PS-TLB: Leveraging Page Classification Information for Fast, Scalable and Efficient Translation for Future CMPs

YONG LI, RAMI MELHEM, and ALEX K. JONES, University of Pittsburgh

Traversing the page table during virtual to physical address translation causes pipeline stalls when misses occur in the translation-lookaside buffer (TLB). State-of-the-art translation proposals typically optimize a single aspect of translation performance (e.g., translation sharing, context switch performance, etc.) with potential trade-offs of additional hardware complexity, increased translation latency, or reduced scalability. In this article, we propose the *partial sharing TLB* (PS-TLB), a fast and scalable solution that reduces off-chip translation misses without sacrificing the timing-critical requirement of on-chip translation. We introduce the *partial sharing buffer* (PSB) which leverages application page sharing characteristics using minimal additional hardware resources. Compared to the leading TLB proposal that leverages sharing, PS-TLB provides a more than 45% improvement in translation latency with a 9% application speedup while using fewer storage resources. In addition, the page classification and PS-TLB architecture provide further optimizations including an over 30% reduction of interprocessor interrupts for coherence, and reduced context switch misses with fewer resources compared with existing methods.

Categories and Subject Descriptors: B.3 [Hardware]: Memory Structures

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Shared TLB, tagged TLB, translation classification, shutdown downgrade

ACM Reference Format:

Li, Y., Melhem, R., and Jones, A. K. 2013. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future CMPs. *ACM Trans. Architect. Code Optim.* 9, 4, Article 28 (January 2013), 21 pages.

DOI = 10.1145/2400682.2400687 <http://doi.acm.org/10.1145/2400682.2400687>

1. INTRODUCTION

Translation-lookaside buffers (TLBs) have been shown effective in accelerating virtual to physical address translation. They are a standard component of commodity CPU products, including multicore systems such as AMD Opteron and Intel Sandy/Ivy Bridge processors. To date, these processors use multilevel per-core TLBs that are exclusive and private to each core. The primary reason for this private design is due to the timing-critical nature of the TLB, which is on the critical path of cache/memory accesses. In addition, private TLBs simplify translation lookup in multi-program environments. In multicore systems, private TLBs also lend themselves to lightweight methods of addressing TLB consistency problems. For example, in contrast to caches utilizing complicated coherence protocols, translation coherency can be maintained using

This work is supported by the National Science Foundation under grant CCF-1064976.

Authors' addresses: Y. Li (corresponding author), Department of ECE, University of Pittsburgh, Pittsburgh, PA; email: yol126@pitt.edu; R. Melhem, Computer Science Department, University of Pittsburgh, Pittsburgh, PA; A. K. Jones, Department of ECE, University of Pittsburgh, Pittsburgh, PA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART28 \$15.00

DOI 10.1145/2400682.2400687 <http://doi.acm.org/10.1145/2400682.2400687>

TLB shutdown, a process that evicts invalid TLB entries using Inter-Processor Interrupts (IPIs) [Villavieja et al. 2011]. This mechanism leverages the infrequency of translation entry changes in the TLB compared to data changes in the memory system.

Unfortunately, the merits of private TLBs are often traded for an increased translation miss rate. This increase is typically due to poor TLB capacity utilization of private TLBs from replicated entries. Additionally, the inability to locate entries cached in remote TLBs due to the lack of coherence results in potentially unnecessary TLB misses. These trade-offs are particularly undesirable for architectures such as Intel IA-32, in which filling up a TLB entry upon a miss may require up to four memory accesses traversing a four-level hierarchical page table structure, which is expensive.

In addition to the above issues, traditional translation operations such as TLB shutdowns [Black et al. 1989] and TLB flushes caused by context switches exacerbate these inefficiencies. These operations create significant off-chip translation misses, further adding to the aggregate translation penalty.

Thus, to provide a scalable alternative to a physically shared TLB without requiring the latency and storage overhead of a tagged solution [Venkatasubramanian et al. 2011], we propose a *partial sharing TLB (PS-TLB)*. PS-TLB extends traditional private TLBs with a small *partial sharing buffer (PSB)*. Assisted by a lightweight runtime translation classification scheme, private translations are placed locally for low-latency access and shared translations are distributed across all cores' PSBs in a similar fashion as nonuniform access memories.

The PS-TLB provides a significant performance improvement over state-of-the-art second-level TLB techniques that leverage translation sharing while reducing storage and runtime overheads. PS-TLB also optimizes other TLB functions. For example, using PS-TLB, TLB shutdowns can often be improved by downgrading them to individual invalidations, preventing stalls in cores not storing that translation. Additionally, like tagged TLBs, PS-TLB can reduce the impact from flushing required in private TLBs during a context switch, by retaining entries in the PSBs, which do not require flushing. As such, we demonstrate that the proposed PS-TLB reduces several categories of translation misses and overhead, as classified below:

- TLB capacity misses* Private TLBs duplicate requested entries that are shared by multiple cores, leading to capacity misses for workload sizes that exceed the capacity of local TLBs.
- TLB sharing misses* Even with adequate capacity, a private TLB structure still suffers unnecessary misses since it is not aware of nonlocal requested entries already cached in other processor TLBs.
- Additional overheads* TLB shutdown may unnecessarily stall all cores to invalidate a private page and TLB flushing may unnecessarily evict translations from multiple processes running on the same core.

We evaluate the impact of PS-TLB on translation latency, translation miss rate, system performance, TLB shutdown, and TLB flush effects. We demonstrate a 45% latency reduction and an application performance improvement of 9% compared to the state-of-the-art TLB mechanism that leverages sharing [Bhattacharjee et al. 2011]. In addition, we quantitatively demonstrate the superiority of the proposed mechanism over a state-of-the-art distributed/scalable prefetching-based TLB organization [Bhattacharjee and Martonosi 2010] in terms of both performance and efficiency. We conduct sensitivity analyses to show PS-TLB is effective for various size L2 TLBs and how performance scales with the size of the sharing buffer. Finally, we show that more than 30% of coherence shutdowns can avoid an interruption all cores and context switch misses and latency are improved over tagged methods.

Fig. 1. Application page sharing characteristics.

The remainder of the article is organized as follows: We further motivate the need for PS-TLB and enumerate our contributions in more detail in Section 2. Section 3 describes address translation background and related efforts for improving TLB performance. Section 4 introduces our proposed translation architecture as well as a set of optimized translation operations. We evaluate the efficiency and performance of the proposed mechanism in Section 5. Finally, we draw conclusions and describe ongoing and future work of this effort in Section 6.

2. MOTIVATION

Address translation and TLB handling are known to consume a considerable amount of system running time [Clark and Emer 2000; Uhlig et al. 1994; Rosenblum et al. 1995]. This can be attributed to the need for address translation to occur in the critical path for all memory accesses (including instructions). With uniprocessor systems, researchers have shown that TLB-related overhead can be as high as 40% of the total running time [Huck and Hays 1993] and a wide range of literature has been produced to mitigate this potential bottleneck.

With the advent of shared memory chip-multiprocessors many assumptions from uniprocessor TLBs lead to new inefficiencies. For example, multithreaded parallel workloads will exhibit ubiquitous sharing. Thus, an entirely private per-core TLB structure leads to the potential for severe performance degradation due to avoidable misses from poor capacity utilization of the private TLBs. This poor utilization is caused by replication of the same translation in multiple tiles. Figure 1 shows that for representative multithreaded benchmarks [Blackburn et al. 2006; Arnold et al. 1992; Bienia et al. 2008], an average of 62% of all pages are heavily shared (i.e., with three or more sharers) and only 29% of them are private.

Recently, a physically shared (or centralized shared) last-level TLB has been shown to improve TLB translation latency in a multicore context with four cores [Bhattacharjee et al. 2011]. Unfortunately, a physically shared solution is not expected to scale well to a large number of cores. Scaling will be hampered by increases in end-to-end latency of accessing a shared TLB and the increased pressure on a shared L2 TLB caused by a higher aggregate of L1 TLB misses due to adding more cores into the system.

A potentially scalable solution is to use a static nonuniform TLB access architecture similar to the concept used in last-level caches [Kim et al. 2003]. This requires adding process ID tags to the L2 TLBs [Venkatasubramanian et al. 2011], a technique already proposed to avoid flushing the TLB on a context switch. We describe the details of this approach further in Section 4.1. For 16 cores, the access latency of such a distributed shared TLB is compared with a physically shared approach, which also requires tags, in Figure 2. In most cases the distributed shared TLB latency is higher due to the

Fig. 2. Translation latency for a L2 TLB using a nonuniform access shared TLB model compared with a centralized shared approach for 16 cores (normalized to centralized shared).

latency of using the network-on-chip. This doubles the latency on average¹. Thus, as the physically shared solution outperforms both private TLBs [Bhattacharjee et al. 2011] and distributed shared TLBs (Figure 2) we utilize this as our baseline for performance comparison in the rest of the article.

This analysis in part demonstrates that the creation of an efficient and scalable L2 TLB architecture that leverages sharing of translations is a difficult problem. However, the scalability of the physically shared solution is not expected to reach into many-core architectures. Recognizing this concern, researchers from the same group that proposed the physically shared L2 TLB solution also proposed a distributed solution that uses prediction to prefetch shared translations from other private TLBs [Bhattacharjee and Martonosi 2010]. Our proposed PS-TLB is scalable, uses less resources, and outperforms both the leading shared [Bhattacharjee et al. 2011] and distributed [Bhattacharjee and Martonosi 2010] solutions.

In particular, we make the following contributions in this article.

- We conduct an analysis using over 20 benchmarks of the inefficiencies of using purely traditional private TLBs and shared TLBs. We demonstrate that fast, efficient, and scalable translation cannot be achieved without considering translation classification in TLB designs.
- We describe the PS-TLB, which combines private and shared translation classification with an efficient translation architecture. PS-TLB offers inter-core translation sharing for shared translations to reduce TLB misses while preserving the low-latency feature of private TLBs to satisfy the timing-critical requirement of on-chip translations.
- We demonstrate a significant performance improvement of PS-TLB over the state-of-the-art methods to leverage page sharing in TLBs while also reducing complexity overheads.
- We develop and support efficient TLB operations for dealing with coherence and mitigating the impact of context switching as components of PS-TLB.

3. BACKGROUND AND RELATED WORK

To frame our proposed PS-TLB and previous work we begin with an overview of a typical translation architecture found in commodity Chip-MultiProcessors (CMPs). We then provide a discussion of research efforts related to improving translation performances.

¹For this comparison we used the previously proposed access latency for a four-core system [Bhattacharjee et al. 2011] which is optimistic for 16 cores. A detailed list of parameters is contained in Section 5.

Fig. 3. Baseline architecture with 2-level TLB translation and hierarchical page table.

3.1. Background

In our overview of TLB function in CMPs we describe a basic TLB architecture, and a method for translation and standard approaches for maintaining consistency.

3.1.1. Address Translation Architecture. Figure 3 illustrates the address translation flow on a typical CMP microarchitecture in which each processing node consists of a CPU, a coherence directory, a MMU (memory management unit), a network interface, L1/L2 caches, and L1/L2 TLBs. The TLB is organized as a per-core hierarchical structure that features separate L1 instruction and data TLBs backed by a *unified* L2 TLB, serving both instruction and data translations. As in the most common case of virtually indexed physically tagged caches, the virtual address (VA) issued from the CPU needs to be translated before the requested data can be accessed from either the caches or main memory. The resolved translations are cached in the local TLBs to accelerate further translation requests.

3.1.2. Address Translation Basics. As the MMU handles a translation request, it first looks up the virtually indexed TLB hierarchy using the virtual address. Upon a last-level miss, either a hardware page table walker or a software TLB miss handler will be invoked to traverse all levels of the page table hierarchy for the target physical page number (PPN), as illustrated on the right-hand side of Figure 3. The hardware mechanism, as adopted in the Intel x86 architecture, usually offers a performance benefit by preventing the pipeline from being polluted by the execution of a miss handler's code as occurs in a software-managed TLB. Software approaches, on the other hand, reduce hardware complexity and enable more flexible page table structures that are largely independent of the underlying architectures (e.g., MIPS, SPARC). In both cases, a page table walk that traverses several page table hierarchies to locate the requested translation from main memory incurs significant overhead. Even if all page table entries are present in the L2 data cache, accessing them in a daisy chain fashion still incurs a penalty of several tens of cycles per TLB miss [Barr et al. 2010].

3.1.3. Address Translation Consistency. To ensure the translation consistency in CMPs, certain TLB operations are typically performed in response to modification of PTEs. One such operation is TLB shutdown [Black et al. 1989], which is necessary in scenarios where unsafe changes [Romanescu et al. 2010] take place (e.g., page remapping, page swapping, decreasing page privileges, etc.). TLB shutdown is also recommended even for safe changes (e.g., dirty/access bit reset, privilege increasing, etc.) to avoid undesirable consequences. For example, choosing not to shutdown an entry upon the OS

resetting the dirty bit in the corresponding PTE may result in the processor not setting the dirty bit again in response to a subsequent write access to the corresponding page. Consequently, the software cannot rely on the dirty bit being set as an indication that the page is dirty. Shootdowns typically require all cores be stalled while the shootdown takes place even if the changed TLB entry is not stored within many cores' local TLB.

Another important operation is TLB flush, an operation that invalidates all except global entries upon context switches. Flushing the TLB is necessary since TLB entries are indexed by virtual addresses from the local running process, which might overlap with those from another process running on the same core.

In summary, there are several inefficiencies from standard private TLBs used in CMP architectures. These inefficiencies include the inability to leverage shared TLB entries stored in remote private TLBs, necessity to flush TLB entries on a context switch, and system-wide stalls related to coherence. In the next section we cite relevant prior work that provides proposed solutions to these problems.

3.2. Related Work

The TLB design space for uniprocessor systems has been heavily studied. For example, many efforts have explored methods to improve performance by optimizing various core TLB parameters such as size, set associativity, etc. [Chen et al. 1992; Uhlig et al. 1994]. In particular, in this context, Chen et al. [1992] investigated a second-level TLB and analyzed the potential performance impact of variable sized pages.

Several attempts have been made to improve TLB inefficiencies due to multiprocess workloads. Prior research efforts indicate that “tagged” TLB sharing can be accomplished by adding tags (e.g., context ID, process ID (PID) or address space ID (ASID)) to the private TLBs in order to associate TLB entries with specific processes [Venkatasubramanian et al. 2011; Bhattacharjee and Martonosi 2010]. This prevents the need to flush the TLB upon a context switch. Unfortunately, the storage overhead from adding tags can be significant, potentially requiring more than 25% additional storage. To reduce the coherence overhead of shootdowns, Villavieja et al. [2011] extend the “tagged” concept to include a coherence directory to alleviate the need TLB shootdown overhead.

In particular for CMPs, Bhattacharjee et al. recently presented a study of TLB sharing characterization [Bhattacharjee and Martonosi 2009] that motivated a number of research efforts including Synergistic TLBs [Srikantaiah and Kandemir 2010], which utilizes victim allocation and migration to improve the TLB hit ratio, similar to techniques applied to last-level caches. Synergistic TLBs leverage victim entries in remote TLBs to emulate a distributed shared TLB for increased capacity. Specifically, processing nodes are classified as donors versus borrowers, based on their TLB pressure, to achieve desirable sharing. It reduces translation latency through translation migration and replication triggered by hardware counters.

Based on the study of TLB sharing behavior, Bhattacharjee et al. proposed a TLB prefetching scheme [Bhattacharjee and Martonosi 2010] to reduce TLB misses. In this state-of-the-art distributed scheme, alluded to in Section 2, a *prefetch buffer* is used to avoid page table accesses for translations that miss in the private TLB. Using a technique called *leader-follower*, when a tile misses in the local TLB it becomes the “leader.” The fetched TLB entry is sent to the prefetch buffer of other “following” cores that utilize the leader’s pages frequently. A second technique, *distance-based cross-core*, matches the historical distance between TLB misses and predicts/prefetches TLB entries based on pattern matching. The system records the two distances between three successive TLB misses in a *distance table*. When two misses in any core match the first distance, the page matching the second distance is prefetched into the prefetch buffer.

All the above schemes provide a significant benefit for reducing TLB misses. However, these schemes have a significant overhead of adding tags/IDs to distinguish process IDs necessary for sharing entries between cores. TLB prefetching adds considerable complexity and significant overheads including complicated prefetching logic for pattern matching, prefetch buffers, distance buffers, and a distance table, which contains hundreds of entries. In addition, it requires $O(n^2)$ confidence counters to reduce bad prefetches (i.e., entries prefetched but never used) and performs a considerable amount of prefetch broadcasts when TLB miss rate is high, which can lead to poor performance. Synergistic TLBs also rely on significant amount of hardware resources including access counters, saturation counters, and complex policies for migration, replication, and victim allocation.

To achieve a similar goal with lower hardware complexity, Bhattacharjee et al. recently propose a physically shared last-level TLB [Bhattacharjee et al. 2011]. This system is demonstrated as effective for a four-core system, significantly outperforming private TLBs with the same overall number of translation entries. This approach represents the current best performing state-of-the-art shared L2 TLB architecture and as mentioned in Section 2, and we use it as the baseline for performance comparisons of our proposed PS-TLB.

Any shared TLB structure (either nonuniform access or physically shared) requires a mechanism, such as tags, to distinguish between translations for different processes/threads. This includes all of the previous work presented here. Thus, we presume that there is a baseline, 25% storage overhead introduced by all of these schemes.

In contrast, the PS-TLB distinguishes between shared and private TLB entries and uses a small tagged partial sharing buffer to retain the most heavily used shared translations on chip. All presented PS-TLB configurations use considerably less storage overhead than a shared structure that requires tags. Compared with schemes such as Synergistic TLBs and TLB prefetching, PS-TLB requires considerably less complexity in the system avoiding the need for hardware counters for migration and prediction. Yet, it retains fast (local) access for all translations, either private or shared, previously utilized within the tile. Additionally, by leveraging the translation classification support available in PS-TLB, the proposed scheme reduces the context switch and TLB shutdown overhead. We describe the PS-TLB in detail in the next section.

4. PARTIAL SHARING TLB

As indicated by the prior discussion, the translation performance is dependent on a variety of factors including off-chip translation rate, on-chip translation latency, and performance of other TLB operations. We present in this section the proposed PS-TLB, which leverages page classification information based on translation sharing to reduce off-chip translations while performing on-chip translations as fast as a traditional private TLB. Additionally, we discuss how optimized translation operations can be developed using the PS-TLB and their performance impact in different scenarios.

4.1. Sharing TLB Entries

Tagged TLBs, described in Section 3.2, can be used to create a shared last-level TLB either using a distributed shared last-level TLB similar to the mechanism used for sharing data in a distributed last-level cache [Kim et al. 2003] or with a physically shared structure. We compared various shared methods in Section 2 and compare the distributed method (tagged shared) with private TLBs (both 16-core, 256-entry L2 TLB/core system). As expected, the 8% miss rate for a private TLB drops to less than 3% for a shared TLB, as shown in Figure 4. However, the latency benefit is less clear. In this configuration, most applications perform better with private and some perform

Fig. 4. Last-level TLB miss rate for private vs. tagged shared TLB with 256 TLB entries per core.

Fig. 5. TLB latency for private vs. tagged shared TLB with 256 entries per core (normalized to private).

Fig. 6. TLB latency for private vs. tagged shared TLB with 64 entries per core (normalized to private).

better with shared. With a 64-entry per core L2 TLB (Figure 6), most applications perform better with shared than private.

Thus, we conclude that neither a simple, S-NUCA-style shared TLB nor a traditional private TLB is always suitable for scalable distributed systems. A scalable L2 TLB that includes properties of both private and shared translation caching is required. We describe the PS-TLB architecture that accomplishes this in the next section.

4.2. PS-TLB Architecture

The proposed PS-TLB, shown in Figure 7, assumes a tiled CMP architecture where each core is locally equipped with a 2-level private inclusive TLB. This allows the design to inherit the merit of low translation latency of the traditional private TLBs. To facilitate inter-core sharing, we augment each core with a tile of a *partial sharing buffer* (PSB), serving as the local contribution of a global pool for page translation sharing. Compared to a TLB entry, a PSB entry has an extra field, *PID*, to distinguish translations from different processes, similar to the tagged shared L2 TLB. Even by adding a tagged PSB, an effective PS-TLB will still use less resources than a fully tagged shared L2 TLB with the same number of entries because the private L2 TLB entries

Fig. 7. Partial sharing TLB organization and translation lookup.

Fig. 8. Structures for virtual address, TLB entry, PSB entry and page table entry (PTE).

do not require tags and the PSB is typically small. The tile where a particular PSB entry is placed is determined by selected bits from the virtual address. The PSB only accommodates translations that are shared by different cores. This prevents private translations from being placed remotely. It also eliminates the pollution of the PSB by private translations and increases the likelihood that shared translations are utilized by multiple cores as much as possible before being evicted from the PSB. In the next section we describe the method for translation/page classification.

4.2.1. Translation/Page Classification Support. Translation classification can be achieved by classifying virtual pages within a specific virtual address space (page table). To classify pages as either private or shared, each page table entry is extended with two additional fields, *FAC* (first accessing core) and *S* (shared), as shown in Figure 8.

On architectures such as MIPS and SPARC where TLB misses are processed by an operating system (OS) interrupt handler, a page classification scheme similar to the one proposed in R-NUCA for caches [Hardavellas et al. 2009] can be modified for use with TLBs. The OS initializes the *FAC* field with the first requester's core ID and clears the *S* flag, indicating that the page is private. On subsequent TLB misses, the OS handler checks the *S* flag to determine whether the accessed page has already been set as shared. If not, the OS compares the *FAC* in the PTE with the requester's ID and sets *S* in the case of a mismatch. This page classification information can then be used to guide on-chip translation caching. Our PS-TLB design only keeps the classification information in the page table, which avoids storage overhead in the TLBs, as shown in Figure 8.

On other CMPs (e.g., Intel x86), a TLB miss does not trap to the OS but rather invokes a hardware walker. In this scenario, the page classification can be easily completed by a hardware walker that is aware of the *FAC* and *S* fields in the page table. During a page table walk, the *FAC* and *S* fields of the target PTE are updated based on their previous contents and the current requesting core.

To determine the location of a shared translation, the *PSB home select (PHS)* bits are selected from the virtual page number (VPN) field of the corresponding virtual address, as depicted in Figure 8. The number of bits required in *PHS* is $\log_2 n$ for a CMP with n cores.

4.3. Basic Translation Operations on PS-TLB

Based on the proposed PS-TLB architecture with PSB and page classification support, we can efficiently migrate existing translation operations (e.g., TLB lookup, TLB placement, TLB shutdown, etc.) to the new TLB architecture with a small implementation overhead. To further improve translation performance in PS-TLB, a set of optimized translation operations are introduced, including exemptive flush, shutdown down-grade, and PSB prefill that leverage the additional information available in PS-TLB.

4.3.1. Parallel Translation Lookup. The address translation process starts when the MMU issues a virtual address in request of a physical address for an instruction or data access. The local L1 and L2 TLBs are first checked in sequence for the requested translation. Upon a hit on either L1 or L2 TLB, the PPN from the target translation is retrieved and returned to the MMU. Upon an L2 TLB miss if the page/translation is shared, then it could be at the PSB home tile on chip, otherwise it is off chip in the page table, requiring longer access time. Since the MMU would not know a priori whether the requested translation is private or shared, both the PSB home tile and the page table are searched in parallel, as illustrated in Figure 7. The MMU either receives the requested translation from the PSB home, or waits for the information from the page table access. Upon receiving the requested translation entry, the TLB, and as appropriate, the PSB home are filled with appropriate contents, as explained in the following section.

4.3.2. Translation-Classification-Aware Fill/Placement. From the perspective of the PS-TLB, the parallel lookup into both the PSB and the page table triggered by an L2 TLB miss could result in three different scenarios. These scenarios are illustrated in Figure 9. The PS-TLB performs a *translation-classification-aware fill/placement* depending on different scenarios.

In Figure 9(a), the PSB home determines that it has a valid translation by indexing its entries using the requested VPN and checking the *PID* field. It replies to the requester with the corresponding PPN and state. The requester, upon receiving the reply, also adds this entry to its local TLB. This local replication will serve fast translation upon subsequent requests on the same entry. In this case, the reply from the page table, which arrives later, is simply discarded.

In Figure 9(b), the translation request misses in the PSB home tile and the returned PTE is private (e.g., $S = 0$). The MMU is informed that the request is satisfied and the entry is added to the local private TLB.

In Figure 9(c), the translation request misses in the PSB home tile and the fetched PTE is either shared ($S = 1$) or private with the requester being different from the *FAC* core (i.e., $S = 0$ and $Req \neq FAC$)². The MMU is informed that the request is satisfied and the entry is added to the private TLB and the PSB home.

4.4. Optimized TLB Shutdown

TLB shutdown is necessary to keep translation consistent. Normally triggered by OS changes to PTEs, TLB shutdown is traditionally handled using IPIs, which requires all processor cores to be halted to handle the IPI. As mentioned in Section 3.1.3, OS changes to PTEs can be classified as unsafe changes and safe changes. Considering private versus shared PTEs, we distinguish four scenarios (i.e., unsafe changes to private PTEs, safe changes to private PTEs, unsafe changes to shared PTEs, and safe changes to shared PTEs) to efficiently integrate the TLB shutdown process into our PS-TLB.

²When $S = 0$ and $Req \neq FAC$ the PTE is changed by the OS from private to shared in the page table.

Fig. 9. Translation reply and fill on PST (a):hit a PSB entry;(b):PSB miss of a private PTE; (c):PSB miss of a shared PTE.

Fig. 10. TLB shutdown process on PS-TLB.

TLB shutdown can be processed using the same mechanism for the first two scenarios (unsafe/safe changes to private PTEs), as illustrated in Figure 10(a). In these two scenarios, the OS is aware that the modified PTE is private. Therefore, the shutdown process can be downgraded to a simple invalidation. This *shutdown downgrade* process reduces overhead by invoking the TLB invalidation instruction (e.g., `INVLPG` in Intel x86) to invalidate the entry in the owner's TLB without interrupting any of the other processing cores.

When an unsafe change occurs on a shared PTE, as shown in Figure 10(b), all the corresponding TLB entries as well as the PSB entry (if any) must be invalidated. This is largely the same as the traditional shutdown process.

The fourth scenario (Figure 10(c)) involves safe changes to shared PTEs. In this scenario, the corresponding TLB entries in all sharers of the PTE are invalidated, as normally occurs in a shutdown process. Meanwhile, the OS *prefills* the PSB home with the updated entry. This avoids future off-chip translation misses when at least one of the sharers reaccesses the page after the shutdown process. We expect such a prefilled entry in the PSB to be used in the near future by its sharers (i.e., prior to eviction from the PSB) as TLB entries invalidated due to safe changes typically contain active translations.

4.5. Optimized TLB Flush

For a typical private TLB organization, a TLB flush is performed upon a context switch to avoid translation conflicts. This is also true for the private TLB in our PS-TLB design. However, for the PSB, the *PID* uniquely identifies a process and like the tagged L2 TLBs [Villavieja et al. 2011] a PSB flush is not required during a context switch.

When a previously switched out process is reactivated, its surviving shared translations residing in the PSB can still be utilized, reducing the context switch overhead. We call this method of only flushing the TLB but not PSB *exemptive flush*.

4.6. Atomicity and Race Conditions

As shared resources, PTEs should be updated atomically, which is typically guaranteed by read-modify-write operations and a page table locking mechanism. The updated information will be observed by the cores upon TLB fills or PSB prefills after shoot-downs. In our PS-TLB, however, a core may receive a stale translation entry from the PSB after the PTE is locked in the page table and the corresponding entries are updated/invalidated in PSB/TLB. This results in a race condition in which the core utilizes a nonupdated translation entry for its local translation. To avoid this effect, we enforce that the PSB entry must be invalidated prior to the TLB shutdown. Alternatively, the TLB can discard the reply message from the PSB for an entry which has been invalidated in a shutdown process within a certain amount of time set using a experimentally determined threshold.

4.7. Discussion

4.7.1. Scalability. The PS-TLB is efficient for future CMPs with increasing numbers of cores (i.e., many-core CMPs). First, each tile retains a local copy of the translation, enabling low and constant translation latency upon an L2 TLB hit independent of the number of cores and NoC topologies. Additionally, the PSB is designed in a distributed manner that naturally scales with the number of cores/tiles in the CMP. Additionally, we require only a very small PSB compared with the private L2 TLB, making the overhead much less than a tagged shared solution.

4.7.2. Multi-program Workloads. The PS-TLB can be applied to multi-program workloads directly, without any significant performance reduction compared to a private approach. In the context of multi-program workloads, most pages are classified as private, resulting in less populated PSBs. However, the PS-TLB will outperform a physically shared solution that increases the access latency of all L2 TLB accesses, which in this scenario are predominantly private. However, the PSB can still serve any global pages that are shared among all workloads, which may also provide some benefit.

4.7.3. Thread Migration. Thread migration can be handled in PS-TLB in a similar way as a context switch. Typically, migration results in a TLB flush to the cores involved in the migration. This avoids any potential conflicts with the translations from different processes. In the PS-TLB, translations in the PSB remain intact during the thread migration since the translations are not associated with the tile requiring the translation. Of course, the *PID* will resolve any conflicts between processes that use the same virtual address. This allows the migrated thread to continue using its shared translations in the PSB after migration to the destination and reduces the migration overhead from private TLB flushing.

5. EVALUATION

In this section we evaluate the PS-TLB and compare it with several leading proposed TLB mechanisms that leverage sharing [Bhattacharjee and Martonosi 2010; Bhattacharjee et al. 2011]. In particular, we compare with the most recent, relevant effort in this area, which we assume provides the best performing, state-of-the-art TLB proposal for CMPs [Bhattacharjee et al. 2011]. We use the Wind River Simics [Magnusson et al. 2002] environment to simulate a 16-core CMP interconnected by a 4×4 mesh network. Table I summarizes the architectural parameters selected for our experiments. We assume the typical 4K page size and adopt a 150-cycle fixed

Table I. Architecture Configurations

System Parameters:	
Processor	16 SPARC cores, 2G Hz, 2 issue width
Operating System	64-bit Solaris 10
Network	4×4 Mesh, packet switching, X-Y routing, 3 cycles per hop
Main Memory	4GB, 150-cycle latency
Cache Parameters:	
L1 Cache Size	16KB/core private (MESI protocol)
L1 Cache Associativity	4-way associative
L1 Cache Latency	1-cycle hit latency
L2 Cache Size	512KB/core distributed shared
L2 Cache Associativity	32-way associative
L2 Cache Latency	3-cycle hit latency
Cache Block Size	64 Bytes
TLB Parameters:	
L1 TLB Size	32 entries/core
L1 TLB Associativity	4-way associative
L2 TLB Size	256 entries/core
L2 TLB Associativity	16-way associative
L2 TLB Latency	1-cycle hit latency
Off-chip Translation Latency	150 cycles
PSB Parameters:	
PSB Size	16 entries/core
PSB Associativity	4-way associative

last-level TLB miss penalty similar to related work. The replacement policy for all the caches, TLBs, and PSBs in the system uses LRU (Least-Recently Used). We evaluate the latency, miss rate, and performance of the PS-TLB and study the effects of using different TLB/PSB sizes. We also study the impact of our techniques on TLB shutdown and context switches.

For input workloads, we selected emerging parallel C/C++ and Java workloads from the DCAPO [Blackburn et al. 2006], SPLASH-2 [Arnold et al. 1992], and PARSEC-2 [Bienia et al. 2008] benchmark suites, which are representative of diverse application domains including scientific computing, text processing, database transaction handling, web hosting, financial modeling, etc. Table II shows the description and input working set sizes we used for these benchmarks.

5.1. Translation Miss Rate

We compare the miss rate of the physically shared L2 TLB with a PS-TLB in Figure 11. In the PS-TLB, last-level TLB misses may be served by the PSB if the requested entries are shared and cached in the PSB. Thus, the PSB hit rate indicates the percentage of page table translations (i.e., misses) that can be eliminated. We also consider the miss rate for the shared TLB to be misses serviced by the page table. As shown in Figure 11, the PS-TLB for most applications is competitive with physically shared, but does increase the miss rate, in some cases by a significant margin. These benchmarks that perform poorly often have a high number of shared pages (e.g., XALAN, LU, WATER, CANNEAL) that put more pressure on the smaller PSB capacity for shared pages than an entirely shared L2 TLB (see Figure 1). The significant advantage of PS-TLB will be in reduction of access latency, which we evaluate in the following section.

Table II. Benchmarks

Benchmark Suits	Benchmarks	Input Workloads
DACAPO	avroa batic eclipse fop h2 jython luindex lusearch pmd sunflow tomcat xalan	default input size 3 SVG files (default) small input size default XSL-FO file 400 SQL database transactions small input size default set of documents 64 keyword query batches default input size range of 1-256 (default) range of 1-4 (small) range of 1-100 (default)
SPLASH-2	ocean lu cholesky raytrace water	1026x1026 matrix 2048x2048 matrix <i>tk23.O</i> input <i>teapot</i> input 3375 molecules
PARSEC-2	blackscholes swaptions fluidanimate canneal x264	200000 options 256 swaptions in-35K.fluid input 100000.nets 640 × 360 4 frames

Fig. 11. Miss rate comparison of the PS-TLB with a PSB size of 16 entries compared with a centralized shared TLB.

5.2. Translation Latency

The key component of performance impact of the TLB design is the stall cycles per access. Figure 12 shows the translation latency reduction of PS-TLB normalized to the baseline of a physically shared L2 TLB. Due to the reduced latency of primarily local hits, nearly all of the tested benchmarks exhibit considerable latency reduction, reaching a reduction of more than 80% in some cases. The three benchmarks with degradations are LU, CHOLESKY, and CANNEAL, for which the centralized shared L2 TLB has a significant miss rate advantage. On average³, PS-TLB reduces translation latency by 45%.

³Average refers to arithmetic mean in the article.

Fig. 12. Translation latency (normalized to centralized shared).

Fig. 13. Speedup over a centralized shared TLB.

5.3. Overall Performance Impact

The impact of translation on overall system performance varies significantly for different applications and is dependent on a variety of factors including memory access intensity, workload/translation set sizes, data access patterns, instruction execution time, etc. It can be seen from Figure 13 that some benchmarks, such as FLUID, WATER, and SWAPTIONS, exhibit negligible performance gain, although they have achieved significant translation latency reduction in Figure 12. This is due to the translation latencies not contributing significant fractions to the entire execution time in these benchmarks. In a similar fashion, benchmarks that saw latency degradations such as LU and CHOLESKY did not introduce a performance degradation. In many of these cases, the TLB miss latency was often overlapped with other delays from memory system misses and/or poor load balancing in the application design. BLACKSCHOLES achieves little speedup because its private-dominant data/translation does not benefit much from any translation sharing mechanism.

In contrast, the latency improvement is critically important in some cases. For example the SUNFLOW benchmark has a 33% performance improvement due to a more than 80% TLB latency reduction. The average performance improvement over all benchmarks is 9%.

5.4. Sensitivity Analyses

To demonstrate the benefit of the proposed technique in different configurations, we study smaller TLBs with 64 entries/core and the effect of different PSB sizes. Figure 14 and Figure 15 show the translation miss elimination and latency reduction on configurations with 256 (denoted as PSTLB256 + PSB16) and 64 (denoted as PSTLB64 + PSB16) TLB entries/core. In each case we compare with the same number of L2 TLB entries per core with a zero size PSB⁴.

⁴A PS-TLB with a zero sized PSB is essentially equivalent to a private L2 TLB.

Fig. 14. Percentage of last-level TLB miss elimination on PS-TLB with 64 TLB entries/core.

Fig. 15. Translation latency of PS-TLB with 64 TLB entries/core (normalized to zero PSB).

Fig. 16. Percentage of last-level TLB miss elimination on PS-TLB with different PSB sizes.

Clearly, smaller TLB capacity (or larger working set size equivalently) increases off-chip translation rate and prolongs the translation latency, enlarging the optimization opportunity for PS-TLB. As can be observed from Figure 14, last-level TLB miss elimination increases from 48% on PSTLB256 + PSB16 to nearly 70% on PSTLB64 + PSB16. On some benchmarks (e.g., LU and WATER) the improvement is not remarkable since the working set sizes are relatively small and fit into the 64-entry/core configuration.

Figure 15 compares the latency improvement over a zero sized PSB. In PSTLB64 + PSB16 configuration, the latency reduction from the PSB is amplified due to more frequent TLB misses than the 256-entry-per-core configuration. On average, the translation latency reduction is 55% on a 64-entry-per-core configuration, as compared to 32% in a 256-entry-per-core system.

Figure 16 reports the percentage of last-level TLB misses eliminated by PS-TLB with variable sized PSBs normalized to a zero sized PSB. With just four entries in the PSB the miss rate is reduced by 33% on average and this reduction rises to 55% by a 20-entry PSB⁵. This leads to a corresponding scaling in the translation latency reduction, as depicted in Figure 17. The average latency reductions brought by 4-, 8-, 16-, and 20-entry PSBs are 23%, 27%, 32%, and 36%, respectively.

⁵We use associativity of 5 to keep the number of sets power of 2.

Fig. 17. Translation latency on PS-TLB with different PSB sizes (normalized to a zero PSB).

Fig. 18. Percentage of last-level TLB miss reduction compared to prefetching scheme.

Fig. 19. Translation latency compared to prefetching scheme.

5.5. Comparing with Prefetching Mechanism

In addition to comparing with shared L2 TLBs, we compare PS-TLB with the state-of-the-art scalable scheme that leverages prefetching, TLBPrefetch [Bhattacharjee and Martonosi 2010].

We implemented the TLBPrefetch technique using a 16-entry per-core prefetch buffer with a centralized distance table that has four entries per tile as well as *distance buffers*, which are local distance table caches consistent with Bhattacharjee and Martonosi [2010]. In comparison, PS-TLB uses a 20-entry PSB which is comparable to the combined resources of the prefetch and distance buffers. The PS-TLB scheme is actually considerably simpler than TLBPrefetch as it does not require the distance table resources, which is as large as adding an additional TLB, nor does it require the $O(n^2)$ *confidence counters* required to determine the tiles from which to allow prefetching.

Figure 18 compares the capability of last-level TLB miss reduction over TLBPrefetch. PS-TLB reduces miss rate for all benchmarks between 10% and 75% with an average of 32% reduction. The reduced off-chip translations result in a maximum of 48% and an average of 16% latency reduction over TLBPrefetch, as illustrated in Figure 19. All benchmarks were either improved or achieved the same latency with PS-TLB+PSB20.

Fig. 20. Unnecessary (wasted) PSB lookups after an L2 TLB miss.

5.6. Parallel Lookup Efficiency

One inefficiency incurred by PS-TLB is the parallel lookup upon a last-level TLB miss. A lookup into the PSB would be unnecessary if the requested entry turns out to be private, since that entry is never placed in a PSB. Unfortunately, the requester cannot know this until it receives the translation from a TLB or the page table. We call such unnecessary lookups *wasted PSB lookups* that could incur a power inefficiency. As revealed from Figure 20, most benchmarks have less than 20% wasted PSB lookups. Only a few benchmarks (BLACKSCHOLES and SWAPTIONS) have high percentages of wasted PSB lookups. This is likely due to the extremely low frequency of L2 TLB misses compared to hits (please refer to the TLB miss rate in Figure 4). We believe the benefit from eliminating page table lookups significantly outweighs the overhead incurred by wasted lookups and does not bring a significant negative impact, in terms of energy consumption, to the overall TLB operations.

5.7. Additional Benefits from PS-TLB

In this section we provide some insights for the additional benefits from private and shared classification and the proposed PS-TLB architecture.

5.7.1. Shutdown. The private and shared classification of translations used by PS-TLB provides an opportunity to improve the TLB shutdown process. If a private entry is identified for shutdown by the OS, it can be downgraded to a simple invalidation only stalling a single core. Shared entries with a common address that are demapped in multiple cores correspond to a shutdown of a shared PTE, which reverts to the standard shutdown process. In multicore systems, TLB shutdown is supported by low-level hardware primitives (e.g., INVLPG instruction in Intel x86 and TLB demap in SPARC/PowerPC) to invalidate TLB entries. This makes it easy to implement an invalidation instead of shutdown for private PTEs. In our experimental machine, we track the TLB demap operation to estimate the impact of shutdown downgrades. Figure 21 estimates the percentage of shutdowns that can be downgraded based on the amount and classification of the demapped translations. On average, more than 30% of the shutdowns can be downgraded to invalidations, which avoids considerable stalls.

5.7.2. Context Switching. The principal reason to introduce tags into the TLB was to avoid flushing due to context switching. PS-TLB provides a similar benefit with a much smaller tagged resource, the PSB. To study the impact of context switches, we ran an experiment to group benchmarks into pairs and run each pair with 32 threads (16 for each benchmark) on the simulated 16-core machine. We use thread binding to ensure that two threads, one from each benchmark in a pair, multiplex the same processing core. We identify context switches by detecting changes of core mode and

Fig. 21. Estimation of TLB shutdown downgrade savings.

Fig. 22. Latency and miss rate saving during context switches.

values in the context register. We compare the last-level TLB miss rate and latency of PS-TLB with different PSB sizes normalized to a fully tagged L2 TLB.

From Figure 22 we can see that PSB size has a significant impact on TLB misses due to context switching. For FLUID and SWAPTIONS, the last-level TLB miss rate drops from 7.6% to 1.4% on by introducing a 16-entry PSB, an 81.6% reduction. For other testing pairs, a 16-entry PSB improves miss rate reduction from 12% to 68% across all benchmarks, which are directly reflected in translation latency savings as shown on the left-hand side of Figure 22. Comparing with a tagged baseline, which uses 16 times as many tags, PSTLB256+PSB16 achieves close or better miss rate for all pairs except WATER and BLACKSCHOLES due to the high percentage of private pages in BLACKSCHOLES, which are not stored in the PSB. When we increase the PSB to 32 entries, PSTLB256+PSB32 outperforms the tagged TLB on both translation miss rate and latency on average. In terms of total additional resources, the tagged TLB requires $256 \times 16 \times 13 = 53248$ bits, assuming each tag has a size of 13 bits (e.g., ASID on UltraSparc) compared to only $32 \times 16 \times (64 + 13) = 39424$ bits for PSTLB256+PSB32.

6. CONCLUSIONS AND FUTURE WORK

In this article, we have presented PS-TLB, a scalable TLB solution for CMPs that leverages a classification of private and shared translations. PS-TLB outperforms the state-of-the-art TLB approach that leverages sharing [Bhattacharjee et al. 2011] by a 9% *application* speedup. PS-TLB also outperforms the leading distributed technique [Bhattacharjee and Martonosi 2010] reducing misses by 32% and improving translation latency by 16%. PS-TLB can also improve coherence by replacing 30% of IPIs with invalidations, and can outperform a fully tagged TLB in a study of context switching. Finally, PS-TLB requires fewer storage resources than existing state-of-the-art techniques due

to the reduction in the number of tags stored in the TLB. We have also studied the overhead of PSB parallel lookup and shown it to be small. We believe that the proposed design can be applied to build future fast, scalable, and efficient CMP systems.

Further directions of this work include a more extensive evaluation of TLB shutdown and context switching. The frequency and percentage of safe changes versus unsafe changes to PTEs can be also studied to understand the impact of the PSB prefill optimization potential not quantified in this work. We also plan to further evaluate the energy consumption of PS-TLB compared with other methods and the impact of thread migration on translation performance.

REFERENCES

- ARNOLD, J. M., BUELL, D. A., AND DAVIS, E. G. 1992. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York. 316–322.
- BARR, T. W., COX, A. L., AND RIXNER, S. 2010. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York. 48–59.
- BHATTACHARJEE, A., LUSTIG, D., AND MARTONOSI, M. 2011. Shared last-level tlbs for chip multiprocessors. In *17th International Symposium on High-Performance Computer Architecture (HPCA)*.
- BHATTACHARJEE, A. AND MARTONOSI, M. 2009. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 29–40.
- BHATTACHARJEE, A. AND MARTONOSI, M. 2010. Inter-Core cooperative tlb for chip multiprocessors. In *Proceedings of the 15th International Conference on (ASPLOS'10). Architectural Support for Programming Languages and Operating Systems*. ACM, New York. 359–370.
- BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The parsec benchmark suite: Characterization and architectural implications. Tech. rep. TR-811-08, Princeton University.
- BLACK, D. L., RASHID, R. F., GOLUB, D. B., AND HILL, C. R. 1989. Translation lookaside buffer consistency: A software approach. *SIGARCH Comput. Archit. News* 17, 113–122.
- BLACKBURN, S. M., GARNER, R., AND ET AL. 2006. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.* 41, 169–190.
- CHEN, J. B., BORG, A., AND JOUPPI, N. P. 1992. A simulation based study of tlb performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*. ACM, New York. 114–123.
- CLARK, D. W. AND EMER, J. S. 2000. Performance of the VAX-11/780 translation buffer: Simulation and measurement. In *Readings in Computer Architecture*. Morgan Kaufmann, San Francisco, CA, 418–433.
- HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. 2009. Reactive nuca: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, New York, 184–195.
- HUCK, J. AND HAYS, J. 1993. Architectural support for translation table management in large address space machines. *SIGARCH Comput. Archit. News* 21, 39–50.
- KIM, C., BURGER, D., AND KECKLER, S. W. 2003. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro* 23, 6, 99–107.
- MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Comput.* 35, 2, 50–58.
- ROMANESCU, B. F., LEBECK, A. R., SORIN, D. J., AND BRACY, A. 2010. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *16th International Symposium on High-Performance Computer Architecture (HPCA)*.
- ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. The impact of architectural trends on operating system performance. *SIGOPS Oper. Syst. Rev.* 29, 285–298.
- SRIKANTIAH, S. AND KANDEMIR, M. 2010. Synergistic tlbs for high performance address translation in chip multiprocessors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE Computer Society, 313–324.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECHREST, S., AND BROWN, R. 1994. Design tradeoffs for software-managed tlbs. *ACM Trans. Comput. Syst.* 12, 175–205.

- VENKATASUBRAMANIAN, G., FIGUEIREDO, R. J., AND ILLIKKAL, R. 2011. On the performance of tagged translation lookaside buffers: A simulation-driven analysis. In *19th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.
- VILLAVIEJA, C., KARAKOSTAS, V., VILANOVA, L., ETSION, Y., RAMIREZ, A., MENDELSON, A., NAVARRO, N., CRISTAL, A., AND UNSAL, O. S. 2011. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Received June 2012; revised September 2012; accepted September 2012