Table II. A classification based on key idea and management approach

Category	References
Management approach	
Superpage or multiple page sizes	[2, 3, 5–7, 11, 12, 20, 21, 34, 36–38, 40, 61–65]
Software managed TLB	[35, 61, 66, 67]
Software caching	[32]
Designing TLB with non-volatile memory	[31]
Reusing last translation	[16, 48, 49, 54, 56, 57, 59, 68, 69]
Use of compiler	[16, 28, 48, 49, 56, 58, 59, 67, 70]
Using memory region or semantic information	[16, 42, 47, 58]
Using private/shared page information	[1, 29, 39, 71–74]
Use of speculation	[6, 18, 58, 63, 75]
TLB reconfiguration	[5, 41, 45]
TLB partitioning	[76]
Prefetching	[3, 24, 32, 37, 39, 61, 67, 70, 73, 77, 78]
Addressing issues in multicores	[1, 15, 29, 33, 39, 73, 74]
Addressing flushing/shootdown	[1, 24, 33, 39, 55, 70, 76, 79]
TLB coherency issue	[10, 29, 80]
Virtual caches	[8–10, 17, 29, 50–53, 55, 60, 80]
Eliminating TLB	[3, 10, 62]
Bypassing TLB	[18, 71]

2. TLB dynamic energy can be reduced by reducing the number of ways consulted [42], bits precharged [58] or entries accessed [43] on each access and by lowering TLB associativity. Similarly, virtual caches can be used to reduce TLB accesses.

3. TLB leakage energy can be reduced by using reconfiguration [5, 41, 45] and using non-volatile (i.e., low-leakage) memory to design TLB [31].

4. TLB miss-rate can be lowered by increasing TLB reach (e.g., by using superpages or variable page-size), by using prefetching, software caching, TLB partitioning and reducing flushing overhead (Table II).

5. Some techniques use tags (e.g., ASID) to remove homonyms and/or reduce flushing overhead [1, 17, 50, 51, 76, 79].

DOI: 10.1002/cpe

6. Since stack/heap/global-static and private/shared data show different characteristics, utilizing this semantic information and page classification information (respectively) allows design of effective management techniques (Table II).

7. Since ITLB miss-rate is generally much lower than that of DTLB [13, 18, 39], some researchers focus only on DTLB [3, 6, 45, 47, 58, 61, 70, 78]. Some other works focus only on ITLB [49, 59].

8. Some works on TLB management also perform modifications to cache [47, 71] or use similar optimization in cache [41, 42].

9. Several techniques work by using additional storage [1, 18, 51, 68, 75], such as Bloom filter [51].

10. Some works perform classification of TLB misses to derive insights [15, 67].

11. Some works propose TLB enhancements for Java applications [3, 44, 71].

3. TLB ARCHITECTURES AND BENCHMARK STUDIES

Several works explore TLB architectures and study the impact of page size (Table II), TLB associativity [11, 21, 61, 64, 79], replacement policy [13, 31, 37, 45, 61, 71, 79], levels in TLB hierarchy [3, 35, 44, 61] and number of ports [47, 68]. We first summarize works that explore TLB design space (§3.1 and §3.2) and then review skewed-associative (§3.3) and software-managed TLBs (§3.4). We further survey works that characterize TLB behavior of benchmarks suites (§3.5), e.g., SPEC [13, 36, 61], PARSEC [15, 39] and HPCC [36]. Finally, we review TLB design issues in multicore processors (§3.6).

3.1. TLB design space exploration studies

Chen et al. [13] evaluate different TLB architectures using SPEC89 benchmarks. Micro-TLB is an FA TLB with at most 8 entries and is accessed in parallel with instruction cache. It seeks to exploit high locality of instruction accesses and on a miss, it is loaded from the shared TLB. They find that due to different WSS of different applications, their requirement of micro-TLB entries is different, e.g., for applications with many nested loops, the miss rate is very high. On increasing the page size from 4KB to 16KB, only few (e.g., 7) entries are sufficient for holding the entire working sets. As for replacement policy, miss-rate increases on going from LRU to FIFO to random, still, FIFO policy performs close to LRU. As for ITLB, they find that due to instruction locality, most applications place low demand on ITLB. For the same reason, data references dominate a shared TLB and hence, the behavior of data TLB and shared TLB are similar. For shared TLB, the performance depends on TLB coverage since it determines how much of application working set can be covered by the TLB. They also compare separate instruction/data TLB with shared TLB and find that due to contention between instruction and data references, shared TLB may show higher miss rate.

Peng et al. [3] evaluate several techniques to improve TLB performance of Java applications on embedded systems. As for superpaging, they evaluate 64KB and 1MB page sizes with a 32-entry TLB. The working set size of their applications is at most 16MB and hence, using 1MB page removes TLB misses completely. By comparison, use of 64KB page provides as much improvement as the use of 1MB page for some applications (having working set of 305KB) and only provides small benefit in others since their working set is larger than the TLB coverage (32×64KB). They further evaluate a 2-level TLB design with 4KB page size and by experimenting with different sizes/associativity, find a design with 4-way 16-entry L1 TLB and 8-way 256-entry L2 TLB to be the best. Also, the 2-level TLB design provides higher performance than the 1-level TLB design. Superpaging requires significant OS changes and little hardware changes and opposite is true for 2-level TLB. Also, unlike superpaging, use of the 2-level TLB design leads to only small increase in coverage.

DOI: 10.1002/cpe

As for prefetching in prefetch buffer, they observe that both accuracy and coverage of prefetching are relatively insensitive to the size of the buffer and hence, a 4-entry buffer is sufficient. Also, use of prefetching provides higher performance than increasing TLB size by 4 entries. They also consider prefetching directly into the TLB (i.e., using no separate prefetch buffer) and find that it works well for most applications, although it leads to TLB pollution for one application which has poor prefetch accuracy and coverage. They further evaluate a no-TLB design which does not use TLB and thus, removes TLB penalty completely, making this design especially attractive for embedded systems. This design is suitable for managed runtime environments which manage object spaces generally residing in the same address space. On comparing all the approaches, they find that superpaging and no-TLB provide the highest performance. Two-level TLB performs better than prefetching which outperforms single-level TLB. Further, in terms of increasing hardware overhead, different techniques can be ordered as shown in Figure 4.

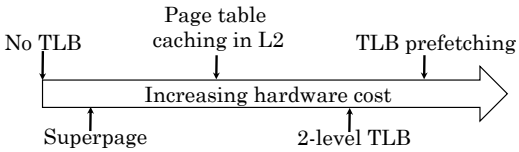


Figure 4. Relative hardware overhead of different techniques [3]

Choi et al. [43] propose organizing TLB as two components: a 2-way filter TLB and a main 2-way TLB, with a total of 8 and 32 entries, respectively. On a TLB access, first only the filter TLB is accessed and on a miss to it, the main TLB is accessed in the next cycle. On a hit in main TLB, TLB access request is served, the entry is moved to filter TLB and the evicted entry from filter TLB is moved to main TLB. On a miss in both filter and main TLBs, the new entry is loaded in filter TLB and its evicted entry in moved to main TLB. Unlike hierarchical TLB design, their design does not force inclusion which increases the total capacity. Also, compared to FA TLB, their technique reduces dynamic energy by consulting smaller number of entries on each access.

Juan et al. [8] compare power consumption of FA, SA and direct-mapped DTLBs for the same miss-rate. They find that for small and large TLB sizes, power consumption is lowest for FA and SA DTLBs, respectively since the size of SA DTLB needs to be higher for achieving the same miss-rate as an FA DTLB. They further propose circuit-level modifications to reduce power consumption of DTLB. They also evaluate VI-VT instruction caches for reducing accesses to ITLB to nearly zero.

3.2. TLB replacement and bypassing policies

Tong et al. [71] note that in managed languages such as Java, garbage collector reclaims allocated memory only at the time of garbage collection. Due to this, some pages become 'futile', i.e., most of their objects are dead. Such pages are responsible for most of the TLB misses since only a few of their blocks are referenced and for a few times only and thus, they have poor spatial and temporal locality. They first propose a sampling-based mechanism to detect futile pages. Their mechanism uses only one 'tagged counter' for entire TLB instead of one counter for each TLB entry. The first incoming TLB entry reserves the counter and its VA tag is stored in the tag of the counter. For each access to the page, the counter is incremented. Thus, the counter is only incremented on access to a particular page (specified by its tag) and hence the term 'tagged counter'. If the counter exceeds a threshold, the page is marked as regular and if the TLB entry is evicted, the page is marked as futile and this information is stored in L2 TLB. In both cases, the counter is released. They show that a statically chosen threshold (e.g., 31) provides reasonably high accuracy.

The further propose two schemes to exploit futile page information. First, they modify the replacement policy such that on an eviction, the futile page is evicted first, else, the baseline replacement policy (LRU) is used. The second scheme extends one way of a VI-PT DL1 cache with a VA. On an L1 TLB miss to futile page, the entry is fetched from L2 TLB, and is inserted in DL1 cache together with the line instead of in L1 TLB. Future references to this line show a miss in L1 TLB and instead hit in the cache. If a different line from the page is accessed, it is fetched in

DOI: 10.1002/cpe

the extended way of DL1 cache following the same procedure. This scheme prevents futile pages from entering and thrashing the TLB. Also, it restricts blocks of a futile page in one way of the cache, alleviating any competition with the blocks of regular page that can use any of the cache ways. The second scheme provides higher performance improvement than the first scheme which in turn outperforms conventional and thrash-resistant replacement policies for cache.

3.3. Skewed-associative TLB

Seznec [64] proposes a "skewed-associative TLB" which is designed after "skewed-associative cache" [81]. Skewed-associative cache seeks to reduce conflict misses for allowing the use of low-associativity cache which also have low access time [82]. It aims to disperse conflicting blocks over multiple sets by using a different hash function for indexing each cache way. A given block address conflicts with a fixed group of blocks but those blocks conflict with other addresses on other ways, which further spreads-out the conflicts. By making careful choice of hashing functions, conflicts can be greatly mitigated and hence, a skewed-associative cache generally shows higher utilization and lower miss-rate than a set-associative cache. However, a skewed-associative cache breaks the concept of a set and hence, it cannot use the replacement policies relying on set-ordering.

The skewed-associative TLB design [64] follows the following properties. First, it requires at least 2^{n+1} ways where 2^n is the first power of two larger than K, the number of page sizes supported. For example, for $K = 4$, an 8-way TLB is required. This is to ensure sufficient number of ways so that any TLB way can map the entire VA space. Also, the number of entries in TLB is kept twice the ratio of maximum and minimum page sizes to allow TLB to map a large contiguous region of the virtual memory. Further, different ways of TLB are indexed using distinct indexing functions. Every VA maps to a unique location in each way and if a VA is mapped to a certain way of TLB, then its page size is known. Furthermore, for any supported page size S, an application using only S page size can use the entire TLB space and this allows concurrently supporting multiple page sizes. They show that skewed-associative TLB effectively supports multiple page sizes, however, its effective associativity (i.e., the number of locations for any virtual page with a specified page size) is smaller than the associativity of TLB (e.g., 2 out of 8 ways when $K = 4$).

3.4. Software managed TLB

Nagle et al. [35] note that software management of TLBs (refer Section 2.1) can lead to large penalties and complexities of TLB management. The relative frequencies of different types of TLB misses may differ across different OSes, such as Ultrix and Mach. For the same application binary, the total TLB miss count and TLB miss penalty can vary significantly with different OSes. Further, total TLB service time can be reduced significantly by lowering the kernel TLB miss penalty. They also study factors such as TLB size and associativity that impact TLB performance, for example, each doubling of TLB size between 32 and 128 entries lowers the TLB service time by nearly half.

3.5. Workload characterization studies

McCurdy et al. [36] study the TLB-behavior of SPEC-FP (floating point portion of SPEC2000 benchmark suite) and HPCC benchmarks [83] and some real scientific applications. Based on implementation-independent experiments, they find that for 4KB and 2MB page sizes, these benchmarks overestimate and underestimate (respectively) the number of TLB entries necessary for good application performance and thus, the TLB-profile of applications is not accurately represented by these benchmarks. They further validate these conclusions on x86 AMD Opteron system. This system uses two different TLBs which have different size and associativity for handling disparate page sizes and hence, page size has significant impact on performance of this system. They observe that an incorrect choice of page size based on benchmark results can lead to up to 50% loss in performance of applications. They also show that based on the implementation-independent results, the TLB behavior of application in other x86 systems (with different TLB and page size values) can be predicted.

DOI: 10.1002/cpe

Kandiraju et al. [61] study the DTLB behavior of SPEC2000 benchmarks. They note that several of SPEC benchmarks have high TLB miss-rates. Optimizations such as increasing TLB size or associativity and superpaging benefit a limited and distinct set of applications. They further compare a 1-level TLB design with a 2-level inclusive TLB design of the same total size. Since inclusivity reduces effective size, 2-level TLB shows higher miss-rate, however, for applications for which miss-rate of 1st level TLB is lower than a threshold, it improves performance due to reduced access time. Most applications do not show clearly-distinct phases of TLB misses and only few program counters are responsible for most of the TLB misses.

To judge the scope of software-management of TLB, they compare performance with OPT (optimal) and LRU replacement policies. For some applications, OPT replacement policy provides much lower miss-rates than LRU which suggests that compiler/software optimizations may be helpful for reducing miss-rates beyond that achieved by hardware-only management policy (LRU). As an example, if the reference behavior of an application can be statically analyzed, compiler can place hints in the application binary to optimize replacement decisions. They also note that even with OPT, the miss-rate is large and hence, techniques for tolerating miss-latency such as prefetching are important. They evaluate three prefetching schemes which bring the data in a prefetch buffer. They observe that prefetching reduces the misses in most applications since they show some regularity of accesses.

Bhattacharjee et al. [15] study the TLB (both ITLB and DTLB) behavior of PARSEC benchmarks. They use real processor for realistic evaluation and execute both 'native' and 'simlarge' datasets on it. Since obtaining address translation information on real system is challenging, they also use a simulator and use 'simlarge' dataset on it. They record how often the same translation leads to misses on multiple cores and contribution of OS and impact of parallelization attributes. They observe that some PARSEC benchmarks show large number of TLB misses which impacts their performance severely. Further, TLB misses from different cores show significant correlation. Since different threads work on similar data/instructions, nearly all (e.g., 95%) the misses are already shown by some other thread. They define "inter-core shared" (ICS) misses, such that a TLB miss on a core is ICS if if it occurs due to access to a translation entry with the same VA page, PA page, page size, protection value and process ID as the translation requested by a prior miss on one of the remaining cores in past M instructions. With increasing M, the fraction of misses classified as ICS increases.

They also note that TLB misses of a core generally have a predictable stride distance from those of another core. Thus, a miss is "inter-core predictable stride" (ICPS) miss with a stride S if the difference between its virtual page and that of previous matching miss (same page size and process ID) is S. Match should occur within M instructions and the stride S should be prominent and repetitive. To remove redundant and stride-predictable misses, TLB miss pattern between cores can be leveraged through schemes such as shared TLB and inter-core prefetching (refer §3.6).

3.6. TLB architectures in multicore processors

For multicore processors, some researchers evaluate private per-core TLB [1, 29, 39, 74] where shared entries are stored in a separate buffer [1, 39] or in the TLB itself [74]. Other researchers evaluate shared TLB [29, 39].

Lustig et al. [39] note that for SPEC and PARSEC benchmarks, ITLB miss rates are orders of magnitude smaller than those of DTLB miss rates. Also, for DTLB, even L2 hits incur overhead and lead to substantial fraction of total TLB penalty. They propose two schemes to reduce TLB misses. Their first scheme aims at reducing ICS misses (§3.5). On each TLB miss, the current core (termed "leader") fills its own TLB and also places this translation in per-core prefetch buffers of other (termed "follower") cores. On a hit to an entry in prefetch buffer, it is moved to TLB. A limitation of this scheme is that it blindly pushes an entry to all the follower cores even though the entry may be shared by only few cores. To avoid such useless prefetches, with each core, they use confidence estimation counters corresponding to remaining cores. The counters of a leader core are incremented/decremented on a hit to or eviction without use of (respectively) a prefetched entry in the follower core. Then, an entry is pushed to a follower only if the leader's counter corresponding

DOI: 10.1002/cpe

to that follower exceeds a threshold. This confidence estimation scheme improves performance significantly.

Their second scheme seeks to reduce ICPS misses (§3.5). For example, if core 0 misses on pages 6,7,9,10 and later, core 1 misses on 10,11,13,14, there is a stride of 4. Thus, although cores miss on different virtual pages, the differences between addresses is same on both cores (1,2,1). Based on it, the second scheme records recurring inter-core strides in virtual pages in a table shared between cores for allowing its use by other cores. This table is looked-up on TLB misses for predicting and prefetching required translations. In the above example, on observing access to pages 10 and 11 on core 1, prefetching of pages 13 and 14 can be issued which avoids misses to them.

They further propose a shared last level TLB (SL-TLB). To avoid the overhead of strict inclusion, SL-TLB is designed to be "mostly inclusive" of L1 TLBs. Thus, on a miss, an entry is placed in both L1 and SL-TLB, however, replacement decisions are made independently by each TLB. In practice, this approach provides near-perfect (e.g., 97%) inclusion, although on shootdown, it requires checking both L1 and SL-TLBs. Since SL-TLB is shared among cores, it incurs higher communication time than private TLBs. To further improve TLB hit-rate, they use stride prefetching in SL-TLB [30]. They show that their techniques reduce TLB misses significantly.

Srikantaiah et al. [74] note that with increasing number of TLB entries, some applications show significant reduction in misses whereas others show negligible reduction. These applications are termed as "borrower" and "donor", respectively. Further, when different threads of same multithreaded application share many pages, duplicate translations of same pages waste TLB capacity. These observations motivate capacity sharing and hence, they present a "synergistic TLB" design that has per-core private TLBs and allows victim entries from one TLB to be stored in another TLB for emulating a distributed-shared TLB. An entry evicted from a borrower's TLB is inserted in donor's TLB and that evicted from a donor's TLB is discarded. By dedicating some entries as borrower or donor, the nature of a TLB is ascertained and remaining entries follow the winning scheme. Compared to the private-TLB design, this design reduces TLB misses and improves performance. However, a limitation of this scheme is that it changes some local TLB hits into remote TLB hits which incurs much larger latency and hence, for some applications, performance improvement is not commensurate with miss reduction. To address this, they present strategies to replicate entries in multiple TLBs and/or migrate them from one TLB to another. For both of them, they present "precise" and "heuristic" approaches where precise approach maintains large amount of access history whereas heuristic approach only tracks local and remote hits. They show that replication strategies do not affect multiprogrammed workloads but improve performance of multithreaded workloads since replication is beneficial only in presence of sharing. Migration strategies boost both multiprogrammed and multithreaded workloads. Also, heuristic approach performs close to the precise approach. Finally, their synergistic TLB design which intelligently uses both replication and migration provides higher performance than using either of them.

Li et al. [1] use page classification (private/shared) information for improving TLB management in multicore processors. They extend each private TLB with a "partial sharing buffer" (PSB) which stores translations shared by different cores and also records process ID. PSB does not store private translations which avoids their remote placement and pollution of PSB. Shared translations are distributed across PSBs of all cores. They classify virtual pages as private or shared by recording the cores that access them. On an L2 TLB miss, if a page/classification is shared, both page table and home PSB are searched in parallel. On a PSB hit, the entry is copied in local TLB of requester core. Otherwise, the entry provided by page table is inserted in local TLB and also in PSB (if shared). TLB shootdown traditionally stalls all the cores, however, their technique avoids stalling cores that do not store a specific translation and thus, their technique achieves shootdown with individual invalidations which incurs much lower overhead. Similar to traditional designs, their private TLB design also requires TLB flush on a context switch or thread migration, however, PSB need not be flushed since it uses process ID as the tag. On reactivation of a process, its left-over shared entries in PSB can be used which reduces the context switch overhead. Compared to shared TLB, their technique scales better with increasing number of cores and compared to private TLB, their technique removes TLB capacity misses and TLB sharing misses by avoiding duplication of shared

DOI: 10.1002/cpe

entries. They show that their technique improves performance by reducing translation latency and miss rate.

4. TECHNIQUES FOR IMPROVING TLB COVERAGE AND PERFORMANCE

In this section, we discuss techniques for improving TLB coverage (§4.1-§4.4), reducing TLB flushing overhead (§4.5) and TLB prefetching (§4.6).

4.1. Leveraging medium-level contiguity

Talluri et al. [2] present two subblock TLB architectures for improving TLB performance which require smaller OS complexity than superpages. The "complete-subblock TLB" (CS-TLB) uses a single tag with a (64KB) superpage-sized zone but has separate PPN, valid bits and attributes for every (4KB) base page mapping (refer Figure 5(b)), which has the disadvantage of increasing TLB size. In "partial-subblock TLB" (PS-TLB), PPN and attributes are shared across base page mappings and hence, the PS-TLB entry is much smaller than a CS-TLB entry. Sharing of TLB entry between base page mappings in PS-TLB is only allowed if base pages map to adjacent physical pages and possess the same attributes (refer Figure 5(c)). Translations for pages not fulfilling these conditions are stored in different TLB entries. They present an algorithm for allocating physical memory, termed "page reservation" that increases the likelihood of such sharing by making a best effort for allocating aligned memory. This algorithm makes best effort (but does not guarantee) to map neighboring base virtual pages to neighboring aligned base physical pages. Overall, CS-TLB allows mapping arbitrary physical pages to a superpage whereas PS-TLB presents stronger restrictions in constructing a superpage. CS-TLB does not require OS changes and hence, is suitable for obtaining high performance when OS changes are not suitable. PS-TLB requires only minor OS changes and on using page reservation, it outperforms superpage TLB.

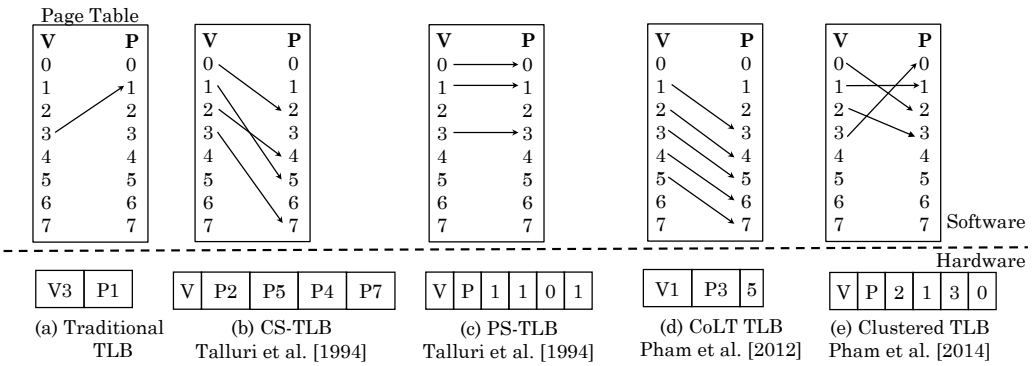


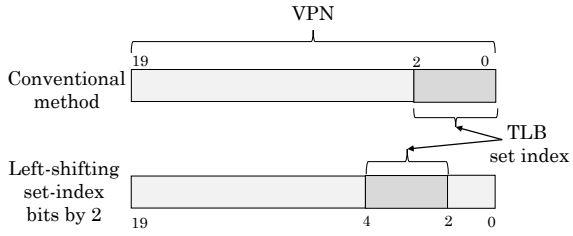
Figure 5. An illustration of (a) conventional TLB (b) CS-TLB [2] (c) PS-TLB [2] (d) "coalesced large reach" (CoLT) TLB [21] and (e) clustered TLB [37]. For CS-TLB and PS-TLB, subblock factor is 4 and for clustered TLB, cluster factor is 4.

Pham et al. [21] note that while superpages reduce TLB misses, they also require generating and managing contiguous physical pages, e.g., with 4KB base page size, 512 adjacent pages are required to generate a 2MB superpage. Further, OS memory management mechanisms naturally lead to medium contiguity (e.g., tens of pages) which is smaller than hundreds of pages of contiguity required for generating superpages, but is achieved without the overheads such as complex algorithms for generating superpage and inflated I/O traffic. Examples of such memory management mechanisms include "buddy allocator", "memory compaction daemons" and Linux "transparent huge page support" [84]. They present a technique which merges multiple contiguous V2PA translations in the TLB to increase its coverage, as shown in Figure 5(d). This is only performed on a TLB miss to avoid any impact on lookup latency. Although merged entries may not be used in close time-window, they do not harm the hit rate. They assume a two-level TLB

DOI: 10.1002/cpe

with SA L1 TLB and L2 TLB for 4KB page size. Another small FA TLB caches superpages and is consulted parallely with L1 TLB. L2 TLB is inclusive of L1 TLB but not superpage TLB.

They propose three variants of their technique. In first variant, merging is done only in SA L1 and L2 TLBs. To merge J entries, the bits used to compute TLB set (index) are left-shifted by $\log_2 J$, as shown in Figure 6. To increase the number of merged entries, index-bits can be further shifted, however, it also increases the conflict misses as a larger number of neighboring entries now map to the same set. Also, increasing J increases the overhead of searching adjacent translations. In second variant, merging is done in FA TLB only. On a miss in all TLBs, up to eight translations are provided by the cache block. If they can be merged, the entry is loaded in FA TLB, otherwise, it is loaded in L1 and L2 TLBs. On insertion into FA TLB, opportunity for further merging is checked between this and an existing entry. In third variant, both L1/L2 TLBs and FA TLB are merged. On a miss in all TLBs, the amount of contiguity present in cache line is checked. If it is smaller than a threshold, it is inserted in L1 and L2 TLBs. However, if it is larger than the contiguity allowed by L1/L2 TLBs, then it is inserted in FA TLB. Since FA TLB is small in size, some useful merged entries may be evicted, and hence, their technique fetches portion of this merged entry into the L2 TLB. Their technique increases TLB coverage and improves performance by virtue of reducing TLB misses.



VPN		TLB-set	
Decimal	Binary	Conventional method	Left-shifting by 2
32	100000	**0**	**0**
33	100001	**1**	**0**
34	100010	**2**	**0**
35	100011	**3**	**0**
36	100100	**4**	**1**
37	100101	**5**	**1**
38	100110	**6**	**1**
39	100111	**7**	**1**

Figure 6. Under conventional set-indexing method, consecutive VPNs are mapped to consecutive TLB sets. On left-shifting index-bits by $\log_2 J$ bits, J VPNs can be mapped to the same set (J=4 in this figure).

Pham et al. [37] note that beyond simple contiguity, many more translations show clustered spatial locality where the clustering of neighboring virtual pages and the corresponding physical pages is similar. They present a TLB design which leverages "clustered spatial locality" and can also leverage "contiguous spatial locality" [21] when it is present. They assume a "cluster factor" of K (e.g., $K = 4$). Then, if a cluster of virtual pages points to a cluster of physical pages, such that on ignoring lower $\log_2 K$ bits, all VPNs of cluster have same bits and all PPNs of cluster have same bits, then the PTEs can be stored in just one "clustered TLB" (cTLB) entry, as shown in Figure 5(e). In addition to cTLB, they also use a "traditional TLB" for storing translations without any spatial locality. Since each cTLB entry may store different number of valid translations, use of traditional replacement policies such as LRU is ineffective in cTLB. Hence, they account for both recency and usefulness of a cTLB entry, where usefulness shows the number of valid and referenced sub-entries. On eviction, these referenced translations are installed into traditional TLB and the remaining translations are evicted. This allows un-merging those translations which are found to be less useful. Their technique does not require OS support and can work even under fragmentation where contiguous spatial locality (exploited by [21]) becomes scarce. Their technique improves performance by reducing TLB misses and increasing TLB coverage.

Romer et al. [38] present policies for dynamically creating superpages to reduce TLB misses while bounding the overhead of superpage promotion. A superpage needs to map a physically contiguous portion of memory, however, the constituent pages of a potential superpage may not be physically contiguous and hence, page copying is required which is the overhead of superpage promotion. They propose a main policy, its approximate variant (both are online policies) and an offline policy. Their main promotion policy attributes each TLB miss to a potential superpage which would have alleviated this miss if it had been already constructed. When the misses attributed to a superpage exceed a threshold (computed based on the tradeoff between overhead of promotion and TLB misses), the superpage is created. There are two ways a potential superpage may avoid miss to a page p. Firstly, the superpage may include both p and an existing page q in TLB and hence, the

DOI: 10.1002/cpe

existing TLB entry for q would have covered p also. Secondly, the TLB capacity increase provided by superpage may avoid eviction of a previous translation of p. Of the two, the first way is more effective in ascertaining the best superpage candidate and is also easier to track. Hence, they also evaluate a promotion policy which only tracks the first reason and the effectiveness of this policy in eliminating TLB misses is close to that of main policy which tracks both reasons. They further evaluate an offline policy that assumes knowledge of complete TLB miss stream and based on this, it iteratively promotes the superpages with the highest ratio of 'benefit over cost' of promotion, till no such superpage exists. Compared to this offline policy which cannot be implemented in practice, their main policy achieves comparable performance. Also, compared to using 4KB pages, their policies reduce execution time.

4.2. Using speculative address translation approach

In "reservation-based physical memory allocation" approach [2, 12], initially small (4KB) pages are used for all memory. When all 4KB pages of a 2MB VM zone are used, that zone gets promoted to a large page. To allow this, small pages are placed into reservations such that 4KB pages are positioned within a 2MB zone of physical memory analogous to their position within the 2MB zone of VM. Thus, in a "large page reservation" (LPR), successive virtual pages are also successive physical pages. Barr et al. [63] present a technique which leverages the contiguity and alignment produced by such memory allocators to predict address translations from that of PA of neighboring pages. They use "SpecTLB" which is a buffer that is consulted on TLB miss to check whether the virtual page showing TLB miss may be part of an LPR. If so, PA of the missing page can be interpolated between the beginning and ending PAs of the LPR using the page's location within the LPR.

Since the mapping predicted by SpecTLB may be incorrect, a hit in SpecTLB must be confirmed against access to the page table. The translation provided by SpecTLB allows speculative execution while page table access is performed in parallel. If translation generated by SpecTLB is validated, the speculative execution is committed, otherwise, the execution restarts from the first wrong prediction. Thus, SpecTLB seeks to reduce TLB miss penalty by removing page table access from critical path, although it does not reduce accesses to page table. Also, it seeks to provide performance close to large-pages when use of large pages may not be practical, e.g., in virtualization. They show that prediction accuracy of SpecTLB is high (e.g., 99%). Further, compared to TLB prefetching, their technique is better suited to programs with random access patterns and the speculation performed by their technique is mostly useful.

Pham et al. [34] note that large pages lead to overhead in memory management, such as reducing the scope of memory deduplication among VMs in overcommitted processors and hampering the ease of VM migrations, etc. Due to this, virtualization modules generally break large pages of guest OS into small physical pages to achieve system-level gains at the cost of losing their TLB advantages. They present a technique to regain the address translation benefit lost by splitting of pages. They note that page-splitting does not basically change PA or VA spaces since most constituent small pages are not relocated. Hence, most of these small pages preserve their original alignment and contiguity in both PA and VA spaces. Their technique tracks such aligned, contiguous yet split portions and speculates about small pages by interpolating based on information stored in TLB about a single speculative large-page translation. For example, in Figure 7, where a large page consists of four small pages, the guest large page GVP4-7 (GVP = guest virtual page) is splintered, still, SPPs (system physical pages) can be speculated using interpolation approach. These speculations are validated against page table walk which now happens off the critical path. Thus, their technique allows splitting large pages for fine-grained memory management while keeping TLB performance close to that of original large pages. Their technique bears similarity with SpecTLB approach [63] but unlike SpecTLB approach, they do not require special unit for speculation. Also, they propose strategies to alleviate performance impact of incorrect speculation using PTE prefetching that lowers the need of accessing page table for verifying some of the speculations.

DOI: 10.1002/cpe

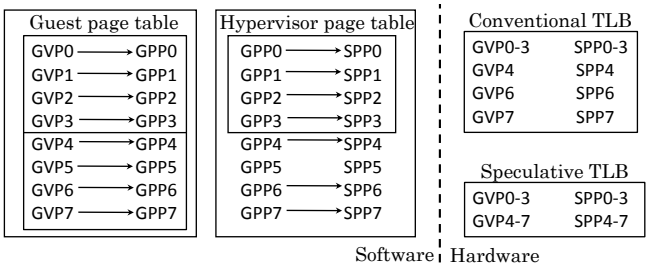


Figure 7. An illustration of speculative TLB design [34] (GVP = guest virtual page, GPP/SPP = guest/system physical page). Here, guest large page GVP4-7 is split, but SPPs can be interpolated. To map the page table, conventional TLB requires four entries, whereas the speculative TLB requires only two entries (including a speculative entry for GVP4-7).

4.3. Using address indirection approach

Swanson et al. [7] note that the range of PAs that can be addressed by processors generally exceeds the actual available memory capacity. For example, with 32 PA bits and 1GB DRAM, only one fourth of PAs generated are valid. For such cases, they propose using a fraction of this unused PA range (called "shadow" pages) to again virtualize the physical memory. This indirection level between application VAs and PAs allows creating arbitrary-size adjacent superpages from non-adjacent "real" pages. On an access, the memory controller translates shadow PAs into real PAs and this mapping is stored in a "memory-controller TLB" (MTLB), as illustrated in Figure 8. The processor and memory management software work with shadow addresses and they are stored in TLB as mappings for VAs and in cache as physical tags for cache blocks. Their technique (referred to as shadow memory design technique or SMDT) allows using the superpage capabilities already present in processors. They show that SMDT allows achieving the performance of a double-sized TLB. However, SMDT cannot work for systems without unused physical memory and it may not support precise exceptions.

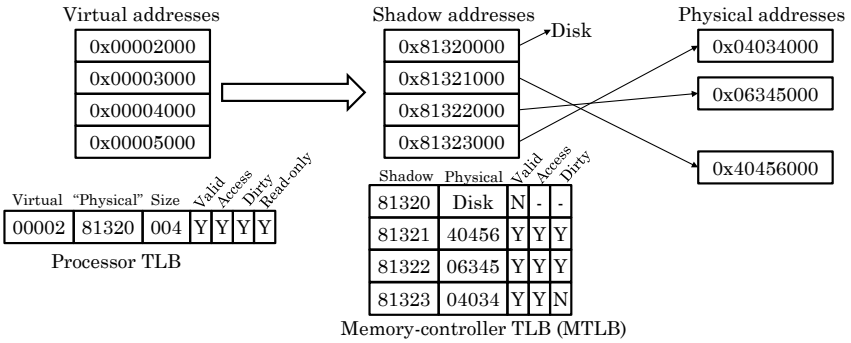


Figure 8. An illustration of shadow memory design technique [7]

Park et al. [65] note that SMDT [7] allows creating a superpage from non-consecutive physical pages, however, it maintains partial coarse mapping in the TLB. Further, PS-TLB scheme [2] stores full mapping information in TLB but offers limited ease of superpage creation. To bring the best of these two together, Park et al. [65] propose a design which uses MTLB from SMDT and TLB entry format from PS-TLB scheme. This format allows identifying invalid mappings inside processor which avoids the need of instruction back-off hardware, unlike in SMDT where invalid mappings are identified late and hence, many instructions need to be backed-off (i.e., squashed and restarted). Also, in SMDT, a change in mapping between shadow and real PA (e.g., on paging-out) requires flushing of all cache blocks related to the mapping, whereas their design only requires resetting associated valid bits in the TLB. In SMDT, an entry for shadow to real PA may not be found in MTLB which leads to a miss. However, in their design, such misses do not happen since the TLB

DOI: 10.1002/cpe

maintains valid bits per base page. Also, per base-page reference information is not managed by MTLB, rather, they are stored in CPU to improve efficacy of page replacement policy. Compared to using a single base-page size, their technique reduces TLB misses significantly and close to that removed by SMDT. The limitation of their technique is that its maximum superpage size is limited by that allowed by PS-TLB.

4.4. Supporting variable-size pages

Papadopoulou et al. [6] study the TLB behavior of TLB-intensive scale-out and commercial applications and observe that several applications frequently use superpages. Further, applications prefer the largest superpage and use of intermediate page sizes is infrequent. Based on it, they present a scheme for predicting whether a memory access is to a superpage or not. For size-prediction, they design two predictors which use instruction's address (PC) and base register value (respectively), to access the predictor tables. Since supporting multiple page sizes in an SA TLB is challenging (§2.2) and predicting the exact page size incurs large latency, they propose distinguishing only between 8KB page and superpage (64KB, 512KB, 4MB) since only 8KB and 4MB sizes are frequently used. Their TLB uses largest tag size for 8KB page and hence, can support any page size. Any page size can use any set. The indexing bits of all superpages are the same and due to infrequent use of 64KB and 512KB pages, they do not cause TLB pressure.

In the general case, the prediction is correct and TLB access is a hit. In case of TLB miss, a second TLB access is required with opposite page size prediction to account for misprediction. If TLB access still leads to miss, the page table needs to be accessed. Overall, they present an SA TLB design which simultaneously stores translations of different page sizes and seeks to achieve miss-rate of FA TLB and access latency/energy of SA TLB. They also discuss exact page size prediction for applications showing large variation in page size usage. Further, they evaluate "skewed-associative TLB" design [64] and enhance it with prominent page-size prediction approach to improve its effective associativity. They show that with only a 32B superpage predictor, their technique allows a 4-way 256-entry TLB to capture more memory references than a 128-way FA TLB. A limitation of their approach is that when the difference in the smallest and the largest page sizes is large, TLB pressure increases greatly, e.g., when 2MB and 1GB pages share the same index, 512 successive 2MB pages compete for the same set. For such cases, prediction can be made among groups of page sizes.

Talluri et al. [11] evaluate use of large page size (32KB) and two page sizes (4KB and 32KB) to improve TLB performance. For two page size, they use the following page-size assignment scheme. For a memory region of 32KB, if at least half of 4KB-size blocks have been accessed in last K references, the region is promoted to the large page. They observe that with each doubling of page size between 4KB and 32KB, the increase in WSS (distinct pages referenced in an interval) ranges from 1% to 30%. The reason for less than 2X increase in page size is the presence of spatial locality. For example, applications that linearly iterate over their memory region in a loop already have their address space covered in the working set and hence, a page size increase does not have much impact on WSS. Large increase in WSS is seen for applications with sparse address space. They also note that on using two page sizes, the increase in WSS is smaller than using either 8KB, 16KB or 32KB pages. Thus, using two page sizes incurs smaller memory demand than a single large page of even 8KB. The CPI contribution of TLB (CPI_{TLB}) is reduced significantly on using 32KB page. On using two page sizes, reduction in CPI_{TLB} ranges from near-zero to that achieved using 32KB page alone. They also discuss different approaches to provide variable page-support in an SA TLB, e.g., indexing the TLB with page number of small or large page size or with exact page size.

4.5. Reducing TLB flushing overhead

Venkatasubramanian et al. [79] note that workload consolidation through virtualization increases the number of distinct VA spaces and the context switche frequency between them. Since context switch requires flushing the TLB, virtualization can increase TLB flushes by 10× and miss rate of DTLB and ITLB by 5× and 70×, respectively. Further, tagging each TLB entry with per-VM tag can avoid TLB flushing on context-switch between VMs, but not between two processes of the same VM. To

DOI: 10.1002/cpe

avoid flushing on both types of context-switches, they propose using process-specific tags which are generated from "page directory base register" (CR3 register in x86). To reduce the number of bits in the tag, bitmasking or perfect hashing can be used such that final tags are still unique across processes. Their technique can be used with both virtualized and non-virtualized platforms and can effectively reduce TLB flushes and miss rate. They also study the impact of TLB size, associativity and replacement policy and TMT size on the improvements.

Venkatasubramanian et al. [76] further note that in case of consolidated workloads, the TLB quota of a VM is decided by the time for which VM is running, WSS of workload executing in the VM and workloads in other VMs sharing the TLB. Using their tagged TLB approach, they propose controlling the TLB quota of a VM at TLB-way level by selecting the victim based on their current share. This TLB partitioning is analogous to way-based cache partitioning. Using this, both performance isolation and priority enforcement between different VMs or processes can be achieved.

Chukhman et al. [70] note that in multi-tasking systems, inter-task interference in TLBs can harm performance and predictability. To reduce this overhead, the TLB entries (specifically VPNs) can be saved during context switch along with process state. Further, by only saving entries that are live or will be used in near future (called "extended live set" or ELS), the storage overhead can be further reduced. Their technique determines ELS at context-switch time. The compiler extracts static information about the program, such as register usage and memory access pattern. Also, OS gathers runtime information, e.g., code location where context switch happened and pointer values. Based on compiler information, their technique finds the pointer values that are useful for a task to ascertain live VPNs. Using this and the access pattern and array sizes, VPNs that are expected to be accessed in future are estimated. Based on it, after preemption of a task, the ELS of the next task is prefetched in the TLB. The number of entries to be prefetched are determined based on the TLB configuration, array dimensions, loop length and algorithm characteristics. Also, the entries required earliest are given first priority. They show that their technique reduces TLB misses due to inter-task interference which improves predictability of TLB and leads to tighter estimation of "worst-case execution time".

Villavieja et al. [33] present a technique to alleviate performance loss from TLB shootdowns caused by page remapping (and not context switches). Based on studies on real systems, they note that with increasing number of cores, the overhead and frequency of shootdown increases rapidly and even superlinearly. A shootdown stalls nearly all the cores, although many of these may not store the changed mapping [14]. Also, shootdowns occur frequently in applications with memory or file IO operations due to memory remapping. Further, up to 4% and 25% of cycles may be spent on shootdown in 16 and 128-core machines. They propose adding a second-level TLB implemented as a 2-way, 4096 entry cache which works as a "dictionary directory" (DiDi). It tracks the location of every entry in L1 TLBs of the entire system, since a translation may be present in multiple L1 TLBs. DiDi intercepts all the insertions and evictions in L1 TLBs without incurring latency penalty and thus, it is kept up-to-date. To perform remote TLB invalidations, it uses a "pending TLB invalidation buffer" (PTIB). On a TLB shootdown, invalidation request is first sent to DiDi and then, DiDi forwards invalidation request to the PTIB of only those cores that store the changed mapping. This avoids sending invalidations to cores that do not store the changed entry and also avoids the need of interrupting the cores and its associated costs, e.g., pipeline flush, context saving and storing, etc. They show that their technique reduces latency overhead of shootdown by an order of magnitude.

4.6. TLB Prefetching

Some prefetching techniques bring data in a separate prefetch buffer [3, 32, 39, 61, 77, 78], whereas others bring data in TLB itself [3, 24, 37, 67, 70]. We now review these techniques.

Bala et al. [32] propose using prefetching and software caching to reduce the number and penalty of kernel TLB misses. They note that after an "inter-process communication" (INPC), a large fraction of page table accesses causing TLB misses are amenable to prediction. Hence, prefetching them in the kernel during INPC can remove a large number of misses. For a 3-level page table

DOI: 10.1002/cpe

hierarchy, they prefetch L1K [32] PTEs that map INPC data structures, L2 PTEs that map message buffers and their associated L3 PTEs and L2 PTEs mapping the process code segments and stack. TLB entries prefetched are stored in a different "prefetch TLB". They further propose a "software cache for TLB" (STLB) which stores L1K, L2 and L3 entries replaced from hardware TLB. Initially, STLB code branches away from the generic trap handler and checks STLB. An STLB hit leads to insertion of the entry into hardware TLB and thus, the penalty of generic trap handler is avoided. An STLB miss leads to branching of the code back to the generic trap handler. Due to hierarchical organization of page table, TLB miss in one level is handled by probing the next level which can lead to cascaded TLB misses. STLB alleviates such cascaded misses by providing a flat organization of TLB entries and avoiding further accesses to page tables. They show that prefetching removes nearly half of kernel TLB misses and caching reduces TLB miss costs by allowing an efficient trap path for TLB misses.

Saulsbury et al. [77] present a TLB prefetching technique which functions on the idea that pages accessed in close temporal neighborhood in the past will be accessed in close temporal neighborhood in future also. Their technique builds an LRU stack of PTEs using a doubly-linked list which has previous and next pointers. On eviction of an entry from TLB, it is placed on top of the stack and its next pointer is stored as previously evicted entry (whose previous pointer is stored as this entry). While loading an entry on a miss, their prefetching technique brings the entries pointed to by previous and next pointers of this entry and stores them in a prefetch buffer. They show that their technique can accurately predict more than half of TLB misses and improve performance by prefetching them. The limitation of their technique is that it stores prediction information (e.g., pointers) in page table which requires modifications to page table. Also, their technique may not work for TLBs which do not use LRU replacement policy.

Kandiraju et al. [78] evaluate several prefetching techniques, e.g., "arbitrary stride prefetching", "Markov prefetching" and "recency-based prefetching" [77] in context of TLBs. They also propose a distance prefetching (DP) technique which predicts the address to prefetch based on the strides between consecutive addresses. For example, if the addresses accessed are 10, 11, 13, 14, 16 and 17, then, a stride of '1' is followed by a stride of '2' and hence, a two-table entry suffices for making prediction. All techniques bring the prefetches into a prefetch buffer. For applications with regular strided accesses, all techniques give good accuracy. For those with large dataset, Markov prefetching performs poorly with small prediction table due to requirement of keeping large history. RP performs well for applications with predictable repetitions of history. Arbitrary stride prefetching performs well for many applications which work well with Markov prefetching and RP, however, for applications without strided accesses, it does not perform well. Overall, DP provides highest prediction accuracy and works well for different page sizes and TLB configurations. DP seeks to exploit strided pattern when it is present and becomes increasingly history-based in absence of strided pattern. Hence, it performs close to the better of strided and history-based techniques.

5. TECHNIQUES FOR IMPROVING ENERGY EFFICIENCY

In this section, we discuss techniques for saving TLB dynamic energy (§5.1-§5.3) and leakage energy (§5.4-§5.5). Use of virtual cache design for reducing TLB dynamic energy is discussed in next section (§6).

5.1. Reusing recent translations

Kadayif et al. [59] note that due to high locality in instruction stream, by storing and reusing the translation to current page, accesses to ITLB can be significantly reduced since an access to ITLB will only be required when another page is accessed. They propose four strategies to implement this which use hardware and/or software control. In first strategy, hardware checks VA generated by the PC and compares it with the VPN part of currently stored translation. In case of a match, access to ITLB is avoided. The limitation of this strategy is the need of checking on every instruction fetch. In second strategy, access to ITLB is explicitly controlled by compiler. The execution can transition

DOI: 10.1002/cpe

between instruction pages in two cases: on a branch instruction and on a page boundary. They assume that for the target of branch instruction, ITLB access is always performed. Further, for page boundary case, a branch instruction is inserted by the compiler with target as the next instruction (i.e., first instruction on the next page). This reduces the second case to first case.

A limitation of second strategy is that assuming branch target to be on different page is overly-conservative. To avoid this, in the third strategy, compiler statically analyzes the code and finds whether the target is on the same page and this happens when targets are supplied as "immediate operands" or "PC relative operands". However, third strategy is still conservative due to its inability to use runtime information. The fourth strategy uses runtime information to find whether the branch target is to a different page and whether branch is taken at all. This is implemented by modifying branch predictors. They show that all four strategies reduce TLB energy significantly. Among the four, second strategy consumes highest energy. First and third strategies perform comparably whereas the fourth strategy is the best and it comes very close to an optimal scheme that accesses ITLB only when required.

Ballapuram et al. [54] evaluate a multi-issue processor with multi-ported TLB (e.g., 4-issue and 4-ported) and observe that a large fraction of DTLB accesses occurring within the same cycle and in successive cycles are to the same page or even the same cache block. They propose two techniques to exploit this behavior. For an N-wide processor, the first technique uses $\binom{N}{2}$ (i.e., 'N choose 2') comparators to detect identical VPNs in the same cycle. Based on this, only unique VPNs are sent to DTLB. For example, Figure 9(a), and 9(b) show the VAs and corresponding VPNs arriving at four ports of a TLB, respectively. With intra-cycle compaction, VPNs in the same cycle are compared to avoid TLB accesses for identical VPNs. In Figure 9(c), accesses for two VPNs in cycle J and one VPN in cycle J+1 can be avoided. This "intra-cycle compaction" technique allows reducing number of ports and access latency of TLB.

Their second technique, termed as "inter-cycle compaction", latches most recently used TLB entry and its VPN. In the next cycle, the latch is read to get the translation. For multi-ported TLB, one latch is used for each port. In Figure 9(d), accesses for first and second ports can be avoided in cycle J+1 since they are same as that observed in cycle J. This technique is especially effective with semantic-aware memory architecture [47] (§5.2). They show that their techniques reduce TLB energy significantly. They further combine both techniques and apply them to both ITLB and DTLB and observe that this approach provides even larger energy saving.

(a) VAs at four ports of TLB

Cycle				
J	0x8395BA11	0x8395BAEE	0x8395BA0F	0xEEFFDDAA
J+1	0x8395BAF1	0x8395BCED	0x87241956	-----

(b) Corresponding VPNs

Cycle				
J	0x8395B	0x8395B	0x8395B	0xEEFFD
J+1	0x8395B	0x8395B	0x87241	-----

(c) VPNs after intra-cycle compaction [(b)⟶(c)]

Cycle				
J	0x8395B	-----	-----	0xEEFFD
J+1	0x8395B	-----	0x87241	-----

(d) VPNs after inter-cycle compaction [(b)⟶(d)]

Cycle				
J	0x8395B	0x8395B	0x8395B	0xEEFFD
J+1	-----	-----	0x87241	-----

Figure 9. (a) An example of VAs at four port of TLBs and (b) their corresponding VPNs. (c) VPNs after applying "intra-cycle compaction" in both cycles J and J+1 (d) VPNs after applying "inter-cycle compaction" at the same port for two consecutive cycles [54]. Thus, both techniques exploit locality to reduce TLB accesses.

Jeyapaul et al. [48] note that when successive memory accesses are to different pages, effectiveness of translation-reuse based techniques (Table II) is significantly reduced. They propose compiler techniques to reduce page-transitions in both data and instruction accesses. For DTLB, they perform "array interleaving", "loop unrolling", "instruction scheduling" and "operand

DOI: 10.1002/cpe

reordering". For innermost loop of nested loops, they perform "array interleaving" between two arrays if they are of the same size and have same reference functions (i.e., access pattern), as shown in Figure 10. Then, to increase the effectiveness of instruction scheduling, they perform "loop unrolling" if at least one instruction exists for which consecutively scheduling two instances of it from distinct iterations does not lead to inter-instruction page transition. After these two, both "instruction scheduling" and "operand reordering" are together performed, where the former aggregates instructions accessing same pages and the latter changes memory access pattern by reordering operands. They show that the problem of minimizing page transitions using these two approaches is NP-complete and hence, they propose a greedy heuristic for this. This heuristic records the last scheduled instruction. Then, out of all instructions that are ready for scheduling, a pair of instructions (if possible, otherwise a single instruction) is chosen that does not lead to page transition between them and with the last scheduled instruction.

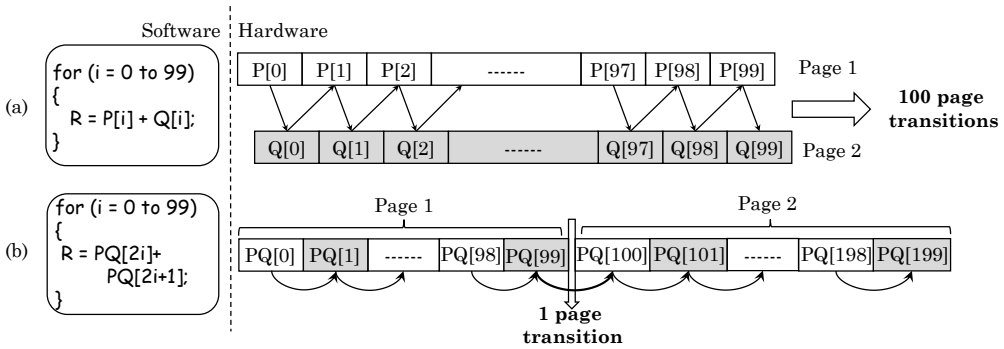


Figure 10. (a) Since P and Q arrays are on different page, accessing them together leads to many page transitions, leading to poor TLB efficiency. (b) Using array interleaving, page transitions can be significantly reduced. [48]

For ITLB optimization, they note that the successive instructions may be from different pages when (1) loop block of a program crosses page limit and (2) callee and caller function are on different pages. They propose a code-placement strategy which reallocates function blocks of a program to minimize page switches. For example, function position limits are used over a single page to ensure placing a function in the same page. Loops and functions are considered in order of decreasing loop-count and number of calls to them (respectively), which makes this a greedy approach. They show that their techniques are effective in reducing page transitions and TLB energy.

Kadayif et al. [16] propose compiler techniques to intelligently store some translations in "translation registers" (TRs) which can be maximally reused for avoiding accesses to DTLB. The compiler directs which TRs should be used for different data accesses and deploys them only when the translation is guaranteed to be stored in them, otherwise, DTLB is accessed. For this, compiler inserts the information about accessing DTLB or TR in the load/store instruction itself and this is seen by the runtime. The compiler identifies four types of data accesses. First type is the global scalar variables, which are statically analyzable. If multiple variables are on the same page, their translation is loaded in TR. Second type is the global array variables which are generally addressed in loops. The compiler applies loop transformations, e.g., for stride-based array scans, a single loop can be split in two loops such that outer loop iteration is performed over virtual pages where a translation is stored in a TR and this TR is used in the inner loop where data access happens within a page.

The third class is the heap accesses for dynamically allocated data (e.g., through `malloc()`) that are generally accessed using pointers. By enforcing that memory allocated is within one page (assuming that allocated size is smaller than a page) and tracking the period over which the memory is accessed, TR can be loaded and used over this period. The fourth class is the stack accesses which show high locality, especially in C and Fortran languages where arrays are passed by pointer/reference. For them, just one TR is sufficient. They show that for array- and pointer-based

DOI: 10.1002/cpe

applications, their technique provides large energy saving by avoiding DTLB accesses, although their technique increases the number of instructions executed.

5.2. *Utilizing semantic information*

Based on the convention of programming languages, virtual address space is generally partitioned into multiple non-overlapping semantic regions, such as stack, heap, etc., which show different characteristics. Several techniques leverage this information to rearchitect and optimize TLB.

Lee et al. [47] propose separating stack, heap and global static accesses in different streams and directing them to dedicated TLB and cache components. They decompose a DTLB into a "stack TLB", a "global static TLB" and a "heap TLB". Heap TLB is also used as second level TLB for stack and global static TLBs. Each DTLB access is channeled to corresponding TLB. Compared to accessing FA TLB, this approach avoids conflict misses and reduces dynamic energy by serving most of the accesses with stack and global static TLBs that are much smaller. Also, compared to a monolithic TLB, these small TLBs can be easily multi-ported with less area/latency overhead. Similarly, they decompose DL1 cache into a small "stack cache", a small "global static cache" and a large cache for serving remaining accesses (refer Figure 11). Their technique saves significant energy in DTLB and DL1 cache.

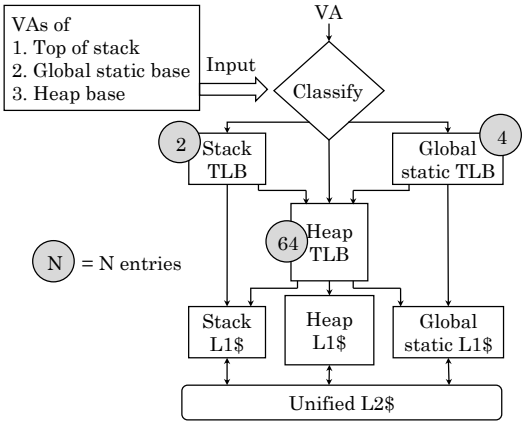


Figure 11. Semantic-aware partitioning of TLB and cache [47]. By directing stack, heap and global static accesses to dedicated TLB and cache components, their different characteristics can be more effectively exploited than in a conventional design using a single TLB for all the streams. The size of these TLBs/caches is determined based on the locality present in each stream.

Ballapuram et al. [44] note that in Java applications, most accesses to ITLB originate from heap for fetching and executing the native code. However, these accesses conflict with JVM's code accesses and increase ITLB misses. They propose dividing ITLB into an ITLB for Java code accesses and an ITLB for Java application accesses to heap. The first ITLB is small (2 entry) since the Java code is static, whereas the second ITLB is large (32 entry) since Java application codes are dynamic involving creation and release of objects at runtime due to which the chances of allocation of new objects from same or consecutive pages is low. They also note that Java applications generally create short-lived objects which show good locality and account for a large fraction of total memory accesses. Based on it, they further propose dividing ITLB for Java application code into two levels, where the smaller first-level ITLB supports dynamic object code accesses in Java applications and the larger second-level is used as a backup.

Furthermore, in Java applications, reads in the global static data region are negligible, and most data memory accesses are due to reads and writes to the heap. Based on it, they propose splitting DTLB into a read DTLB and a write DTLB. Load addresses and store addresses go to read DTLB and write DTLB, respectively, which avoids contention. During execution of Java application, read DTLB is referenced by JVM and application data references for reading heap data and write DTLB is accessed by dynamic or JIT compiler for native code write accesses. Their techniques save TLB

DOI: 10.1002/cpe

energy with negligible impact on performance, and by combining all the techniques (refer Figure 12), energy saving can be further improved.

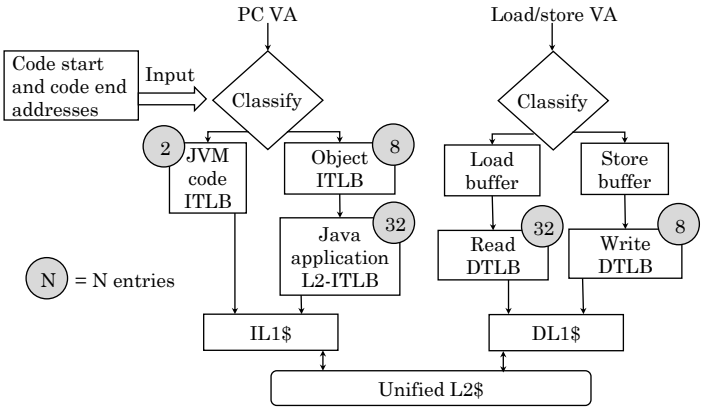


Figure 12. Semantic-aware TLB design for Java applications [44]

Ballapuram et al. [58] note that in many applications, accesses to stack, global data and heap happen in decreasing frequency. Due to the random nature of heap accesses, predicting them is challenging. By comparison, stack accesses are highly predictable and the size of global data accessed by an application is fixed at compile time. Due to this, address translations for both stack and global data require only few bits instead of the complete VPN. Based on these, they reorganize the L1 TLB into a "stack TLB" and a "global static TLB", whereas the L2 TLB is used for all data accesses. They further propose a speculative translation scheme for "stack TLB" and a deterministic translation scheme for global static TLB. Both techniques save energy by reducing the number of bits precharged in each TLB access. For any VPN translation, the first scheme speculatively precharges the same number of bits that were precharged in last translation. Then, it checks for correctness of translation. If the translation is found to be incorrect (which happens rarely), it is squashed, the number of precharged bits is increased and correct translation is obtained in the next cycle. The second scheme works by computing the minimum number of bits required for addressing global data, based on the global data footprint of an application. From this, the number of lower-order bits that need to be precharged and compared are deterministically computed by the loader and thus, this scheme does not incur performance overhead. Remaining higher order bits in global data accesses remain same during application execution. Their schemes save large amount of energy with negligible performance loss.

Fang et al. [42] note that in parallel access mode, tag and data of all K-ways of the chosen L1 cache set are accessed in parallel. Since at least $K-1$ of these accesses are useless, this approach leads to energy wastage. They propose a technique which uses privilege-level and memory-region information to reduce energy of TLB and L1 cache. They note that a "user-mode" instruction fetch cannot hit in a cache block storing "kernel-mode" code. Hence, they store this privilege-level information with each block of IL1 and ITLB and on a user-mode instruction fetch, only the ways storing usermode code are consulted. Similarly, an access to heap data cannot hit in a block storing stack data and vice versa. By storing this information in each DL1 cache block, accesses to those ways are avoided which are guaranteed to lead to a miss. Their technique saves cache and TLB dynamic energy without impacting access latency or hit rate.

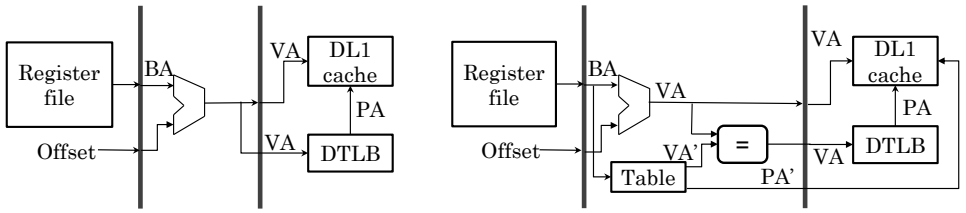
5.3. Using pre-translation approach

Xue et al. [18] propose a technique to reduce TLB accesses through page number prediction and pretranslation. They assume PT L1 cache which avoids the synonym problem. They note that in base-displacement addressing mode, the base address (BA) is a good predictor of final VPN due to three possible reasons: (1) a large percentage of static instructions use BAs which map to precisely one VPN for entire program execution. For cases where page number is on a different

DOI: 10.1002/cpe

page compared to the BA, the previous page number can be used as the prediction. (2) The dynamic fraction of such BAs is even higher than the static fraction and they can be easily predicted (3) If BA to VPN mapping remains same for large duration of execution, accurate prediction can be made even if many BAs have one-to-many mappings over the entire execution. They observe that for PARSEC and SPEC benchmarks, reasons (2) and (3) primarily contribute to high predictability of page numbers. Also, their approach achieves high prediction accuracy (99% for ITLB and between 52%-75% for DTLB for different ISAs).

Based on it, they use BA to access a predictor table which includes tag and TLB entry of the predicted page. Using this support, their technique predicts the page of the computed VA. A correct prediction obviates the need of DTLB access whereas an incorrect prediction does not incur any penalty since the computed VA is used to access DTLB for translation (refer Figure 13). No prediction is made if a BA is not found in the table. In this case and in case of misprediction, the table is updated with a predicted VA and translation. As for instruction fetch, which uses no BA, they leverage page-level locality and predict the next PC page based on previous PC. For workloads with small instruction footprint, this scheme works well. The table is kept coherent with L1 TLB by enforcing inclusion. Their technique brings significant reduction in power consumption without harming performance and provides higher performance/energy improvement than using L0 TLB.



(a) Baseline (b) Design proposed by Xue et al. [2013]

Figure 13. (a) Baseline design in base-displacement addressing mode (b) Design proposed by Xue et al. [18] for avoiding DTLB accesses

5.4. Using TLB reconfiguration

Some processors use two-level TLB design with different L1 TLBs for every page size, e.g., 4KB, 2MB and 1GB as shown in Figure 14. Karakostas et al. [5] note that in these processors, large dynamic energy is consumed by L1 TLB (due to accesses to multiple TLBs) and page walks for small page size (4KB). Further, the contribution of all L1 TLBs to hits is not same especially on using schemes for increasing TLB reach, and hence, accessing all the L1 TLBs may not boost performance. Based on this, they propose a TLB way-reconfiguration technique which finds the utility of each L1 TLB way by recording hits to them. Then, if the difference between the actual miss-rate and that with fewer ways is found to be below threshold, then their technique disables ways at power-of-two granularity. If the miss-rate difference between two intervals is larger than a threshold, all the ways are enabled. They further add an L1-range TLB and apply their TLB reconfiguration scheme to "redundant memory mapping" scheme [27]. They show that their techniques save dynamic energy of address translation for a slight increase in TLB miss penalty.

Figure 14. A two-level TLB organization which supports multiple page sizes using different L1 TLBs.

Balasubramonian et al. [41] present reconfigurable TLB and cache organizations that are designed as logically two-level and physically one-level non-inclusive hierarchies. The partitioning between two levels can be dynamically performed to control latency and capacity. For example, for total cache space of 2MB, the L1 size can range between 256KB to 2MB (with remaining for L2) and

DOI: 10.1002/cpe

the associativity can range between one and four. Similarly, a 512-entry FA TLB can be designed in multiple of 64 entries, for a total of eight possible sizes. Their management technique seeks to achieve the right tradeoff between miss rate and hit latency in each phase depending on hit/miss latency tolerance of each application. On a change in branch frequency and cache miss-count, a phase change is detected and at this point, every cache configuration is evaluated for 100K cycles and the one providing best performance is used until a phase change is detected. Along with cache reconfiguration, they also perform TLB reconfiguration. If the TLB miss handler cycles exceed a threshold fraction (e.g., 3%) of total execution time of a 1M cycle interval, then the L1 TLB size is increased. Further, if the number of TLB entries referenced in an interval is less than half, the L1 TLB size is decreased. They show that their technique is effective in improving performance and energy efficiency and different cache/TLB sizes are optimal for different applications/phases.

Delaluz et al. [45] present two techniques to dynamically vary DTLB size. In the first technique, in each interval (e.g., 1M TLB accesses) the DTLB size is initially set to the lowest (e.g., 1/64 of original size). After a fixed period (e.g., 100K TLB accesses), if the miss-rate is higher than that in the previous interval, DTLB size is doubled and if the miss-rate is lower, the size is halved. Their second technique works on the observation that a large percentage of TLB entries are dead (i.e., unlikely to see an access before eviction). They use counters with each entry that are suitably updated on hits and misses. An entry that has not been accessed for a fixed (e.g., 32) inter-miss intervals is considered to be dead and is a candidate for eviction. If no dead entry is found, a candidate is chosen using LRU replacement policy. Thus, by preferentially evicting dead entries, their technique aims to keep the live entries in TLB for longer time. Their techniques provide large energy saving without impacting miss-rate and the second technique is more effective than the first one.

5.5. Designing TLB with non-volatile memory

Liu et al. [31] note that high memory access intensity of GPU workloads along with rare writes to TLB make STT-RAM a suitable candidate for designing GPU TLBs, since STT-RAM has higher density than SRAM, but also higher write latency/energy [85, 86]. Further, since most reads to TLB are directed to few entries, they propose a scheme which seeks to reduce read latency of hot blocks by selectively trading off capacity. This scheme uses "differential sensing" (DS) strategy which reduces read latency but uses two cells to store a value. To find hot data, they note that page size of a data-item is related to its access pattern and thus, PTEs of large pages show higher access frequency than those of small pages. Based on it, at the time of loading the PTE into TLB, a PTE with larger than 4KB page size is stored in DS mode, otherwise it is stored in normal mode. Since this heuristic may not capture a few hot small pages, they use counters to record access frequency and when the counter exceeds a threshold, it is transitioned to DS mode by utilizing an unused entry. As for demotion from DS mode to normal mode, when a TLB entry in DS mode is ready for retirement, it is first converted to normal mode. Only entries in normal mode are directly evicted from the TLB. Their technique improves performance and energy efficiency compared to both an SRAM-based TLB and a simple STT-RAM based TLB.

6. TECHNIQUES FOR DESIGNING VIRTUAL CACHE

Different works use different approaches to address synonym issue in virtual caches. Some works remap accesses directed towards different VAs to a single VA [50, 55], which is called leading [50] or primary [55] and is found based on the first touch (§6.1). Other works propose hybrid virtual cache designs [17, 51], where PA is used for addresses with synonyms and VA is used for non-synonym addresses (§6.2). Other approaches to handle synonyms include maintaining pointers between two levels of cache for invalidating synonyms at the time of replacement [52, 60] (§6.3), using intelligent coherence protocols [29] (§6.4), using a single global VA space shared between applications sharing data (§6.5) and using both virtual and physical tags [17, 80]. We now review some virtual cache designs.

DOI: 10.1002/cpe

6.1. Remapping the accesses to synonyms

Yoon et al. [50] present a virtual L1 cache design which performs dynamic mapping of synonyms, i.e., for a data-block cached with one VA (say V_i), synonymous accesses made to same data-block with a different address (V_j) are remapped to use V_i. They define "active synonyms" of a physical page P as the virtual pages that are synonymous with P in a given time interval T, where T shows the interval where one or multiple blocks from page P are stored in the cache. The group of synonymous pages is called "equivalent page set" (EPS). They note that for L1 caches of typical size (e.g., 32-64KB), the number of pages with active synonyms is small and the number of such active synonyms is also quite small. Based on this, only a small table is required for remembering [V_j, V_i] remapping. Further, at different times in program execution, different VAs from same EPS are leading VAs, and hence, the remapping information needs to be dynamically tracked.

Furthermore, accesses to cache blocks from pages with active synonyms and especially those that are cached with a different VA are both only a small percentage of total accesses for most applications. Hence, the remapping table will be infrequently accessed. Based on this, they use hashed bits of VA to predict the presence of a synonym in the table using another small storage structure and avoid the access to table when no synonym is present. On L1 miss, TLB is accessed with V_i and PA (P_i) is obtained. P_i is used to access another table called "active synonym detection table". In case of no match, a [P_i, V_i] entry is created and V_i is used as the leading VA. On a match, V_k is noted and if $V_i \neq V_k$, an active synonym is detected and an entry is created in the remapping table. Overall, by virtue of using a unique leading VA for accessing a page in cache, their technique simplifies the management of memory hierarchy. They show that their technique reduces TLB energy significantly and provides same latency advantage as an ideal yet infeasible virtual cache.

Qiu et al. [55] present a technique to support virtual caches by tracking synonyms. For a page with synonyms, one VA is termed as primary VA and remaining VAs are termed as secondary VAs. For pages without synonyms, the VA itself is primary VA. For virtual cache, primary VA works as the unique identifier for each page instead of PA. Thus, use of primary VA resolves coherence issues in cache hierarchies of both single and multi-core processors. To store translations of secondary VAs into primary VAs, they use "synonym lookaside buffer" (SLB) with each core. In their design, L1 cache uses VA for index and primary VA for tag. For L2 cache, both index and tag are derived from primary VAs. L2 cache stores backpointers to detect synonyms in L1 cache. On a miss in SLB, L1 cache tag is checked with the VA which could be primary or secondary. L1 cache hit confirms primary VA but on a miss, the block may be in a different set. Hence, L2 cache is accessed and a hit with a backpointer confirms the address to be a primary VA and leads to a miss in L1 cache. Else, the L2 cache or memory returns the data. Ultimately, synonyms are detected at TLB which ascertains between primary or secondary VA. For a secondary VA, secondary/primary translation is stored in the SLB. Since synonyms are rare, SLB is very small and hence, can be accessed sequentially or in parallel with L1 cache. The coverage of SLB is much higher than TLB since every SLB entry represents an entire segment including several pages and thus, coverage of SLB scales with number of cores, application footprints and physical memory sizes. SLB does not record PAs and hence, is unaffected by changes in V2PA mapping. SLB is flushed only on elimination of synonyms which happens much less frequently than changes in V2PA mapping. Also, SLB-supported caches have higher hit rate than TLB-supported caches. They demonstrate that a small SLB of 8-16 entries can adequately address synonym issue and incurs negligible performance overhead.

6.2. Using both PAs and VAs in virtual cache

Park et al. [51] present a hybrid virtual cache design where VAs and PAs are used for non-synonym and synonym pages, respectively. Thus, they use either VA or PA consistently to uniquely identify each physical memory block in all cache levels and this avoids coherence issue. For translation, every VA goes to a "synonym filter" and since synonyms occur infrequently, it is designed using Bloom filter. The filter correctly determines whether an address is synonym, although it may incorrectly determine a non-synonym as synonym. For non-synonyms, VA (and ASID) is used in all cache levels and when memory is required to be referenced, TLB is referenced using ASID+VA

DOI: 10.1002/cpe

to obtain the PA (referred to as "delayed translation"). For addresses predicted to be synonyms by the filter, TLB is accessed, but if the result was false positive, translation is not performed and just ASID+VA is used (refer Figure 15). On remapping of a virtual page or a change in status of a page from private (non-synonym) to shared (synonym), cache blocks of the page are flushed from cache hierarchy, although such events occur infrequently. They show that their technique reduces TLB power significantly.

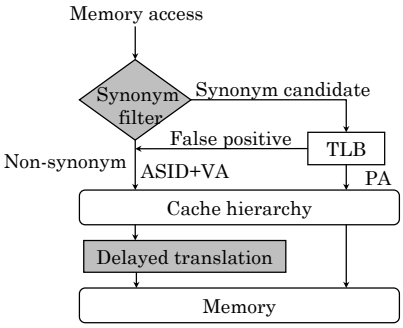


Figure 15. Hybrid virtual cache design using synonym filter [51]

Basu et al. [17] note that in most applications, very few pages have synonyms and they are accessed relatively infrequently. Also, a majority of synonym pages only see read-accesses and hence, cannot generate read/write inconsistency. Further, TLB invalidations due to changes in page mapping or protection/permission happen infrequently and they generally involve all pages of a process and thus, can be easily handled with cache flushes. Based on these, they propose using hybrid virtual caching technique which caches some data with VAs and others with PAs. In their technique, OS ascertains the addresses that have read/write synonyms or frequent changes in mapping/permission, based on flags defined by VA allocator, (e.g., VM_PRIVATE, VM_SHARED) and hints from application. Such addresses use PAs for caching. For remaining addresses, virtual caching is used; TLB is accessed only on L1 cache misses and only half of the L1 cache ways need to be looked up. Since eviction of a dirty L1 cache block requires writeback to physical L2 cache, they add a physical tag along with each virtually-tagged block to smoothly handle such evictions. Also, sequentially searching these physical tags for all the sets that may hold a block alleviates the requirement of PA to VA translation when L1 cache receives coherence messages since the L2 cache uses PAs for handling coherence. When required, their technique defaults to physical caching and thus, provides compatibility with conventional OSes/architectures. Their technique reduces TLB accesses significantly and also saves L1 dynamic energy compared to VI-PT cache.

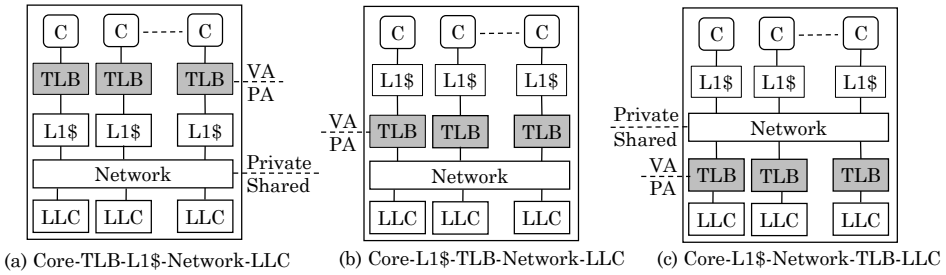
6.3. Designing data filter cache as a virtual cache

Bardizbanyan et al. [52] present the design of a "data filter cache" (DFC) which is accessed before DL1 cache. DFC uses virtual tags which avoids the need of DTLB accesses and allows DFC to be accessed early in the pipeline, e.g., in parallel with the memory address generation. Due to this, DFC miss does not incur penalty whereas a DFC hit avoids load hazards that typically accompany DL1 hits. DL1 cache is inclusive of DFC and each DFC block stores the DL1 cache way where the block is stored and also the page protection information. On replacement of a DFC block, its DL1 cache way and index (obtained from least-significant bits of DFC tag since DL1 cache is virtually-indexed) are compared with those of other DFC blocks and a synonym DFC block is invalidated if present. This addresses synonym problem and leads to negligible overhead due to small size of DFC. DFC uses write-through policy which provides the advantage of simpler implementation and since stores are less frequent than loads, it leads to only minor overhead. On context-switches, DFC is invalidated and this handles homonym issue. They also discuss strategies to access DFC speculatively for load operations and use critical word-first fill scheme for fill operations. Their technique reduces energy consumption of DL1 cache and DTLB and also improves performance.

DOI: 10.1002/cpe

6.4. Using prudent coherence protocols

Kaxiras et al. [29] note that ensuring coherence for virtual L1 cache in multicore processors is challenging since it requires PA to VA address translation for coherence requests sent to virtual cache due to the presence of synonyms. To address this, they propose using a protocol which does not send any request traffic (i.e., downgrades, forwardings or invalidations) to virtual L1 cache. Hence, they use a directoryless protocol which implements "self-invalidation" and "self-downgrade" using synchronization and "data-race-free" semantics. They further consider three options for TLB placement, as shown in Figure 16: (a) "Core-TLB-L1$-Network-LLC" (b) "Core-L1$-TLB-Network-LLC" and (c) "Core-L1$-Network-TLB-LLC". Options (a) and (b) use PT and VT L1 cache, respectively and both options use per-core TLBs. Their technique allows placing the TLB after L1 and network such that it is logically shared and is accessed just before accessing LLC (Figure 16(c)). This avoids the overhead of per-core TLBs that need to be kept coherent. Also, shared TLB provides higher effective capacity by avoiding replications and simplifies shared/private classification. Their technique simplifies coherence while still supporting synonyms. Compared to PT L1 cache, their VT L1 cache (option (c)) reduces TLB accesses and use of a shared TLB helps in reducing TLB misses. Also, compared to use of "modified, exclusive, shared, invalid" (MESI) directory protocol which requires PA to VA translation, their technique provides energy and performance benefits.



(a) Core-TLB-L1$-Network-LLC (b) Core-L1$-TLB-Network-LLC (c) Core-L1$-Network-TLB-LLC

Figure 16. Different options for TLB placement [29]

6.5. Eliminating TLB

Wood et al. [10] propose eliminating TLB and instead using VT data cache for storing PTEs. To avoid synonym problem, they provision that all processes share a single global VA space which must be used by processes sharing data. In their design, PTEs are stored in same cache as data and instructions and are accessed using VAs. For translating a VA, its PTE needs to be fetched for determining the PA. For this, global VA of PTE is computed. By enforcing that all page tables are contiguous in VA space, PTE can be obtained by accessing the page table with VPN as the index. For multicore processors, their approach avoids TLB consistency problem since page tables are cached only in data cache of a core and not TLB. Hence, cached PTEs are updated consistently as directed by the cache coherency policy. Since translation is performed only on cache misses, storing PTEs in cache has minimal impact on its normal behavior and the overall performance. They show that performance of their approach depends crucially on cache miss rate (and hence cache size) and for large sized caches, their approach provides good performance.

Jacob et al. [62] present software-managed address translation as an approach to completely avoid using TLB. They use a VI-VT cache hierarchy and an LLC miss invokes OS cache miss handler which translates the address and fetches the corresponding physical data. They note that with large LLC, the miss rates are small and hence, translations are required infrequently. Further, since the LLC miss is handled by the memory manager, OS has the flexibility to implement any V2PA mapping, replacement policy, protection scheme or software-defined page size. Hence, their approach allows defining superpages, shared memory and fine-grained protection entirely in software which offers higher flexibility than hardware-based implementations. They show that for large LLC and suitable block size ($\geq$ 64B), their approach performs close to hardware-based

DOI: 10.1002/cpe

approaches using TLB. The performance of their technique is highly sensitive to cache size and block size, whereas that of hardware-based approaches is sensitive to TLB configuration.

7. TLB MANAGEMENT IN GPUS AND CPU-GPU SYSTEMS

While conventional CPU-GPU heterogeneous systems used separate VA and PA spaces for CPU and GPU, recent systems are moving towards fully unified address spaces since they facilitate programming and reduce data transfer overhead by making data objects globally accessible from both CPU and GPU. However, the fundamental differences in CPU and GPU architecture, along with the requirement of unified address spaces necessitate detailed study of address translation on GPU. We now review TLB design proposals for GPUs and CPU-GPU systems.

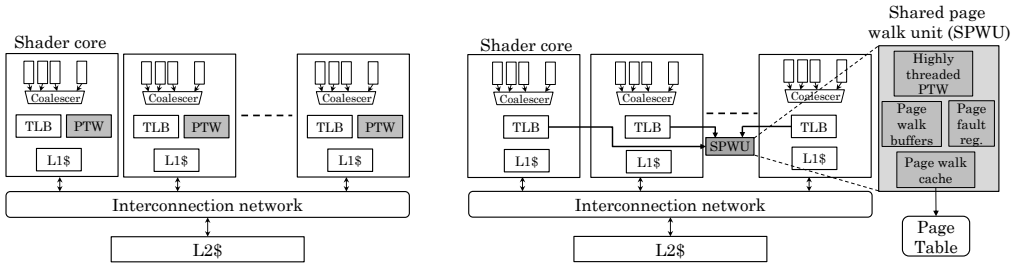
Vesely et al. [24] analyze shared VM across CPU and GPU on a commercial integrated heterogeneous processor. They observe a TLB miss on GPU to have nearly 25× higher latency than that on the CPU. This is because TLB miss requests and responses are communicated to IOMMU over PCIe ("peripheral component interconnect express") and the PTW of IOMMU cannot access CPU caches which may store the latest PTEs. Also, a miss necessitates rescheduling of the warp (or wavefront). To hide such latency, techniques for improving concurrency in servicing TLB misses are vital. They also note that up to 16 simultaneous page table accesses may be allowed by IOMMU and beyond this, latency of page table access increases quickly from queuing effect. They further run applications with both 4KB and 2MB page sizes. Use of 2MB page size reduces TLB misses significantly, however, it translates to improved performance in only some applications. Furthermore, divergence in memory access pattern has larger impact on address translation overhead than cache and memory latencies. Also, effectiveness of TLB prefetching depends on locality in application's memory access pattern. The TLB shootdown latency on GPU is comparable to that on CPU and the latency of invalidating a single GPU TLB entry is nearly same as that of flushing entire GPU TLB.

Pichai et al. [25] note that designing a GPU TLB similar to CPU TLB leads to significant performance loss compared to a design without TLB. This GPU TLB design is: 128-entry, 3-port, per-core TLB which is blocking (i.e., on TLB miss, another warp is switched-in which continues execution if it is without memory instruction). Also, TLB and PTW are accessed in parallel with cache. The reason for performance loss is that GPU's "round robin warp scheduling" interleaves memory accesses of different threads which increases cache and TLB misses. Also, due to the lock-step execution, a TLB miss in a thread also stalls other warp threads. Further, TLB misses from different warp threads generally occur in close proximity which are serialized by PTWs. They propose simple extensions to TLB and PTW to alleviate this performance loss. First, increasing port-count from 3 to 4 recovers most of the loss since the coalescing unit before TLB merges requests to the same PTE in a single access and hence, most warps require much fewer than 32 translations.

Second, they propose two non-blocking TLB designs: (1) switched-in warp continues even if it has memory instruction as long as they lead to TLB hits. On TLB miss, the warp is switched out. (2) threads hitting in TLB access the cache without stalling for another thread in the same warp that sees TLB miss. This improves cache hit rate since the data of this warp is likely to be present in cache before being evicted by that of the switched-in warp. The design (2) is applied in conjunction with design (1). They note that these two designs improve the performance significantly. They also propose PTW scheduling strategies and by using above mentioned strategies together, the performance approaches that of an ideal (but infeasible) 32-port, 512-entry TLB without access latency overhead. Their proposed design is illustrated in Figure 17(a). They also demonstrate that integrating TLB-awareness into warp scheduling and dynamic warp construction schemes can bring the overhead of including TLB in acceptable range.

Power et al. [26] present a GPU MMU architecture that has compatibility with (x86-64) CPU page tables. Their findings are based on insights obtained from executing Rodinia benchmark. Their baseline design is an "ideal MMU" which has infinite size per-core TLBs and incurs 1 cycle for page walks. Their first design uses a per-core private MMU which consumes large power and also lies on critical path. Their second design moves TLB to after coalescing unit which lowers TLB

DOI: 10.1002/cpe



(a) Design with per-core TLBs and PTWs [25]

(b) Design with per-core TLBs, shared highly-threaded PTW and shared page walk cache (reg. = register) [26]

Figure 17. MMU designs proposed for GPU. Pichai et al. [25] propose using per-core PTWs whereas Power et al. [26] propose a multi-threaded shared PTW design with a page walk cache.

accesses greatly (e.g., by 85%). They note that coalescing unit reduces memory accesses greatly and hence, their second design uses per-core private L1 TLB after scratchpad memory access and after coalescing unit. Thus, only global memory references lead to access to MMU. However, they note that since GPU is highly bandwidth-intensive, on average 60 and up to 1000 page table walks may be active at any time in each core and hence, traditional blocking PTW designs lead to large queuing delays. Due to this, the second design has poor performance.

Hence, their third design uses a multi-threaded PTW with 32 threads that is shared between all GPU cores. On a TLB miss in the per-core L1 TLB, a page table access is performed in the shared PTW. On average, the third design performs similar to the second design and provides nearly 30% of performance of the ideal MMU. The reason for poor performance of the third design is queuing of requests from one core before those of other cores. Also, GPU TLB miss rates are much higher than those of CPU TLB. To reduce the TLB miss latency, their fourth design adds a page walk cache to shared PTW which reduces latency of page table walks and stall time of cores. Thus, by virtue of avoiding queuing delays, this design achieves almost same performance as the ideal MMU design. This fourth design is their recommended design and is illustrated in Figure 17(b). They also discuss handling of page faults, TLB flushing and shootdown. For several applications, degree of sharing of a TLB entry between cores is low and hence, a shared L2 TLB does not improve performance and adding a page walk cache is more effective than a shared L2 TLB. Similarly, a TLB prefetcher provides only marginal improvement in performance. They also observe that use of large pages (e.g., 2MB) reduces TLB miss rate, however, due to requirement of maintaining compatibility with CPU page tables, use of large pages alone may not solve the performance issue.

8. CONCLUDING REMARKS

In this paper, we presented a survey of techniques for designing and optimizing TLBs. We presented terminology and a background on TLB and underscored tradeoffs in TLB management. We reviewed workload characterization studies on TLB, techniques for reducing TLB latency, miss rate and energy consumption and virtual cache designs for reducing accesses to TLB. We highlighted key attributes of the research works to emphasize their similar and distinct features. We now briefly summarize some future research challenges.

As evident from Table I, most existing TLB management techniques have been proposed for CPUs. It is unclear whether these techniques will be effective and even applicable for GPUs and CPU-GPU systems [87]. Exploring the emerging design space of GPUs and heterogeneous systems while maximally utilizing the conventional wisdom on TLB management will be vital for the future research studies.

Given the large simulation time required for evaluating the complete design space of TLBs, many works only measure TLB misses and not performance [8, 9, 67, 79], whereas others estimate performance by analytical models [11, 13, 40, 77]. However, this approach may not accurately

DOI: 10.1002/cpe

model the performance impact of a TLB management technique. Going forward, developing high-speed and versatile simulation platforms will be essential to boost future research on TLBs.

Recent trends such as increasing system core-count and application working set sizes, virtualization, etc. make management of TLB even more important for future systems. In our opinion, a synergistic approach at all levels of abstraction is required for meeting this challenge. At circuit-level, non-volatile memory and innovative array organizations can lower area/energy overhead. At architecture-level, techniques such as TLB reconfiguration, superpaging, prefetching, etc. can be combined to bring together the best of them. At compiler-level, reorganizing application's memory access pattern in TLB-friendly manner can enhance the effectiveness of architectural techniques of TLB management. Finally, the inherent error-tolerance of applications can be leveraged for aggressive optimization such as speculating the translation and operating TLB at near-threshold voltage for energy saving.

REFERENCES

[1] Li Y, Melhem R, Jones AK. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future CMPs. *ACM Transactions on Architecture and Code Optimization (TACO)* 2013; **9**(4):28.

[2] Talluri M, Hill MD. Surpassing the TLB performance of superpages with less operating system support. *ASPLOS* 1994; .

[3] Peng J, Lueh GY, Wu G, Gou X, Rakvic R. A comprehensive study of hardware/software approaches to improve TLB performance for java applications on embedded systems. *Workshop on Memory System Performance and Correctness*, 2006; 102–111.

[4] Cekleov M, Dubois M. Virtual-address caches. part 1: problems and solutions in uniprocessors. *IEEE Micro* 1997; **17**(5):64–71.

[5] Karakostas V, Gandhi J, Hill MD, McKinley KS, Nemirovsky M, Swift MM, Unsal OS, *et al.*. Energy-efficient address translation. *International Symposium on High Performance Computer Architecture (HPCA)*, 2016; 631–643.

[6] Papadopoulou MM, Tong X, Seznec A, Moshovos A. Prediction-based superpage-friendly TLB designs. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015; 210–222.

[7] Swanson M, Stoller L, Carter J. Increasing TLB reach using superpages backed by shadow memory. *ISCA*, 1998; 204–213.

[8] Juan T, Lang T, Navarro JJ. Reducing TLB power requirements. *ISLPED*, 1997; 196–201.

[9] Taylor G, Davies P, Farmwald M. The TLB slicea low-cost high-speed address translation mechanism. *ISCA* 1990; :355–363.

[10] Wood DA, Eggers SJ, Gibson G, Hill MD, Pendleton JM. An in-cache address translation mechanism. *ACM SIGARCH Computer Architecture News*, vol. 14, 1986; 358–365.

[11] Talluri M, Kong S, Hill MD, Patterson DA. Tradeoffs in supporting two page sizes. *ISCA*, 1992; 415–424.

[12] Navarro J, Iyer S, Druschel P, Cox A. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review* 2002; **36**(SI):89–104.

[13] Chen JB, Borg A, Jouppi NP. A Simulation Based Study of TLB Performance. *ISCA* 1992; :114–123.

DOI: 10.1002/cpe

[14] Romanescu BF, Lebeck AR, Sorin DJ, Bracy A. UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010; 1–12.

[15] Bhattacharjee A, Martonosi M. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. *PACT*, 2009; 29–40.

[16] Kadayif I, Nath P, Kandemir M, Sivasubramaniam A. Reducing data TLB power via compiler-directed address generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2007; **26**(2):312–324.

[17] Basu A, Hill MD, Swift MM. Reducing memory reference energy with opportunistic virtual caching. *ACM SIGARCH Computer Architecture News*, vol. 40, 2012; 297–308.

[18] Xue J, Thottethodi M. PreTrans: reducing TLB CAM-search via page number prediction and speculative pre-translation. *International Symposium on Low Power Electronics and Design*, 2013; 341–346.

[19] Bhattacharjee A, *et al.*. Shared Last-Level TLBs for Chip Multiprocessors. *HPCA*, 2011.

[20] Chen L, Wang Y, Cui Z, Huang Y, Bao Y, Chen M. Scattered superpage: A case for bridging the gap between superpage and page coloring. *International Conference on Computer Design (ICCD)*, 2013; 177–184.

[21] Pham B, Vaidyanathan V, Jaleel A, Bhattacharjee A. CoLT: Coalesced large-reach TLBs. *International Symposium on Microarchitecture*, 2012; 258–269.

[22] Lowe E. Translation Storage Buffers. https://blogs.oracle.com/elowe/entry/translation_storage_buffers 2005.

[23] Bhattacharjee A. Large-reach memory management unit caches. *International Symposium on Microarchitecture*, 2013; 383–394.

[24] Vesely J, Basu A, Oskin M, Loh GH, Bhattacharjee A. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016; 161–171.

[25] Pichai B, Hsu L, Bhattacharjee A. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. *ASPLOS*, 2014; 743–758.

[26] Power J, Hill MD, Wood DA. Supporting x86-64 address translation for 100s of GPU lanes. *International Symposium on High Performance Computer Architecture (HPCA)*, 2014; 568–578.

[27] Karakostas V, Gandhi J, Ayar F, Cristal A, Hill MD, McKinley KS, Nemirovsky M, Swift MM, Ünsal O. Redundant memory mappings for fast access to large memories. *International Symposium on Computer Architecture (ISCA)*, 2015; 66–78.

[28] Kadayif I, Sivasubramaniam A, Kandemir M, Kandiraju G, Chen G. Optimizing instruction TLB energy using software and hardware techniques. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 2005; **10**(2):229–257.

[29] Kaxiras S, Ros A. A new perspective for efficient virtual-cache coherence. *ISCA*, 2013; 535–546.

[30] Mittal S. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys* 2016; .

DOI: 10.1002/cpe

[31] Liu X, Li Y, Zhang Y, Jones AK, Chen Y. STD-TLB: A STT-RAM-based dynamically-configurable translation lookaside buffer for GPU architectures. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014; 355–360.

[32] Bala K, Kaashoek MF, Weihl WE. Software prefetching and caching for translation lookaside buffers. *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, USENIX Association, 1994; 18.

[33] Villavieja C, Karakostas V, Vilanova L, Etsion Y, Ramirez A, Mendelson A, Navarro N, Cristal A, Unsal OS. DiDi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011; 340–349.

[34] Pham B, Veselỳ J, Loh GH, Bhattacharjee A. Large pages and lightweight memory management in virtualized environments: can you have it both ways? *International Symposium on Microarchitecture*, 2015; 1–12.

[35] Nagle D, Uhlig R, Stanley T, Sechrest S, Mudge T, Brown R. Design tradeoffs for software-managed TLBs. *ACM SIGARCH Computer Architecture News*, vol. 21, 1993; 27–38.

[36] McCurdy C, Coxa AL, Vetter J. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. *ISPASS*, 2008; 95–104.

[37] Pham B, Bhattacharjee A, Eckert Y, Loh GH. Increasing TLB reach by exploiting clustering in page translations. *International Symposium on High Performance Computer Architecture (HPCA)*, 2014; 558–567.

[38] Romer TH, Ohlrich WH, Karlin AR, Bershad BN. Reducing TLB and memory overhead using online superpage promotion. *ACM SIGARCH Computer Architecture News*, vol. 23, 1995; 176–187.

[39] Lustig D, Bhattacharjee A, Martonosi M. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)* 2013; **10**(1):2.

[40] Lee JH, Lee JS, Jeong SW, Kim SD. A banked-promotion TLB for high performance and low power. *International Conference on Computer Design*, 2001; 118–123.

[41] Balasubramonian R, Albonesi DH, Buyuktosunoglu A, Dwarkadas S. A dynamically tunable memory hierarchy. *IEEE transactions on computers* 2003; **52**(10):1243–1258.

[42] Fang Z, Zhao L, Jiang X, Lu Sl, Iyer R, Li T, Lee SE. Reducing cache and TLB power by exploiting memory region and privilege level semantics. *Journal of Systems Architecture* 2013; **59**(6):279–295.

[43] Choi JH, Lee JH, Jeong SW, Kim SD, Weems C. A low power TLB structure for embedded systems. *IEEE Computer Architecture Letters* 2002; :3.

[44] Ballapuram CS, Lee HHS. Improving TLB energy for Java applications on JVM. *SAMOS*, 2008; 218–223.

[45] Delaluz V, Kandemir M, Sivasubramaniam A, Irwin MJ, Vijaykrishnan N. Reducing dTLB energy through dynamic resizing. *ICCD*, 2003; 358–363.

[46] Chang YJ. An ultra low-power TLB design. *DATE*, 2006; 1122–1127.

[47] Lee HHS, Ballapuram CS. Energy efficient D-TLB and data cache using semantic-aware multilateral partitioning. *International symposium on Low power electronics and design*, 2003; 306–311.

DOI: 10.1002/cpe

[48] Jeyapaul R, Shrivastava A. Code transformations for TLB power reduction. *International journal of parallel programming* 2010; **38**(3-4):254–276.

[49] Jeyapaul R, Shrivastava A. B2P2: bounds based procedure placement for instruction TLB power reduction in embedded systems. *International Workshop on Software & Compilers for Embedded Systems*, 2010; 2.

[50] Yoon H, Sohi GS. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. *International Symposium on High Performance Computer Architecture*, 2016; 212–224.

[51] Park CH, Heo T, Huh J. Efficient synonym filtering and scalable delayed translation for hybrid virtual caching. *ISCA* 2016; .

[52] Bardizbanyan A, Själander M, Whalley D, Larsson-Edefors P. Designing a practical data filter cache to improve both energy efficiency and performance. *ACM Transactions on Architecture and Code Optimization (TACO)* 2013; **10**(4):54.

[53] Hsia A, Chen CW, Liu TJ. Energy-efficient synonym data detection and consistency for virtual cache. *Microprocessors and Microsystems* 2016; **40**:27–44.

[54] Ballapuram CS, Lee HHS, Prvulovic M. Synonymous address compaction for energy reduction in data TLB. *ISLPED*, 2005; 357–362.

[55] Qiu X, Dubois M. The synonym lookaside buffer: A solution to the synonym problem in virtual caches. *IEEE Transactions on Computers* 2008; **57**(12):1585–1599.

[56] Kandemir M, Kadayif I, Chen G. Compiler-directed code restructuring for reducing data TLB energy. *International Conference on Hardware/software Codesign and System Synthesis*, 2004; 98–103.

[57] Haigh JR, Wilkerson MW, Miller JB, Beatty TS, Strazdus SJ, Clark LT. A low-power 2.5-GHz 90-nm level 1 cache and memory management unit. *IEEE journal of solid-state circuits* 2005; **40**(5):1190–1199.

[58] Ballapuram C, Puttaswamy K, Loh GH, Lee HHS. Entropy-based low power data TLB design. *International conference on Compilers, architecture and synthesis for embedded systems*, 2006; 304–311.

[59] Kadayif I, Sivasubramaniam A, Kandemir M, Kandiraju G, Chen G. Generating physical addresses directly for saving instruction TLB energy. *International Symposium on Microarchitecture*, 2002; 185–196.

[60] Wang WH, Baer JL, Levy HM. Organization and performance of a two-level virtual-real cache hierarchy. *ISCA* 1989; :140–148.

[61] Kandiraju GB, Sivasubramaniam A. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. *SIGMETRICS*, 2002; 129–139.

[62] Jacob B, Mudge T. Uniprocessor virtual memory without TLBs. *IEEE Transactions on Computers* 2001; **50**(5):482–499.

[63] Barr TW, Cox AL, Rixner S. SpecTLB: a mechanism for speculative address translation. *International Symposium on Computer Architecture (ISCA)*, 2011; 307–317.

[64] Seznec A. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Transactions on Computers* 2004; **53**(7):924–927.

DOI: 10.1002/cpe

[65] Park CH, Chung J, Seong BH, Roh Y, Park D. Boosting superpage utilization with the shadow memory and the partial-subblock TLB. *International Conference on Supercomputing*, 2000; 187–195.

[66] Jacob BL, Mudge TN. A look at several memory management units, TLB-refill mechanisms, and page table organizations. *ACM SIGOPS Operating Systems Review*, vol. 32, 1998; 295–306.

[67] Park JS, Ahn GS. A software-controlled prefetching mechanism for software-managed TLBs. *Microprocessing and Microprogramming* 1995; **41**(2):121–136.

[68] Austin TM, Sohi GS. High-bandwidth address translation for multiple-issue processors. *ISCA*, 1996; 158–167.

[69] Strecker WD. VAX-11/780: A virtual address extension to the DEC PDP-11 family. *AFIPS*, 1978.

[70] Chukhman I, Petrov P. Context-aware TLB preloading for interference reduction in embedded multi-tasked systems. *Great lakes symposium on VLSI*, 2010; 401–404.

[71] Tong X, Moshovos A. BarTLB: Barren page resistant TLB for managed runtime languages. *International Conference on Computer Design (ICCD)*, 2014; 270–277.

[72] Esteve A, Ros A, Gómez ME, Robles A, Duato J. Efficient TLB-based detection of private pages in chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 2016; **27**(3):748–761.

[73] Bhattacharjee A, Martonosi M. Inter-core cooperative TLB for chip multiprocessors. *ASPLOS* 2010; :359–370.

[74] Srikantaiah S, Kandemir M. Synergistic TLBs for high performance address translation in chip multiprocessors. *International Symposium on Microarchitecture*, 2010; 313–324.

[75] Chiueh Tc, Katz RH. Eliminating the address translation bottleneck for physical address cache. *ACM SIGPLAN Notices*, vol. 27, 1992; 137–148.

[76] Venkatasubramanian G, Figueiredo RJ, Illikkal R. On the performance of tagged translation lookaside buffers: A simulation-driven analysis. *MASCOTS*, IEEE, 2011; 139–149.

[77] Saulsbury A, Dahlgren F, Stenström P. Recency-based TLB preloading. *ISCA* 2000; .

[78] Kandiraju GB, Sivasubramaniam A. Going the Distance for TLB Prefetching: An Application-driven Study. *International Symposium on Computer Architecture*, 2002; 195–206.

[79] Venkatasubramanian G, Figueiredo RJ, Illikkal R, Newell D. TMT: A TLB tag management framework for virtualized platforms. *International Journal of Parallel Programming* 2012; **40**(3):353–380.

[80] Goodman JR. Coherency for multiprocessor virtual address caches. *ASPLOS*, 1987; 72–81.

[81] Seznec A. A case for two-way skewed-associative caches. *ACM SIGARCH Computer Architecture News*, vol. 21, 1993; 169–178.

[82] Mittal S, Poremba M, Vetter J, Xie Y. Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool. *Technical Report ORNL/TM-2014/636*, Oak Ridge National Laboratory, USA 2014.

[83] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/ 2016.

[84] Arcangeli A. Transparent hugepage support. *KVM Forum*, vol. 9, 2010.

DOI: 10.1002/cpe

[85] Mittal S, Vetter JS. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016; **27**(5):1537–1550.

[86] Vetter J, Mittal S. Opportunities for nonvolatile memory systems in extreme-scale high performance computing. *Computing in Science and Engineering (CiSE)* 2015; **17**(2):73 – 82.

[87] Mittal S, Vetter J. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys* 2015; **47**(4):69:1–69:35.

DOI: 10.1002/cpe