Shared Last-Level TLBs for Chip Multiprocessors

Abhishek Bhattacharjee
Dept. of Computer Science
Rutgers University
abhib@cs.rutgers.edu

Daniel Lustig
Dept. of Electrical Engineering
Princeton University
dlustig@princeton.edu

Margaret Martonosi
Dept. of Computer Science
Princeton University
mrm@princeton.edu

Abstract

Translation Lookaside Buffers (TLBs) are critical to processor performance. Much past research has addressed uniprocessor TLBs, lowering access times and miss rates. However, as chip multiprocessors (CMPs) become ubiquitous, TLB design must be re-evaluated.

This paper is the first to propose and evaluate shared last-level (SLL) TLBs as an alternative to the commercial norm of private, per-core L2 TLBs. SLL TLBs eliminate 7-79% of system-wide misses for parallel workloads. This is an average of 27% better than conventional private, per-core L2 TLBs, translating to notable runtime gains. SLL TLBs also provide benefits comparable to recently-proposed Inter-Core Cooperative (ICC) TLB prefetchers, but with considerably simpler hardware. Furthermore, unlike these prefetchers, SLL TLBs can aid sequential applications, eliminating 35-95% of the TLB misses for various multiprogrammed combinations of sequential applications. This corresponds to a 21% average increase in TLB miss eliminations compared to private, per-core L2 TLBs.

Because of their benefits for parallel and sequential applications, and their readily-implementable hardware, SLL TLBs hold great promise for CMPs.

1 Introduction

Processors supporting virtual memory employ instruction and data Translation Lookaside Buffers (TLBs) to store commonly-used virtual-to-physical page translations and to avoid high-latency accesses to operating system (OS) page tables. TLBs are crucial to system performance due to their long miss penalties [2, 7, 15, 20, 23]. While past work has addressed options for TLB placement and lookup [6, 22], contemporary systems place them in parallel with the L1 cache. TLB size and associativity have also been explored [6], as have superpaging [29] and prefetching [15, 24]. However, these proposals have all targeted uniprocessors. As chip multiprocessors (CMPs) become dominant, it is crucial to examine TLBs in this context.

This paper is the first to propose and analyze shared last-level (SLL) TLBs for CMPs. While processor vendors have addressed increasing application memory footprints and associated TLB stress by implementing two-level TLB hierarchies (eg. Intel i7 [11], AMD K8 and K10 architectures [1]), these designs use private, per-core L2 TLBs. No shared last-level TLB has been built commercially. While the commercial use of shared last-level *caches* may make SLL TLBs seem familiar, important design issues remain to be explored.

We show that a single last-level TLB shared among all CMP cores significantly outperforms private L2 TLBs for *parallel* applications. More surprisingly, it also often aids multiprogrammed workloads of *sequential* applications. For parallel applications, SLL TLBs exploit the fact that multiple threads experience TLB misses on identical address translation entries. This tendency of parallel programs to show *inter-core sharing* was recently observed [3] and subsequently exploited to develop Inter-Core Cooperative (ICC) TLB prefetchers [4]. However, unlike ICC prefetchers, which require complex hardware, SLL TLBs achieve comparable performance benefits with readily-implementable hardware. Moreover, unlike ICC prefetchers, SLL TLBs also benefit workloads combining sequential applications.

The design issues of SLL TLBs for multiprogrammed combinations of sequential applications are in particular need of study. For these workloads, applications do not access the same TLB entries, so one might expect SLL TLBs to show little improvement over private TLBs, or even degrade performance if a larger array requires longer access times. Contrary to these expectations, SLL TLBs usually do not hurt and often actually improve performance over private L2 TLBs. They accomplish this by allowing more flexible and adaptive use of a single pool of TLB resources, rather than more constrained per-core entries in the private L2 TLB approach. Overall, our contributions are as follows:

1. Foremost, our work is the first to explore SLL TLB design. While similar in spirit to the shared last-level caches on current CMPs, their unique design tradeoffs and requirements warrant focused study. Our results show that SLL TLBs provide considerable performance benefits not just for parallel programs where the performance advantages are expected, but also for multiprogrammed sequential workloads as well.

2. We analyze SLL TLB benefits for parallel programs. We show that they eliminate 7-79% of misses, a 27% improvement on average compared to private L2 TLBs with equivalent hardware requirements. These benefits are comparable to those achieved by ICC prefetchers, but with simpler hardware. Then, we investigate augmenting the baseline SLL TLB with rudimentary stride prefetching techniques, further increasing hit rates by an average of 5%. We also study the increasing effectiveness of SLL TLBs at higher core counts, showing that they eliminate an additional 6% of baseline TLB misses on average. Finally, we consider the performance implications of these eliminations, showing that they can cut down TLB miss handling times by up to 0.25 CPI.

3. We then analyze workloads consisting of sequential applications running one per core in a multi-

programmed fashion. We may expect such workloads to show insufficient commonality to benefit from SLL TLBs. Nonetheless, across workload mixes, we find that SLL TLBs eliminate 35-95% of the misses compared to no L2 TLBs. More notably, they outperform per-core private L2 TLBs, eliminating an additional 21% of the misses on average. These improvements arise because the SLL TLB is better able to allocate resources to the differing working set needs of simultaneously-running sequential applications. This typically leads to performance improvements, saving as much as 0.4 CPI.

The paper is structured as follows. Section 2 covers background material and Section 3 details SLL TLBs. Section 4 presents our evaluation methodology. Then, Section 5 shows results for parallel workloads while Section 6 does the same for multiprogrammed sequential workloads. Finally, Section 7 offers conclusions.

2 Background and Related Work

2.1 Uniprocessor TLB Characterizations

Contemporary architectures typically maintain private, per-core TLBs placed in parallel with first-level caches [1, 11]. Numerous past studies measured TLBs as comprising 5% to 10% of system runtime [7, 15, 20, 23] with extreme cases at 40% [10]. In response, a number of enhancement techniques were proposed. Early work addressed hardware characteristics such as TLB size and associativity [6]. Eventually, prefetching techniques [15, 24] and superpaging [29] were also studied with promising results.

While useful, this prior work specifically targets uniprocessors. As CMPs become ubiquitous, we must re-evaluate the role and design of TLBs. Although there is surprisingly little work in this area, processor vendors and the research community have proposed some solutions, as discussed below.

2.2 Private Multilevel TLB Hierarchies

Recognizing the increasingly critical role of TLBs to system performance, processor vendors have, over the years, extended the concept of multilevel hierarchies from caches to TLBs. Since the turn of the decade, AMD's K7, K8, and K10, Intel's i7, and the HAL SPARC64-III have embraced two-level TLB hierarchies [1, 11, 28]. Private L2 TLBs first appeared in uniprocessors, but they have become even more prevalent with the adoption of CMPs, with L2 TLBs approaching relatively large sizes with 512 and 1024 entries.

Though they are beneficial, all commercial L2 TLBs are implemented as private to individual cores. This paper shows that this strategy is deficient in two ways. First, per-core, private TLBs cannot leverage the inter-core TLB sharing behavior of parallel programs. Second, even for multiprogrammed combinations of sequential applications, per-core TLBs allocate a fixed set of resources to each individual core, regardless of the needs of applications running on them. Therefore, one core may execute an application with only a small TLB footprint, and another core may simultaneously experience TLB thrashing. This wastes resources since the unused TLB entries of the first core would have been better used if made available to the thrashing core.

As we show, SLL TLBs overcome both these deficiencies by exploiting the inter-core sharing of parallel programs and allocating resources gracefully among sequential applications in multiprogrammed workloads.

2.3 Inter-Core Cooperative Prefetching

The research community has also recently studied the impact of emerging parallel workloads on TLBs [3]. Characterizations of several parallel workloads show how significant commonality exists in TLB miss patterns across cores of a CMP, leading to two types of *predictable* TLB misses in the system. The first type is *Inter-Core Shared (ICS)*. This occurs when multiple cores TLB miss on the same translation. These misses occur often in parallel programs; for example, 94% of `Streamcluster's` misses and 80% of `Canneal's` misses are seen by at least 2 cores on a 4-core CMP, assuming 64-entry TLBs [3].

Recent work has proposed *Leader-Follower* prefetching to eliminate such ICS misses [4]. In this approach, on every TLB miss, the currently-missing core (the *leader*) refills its TLB with the appropriate entry and also pushes this translation to the other (the *follower*) CMP cores. The prefetches are pushed into per-core Prefetch Buffers (PBs) placed in parallel with the TLBs. Our SLL TLBs capture this same class of ICS TLB misses, but do so with much more streamlined hardware.

A second type of TLB miss is defined as *Inter-Core Predictable Stride (ICPS)*. These occur when multiple cores TLB miss on virtual pages with a consistent stride between them. For example, `Blackscholes` actively employs inter-core strides of ± 4 pages, making 86% of its misses predictable [3]. *Distance-based Cross-Core* prefetching targets ICPS misses [4]. This scheme stores repetitive inter-core strides in virtual pages in a central, shared Distance Table (DT). On TLB misses, the DT predicts subsequent required translations which can be prefetched. In our evaluations, we refer to ICC prefetching as combining both Leader-Follower and Distance-based Cross-Core prefetching.

While SLL TLBs do not directly attack ICPS misses, we show that SLL TLBs augmented with simple stride prefetching can meet or exceed the performance of the more complicated Distance-based Cross-Core prefetchers, and can do so with smaller hardware overheads (no tables) and simpler implementations. Furthermore, this paper shows that SLL TLBs benefit not just parallel workloads, but also, in many cases, aid multiprogrammed combinations of sequential applications. ICC prefetchers have not been evaluated on such workloads.

2.4 Our Approach: Shared Last-Level TLBs

Having detailed the limitations of existing strategies, we now discuss the novelty of our approach. We begin by comparing design issues of SLL TLBs and shared last-level caches. While at first glance there may seem to be parallels between shared last-level caches and a shared TLB, many key differences remain. First and foremost, SLL TLBs see more inter-core sharing since the granularity of storage is in the form of pages rather

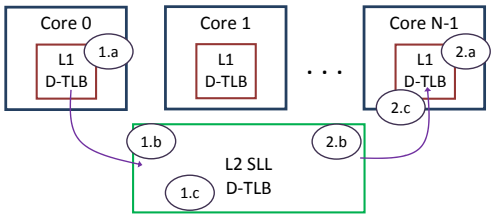

Figure 1: The basic structure of a shared last-level TLB involves a CMP with private, per-core L1 TLBs and a larger, shared L2 TLB. Cases 1 and 2 detail instances of SLL TLB misses and hits respectively.

than cache lines, increasing the chances of sharing and resulting in fundamentally different sharing patterns. Second, since TLBs are far smaller than caches, eviction and sharing play different and important roles in performance. Understanding their behavior, particularly in the context of multiple threads contending and sharing resources, requires a study in its own right. Moreover, the penalty of a TLB miss is typically much more severe than a cache miss since an expensive page table walk is involved. Therefore, shared TLBs warrant a detailed examination distinct from caches.

3 Shared Last-Level TLBs

We first describe SLL TLBs and detail their operation and implementation. We then discuss augmenting SLL TLBs with prefetching mechanisms as well.

3.1 Concept

Figure 1 presents a CMP with private, per-core L1 TLBs backed by an SLL L2 TLB. While this example uses just one level of private TLBs, more levels may be accommodated (for example, each core could maintain two levels of per-core private TLB followed by an L3 SLL TLB). As with last-level caches, the SLL TLB is accessed when there is a miss in any L1 TLB. The SLL TLB strives for inclusion with the L1 TLB, so that entries that are accessed by one core are available to others. Figure 1 shows the SLL TLB residing in a central location, accessible by all the cores. While this centralized approach is a possible implementation, we discuss this and other implementation issues in Section 3.3.

SLL TLBs enjoy two orthogonal benefits. First, they exploit inter-core sharing in parallel programs. Specifically, a core's TLB miss brings an entry into the SLL TLB so that subsequent L2 misses on the same entry from other cores are eliminated. Second, even for unshared misses, SLL TLBs are more flexible regarding where entries can be placed. TLB hits arising from this flexibility aid both parallel and sequential workloads.

3.2 Algorithm

Figure 1 details SLL misses and hits in two example cases respectively. While these cases are numbered, there is no implied ordering between them. We detail the cases below:

Case 1: Figure 1 follows an L1 TLB and SLL TLB miss. First, there is an L1 TLB miss (step 1a). In response, a message is sent to the SLL TLB. After the access latency, we suffer an SLL miss (step 1b). The page table is then walked and the appropriate translation is inserted into both the SLL and L1 TLB. By entering the entry into the SLL TLB (step 1c), future misses on this entry are avoided by both the initiating core as well as the other cores.

Case 2: We now illustrate the steps involved in an SLL TLB hit. First, the L1 TLB sees a miss (step 2a), and a message is sent to the SLL TLB. Now suppose there is an SLL TLB hit (step 2b). As previously detailed, there are two possible causes of this hit. First, the currently missing core may have previously brought this entry into the SLL TLB. Alternately, the entry may be inter-core shared and another core may previously have fetched it into the SLL TLB. Regardless of the reason, an SLL TLB hit avoids the page table walk. Instead, the same entry is now inserted into the L1 TLB (step 2c) in the hope that future accesses to this entry will be L1 hits.

3.3 Implementation Options

Having detailed the basic operation of SLL TLBs, we now address some key implementation attributes:

TLB Entries: SLL TLB entries store information identical to the L1 TLB. Each entry stores a valid bit, the translation entry, and replacement policy bits. We also store the full context or process ID with each entry. Less hardware could be used with fewer bits but our SLL TLB is small, making such optimizations unnecessary.

Replacement Policies: To leverage inter-core sharing in parallel programs, the L1 and SLL TLBs need to be inclusive. However, as with multilevel caches, guaranteeing strict inclusion requires tight coordination between the L1 and the L2 SLL TLB controllers and replacement logic [9]. Instead, we use the approach taken by x86 caches [9] and implement a multilevel TLB hierarchy that is *mostly-inclusive*. Here, while entries are placed into both the L1 and SLL TLB on a miss, each TLB is allowed to make independent replacement decisions, requiring far simpler hardware. Furthermore, processor vendors have noted that while this approach does not guarantee strict inclusion, it achieves almost perfect inclusion in practice. For example, in our applications, we find that above 97% of all L1 TLB entries are present in the SLL TLB.

Consistency: Our SLL TLBs are designed to be *shootdown-aware*. Whenever a translation entry needs to be invalidated, both the SLL and the L1 TLBs must be checked for the presence of this entry. Had our SLL TLB been strictly inclusive of the L1 TLBs, this would be unnecessary in the case of an SLL miss. However, since our two TLB levels are mostly-inclusive, it is possible for an entry to be absent from the SLL TLB but be present in the L1 TLBs. Therefore, a shootdown requires checks in all the system TLBs. Nonetheless, shootdowns are rare and the simpler hardware afforded by the mostly-inclusive policy make it appropriate for our proposed approach.

Placement: Here, we assume a unified, centralized SLL TLB equidistant from all cores. This is feasible for the current size of SLL TLBs we study (512 entries, as detailed in Section 4), which enjoy short hit times (2 cycles for 45nm technology from CACTI experiments [19]). If future SLL TLBs are considerably larger and

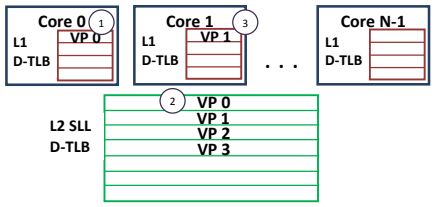

Figure 2: Enhancing the SLL TLB with simple stride prefetching. When an L1 and SLL TLB miss occur, both the requested translation and other translations a stride of 1, 2, and 3 pages away are inserted into the SLL TLB.

require longer hit times, they could be distributed similarly to NUCA caches [16].

As with caches, a communication medium exists between cores and the SLL TLB (eg. on-chip network or bus). Therefore, SLL roundtrip latency is comprised of the network traversal and SLL TLB access time. Given short access latencies of 2 cycles, network traversal time dominates. We assume network traversal times of 20 cycles, similar to [4]. While this does mean that 22 cycles total are spent on SLL TLB hits, we will show that this still vastly improves performance by eliminating page table walks that could take hundreds of cycles [12, 13]. Techniques that reduce this communication latency will only amplify the SLL TLB benefits.

Finally, since the SLL TLB is centrally shared among all the cores, it will require longer access times than the private L2 TLBs. Based on CACTI simulations at 45nm, scanning the private L2 TLB takes the same amount of time as the SLL TLB (2 cycles); however, since private L2 TLBs do not need to be centralized among cores, they have a communication time which is shorter by 6 cycles.

Access Policies: While L1 TLBs handle only one request at a time and are blocking, SLL TLBs could potentially be designed to service multiple requests together. This, however, complicates both the hardware and how the OS handles page table walks; our design therefore assumes blocking SLL TLBs. Nevertheless, non-blocking SLL TLBs would likely provide even more performance benefits.

3.4 SLL TLBs with Simple Stride Prefetching

In some of our experiments, we also consider augmenting SLL TLBs with simple prefetching extensions. A number of past studies have shown that due to large-scale spatial locality in memory access patterns, TLBs often exhibit predictable strides in accessed virtual pages. These strides occur in memory access streams from a single core [15] as well as between multiple multiple cores [3, 4]. While sophisticated prefetchers have been proposed to exploit this, this work explores the degree to which simple stride-based prefetching added to the baseline SLL TLB can provide benefits. Specifically, on a TLB miss, we insert the requested translation into the SLL TLB and also prefetch entries for virtual pages consecutive to the current one. Figure 2 describes SLL TLBs with prefetching integrated:

Step 1: First, we assume that a TLB miss has occurred in both the L1 and SLL L2 TLBs. Having walked the page table to find the translation corresponding to the missed virtual page (page 0 in this example), the appropriate entry is placed into the L1 TLB.

Step 2: Having refilled the L1 TLB entry in the first step, we now fill the same entry into the SLL TLB. Next, prefetching is activated. To capture potential intra-core and inter-core strides, we now prefetch entries for virtual pages consecutive to the one just missed upon. The number of and particular strides of the prefetched entries are design choices that subsequent sections will discuss. In this example, virtual page 0 has been missed upon, so we choose to also prefetch translations for pages 1, 2, and 3.

Step 3: Suppose that core 1 requests the translation for virtual page 1 because it has an inter-core stride of 1 page from core 0. Assuming that we miss in the L1 TLB, we scan for the entry in the SLL TLB structure. Fortunately, because of stride prefetching, we find that the entry already exists in the SLL TLB. An expensive page table walk is eliminated and all that remains is for the entry to be refilled into the L1 TLB as well.

It is critical to ensure that these prefetches do not add overheads by requiring extra page table walks. To avoid this, we propose a simple piggyback handling approach. When a TLB miss and its corresponding page table walk occur, we eventually locate the desired translation. Now, this translation either already resides in the cache or is brought into the cache from main memory. Because cache line sizes are larger than translation entries, a single line will maintain multiple translation entries. For our 64-byte cache lines (see Section 4), entries for virtual pages 1, 2, and 3 pages away will also reside in the same line. Therefore, we prefetch these entries into the SLL TLB, with no additional page walk requirements. Moreover, we permit only *non-faulting* prefetches.

4 Methodology

To quantify the benefits of SLL TLBs, we focus on two distinct sets of evaluations. First, we show how parallel programs benefit from SLL TLBs. We then also evaluate workloads in which a different sequential application runs on each core. This section describes each methodology in turn.

While SLL TLBs benefit both I-TLBs and D-TLBs, this study focuses on D-TLBs because of their far greater impact on system performance [3, 24]. Our approaches will, however, reduce I-TLB misses as well.

4.1 Parallel Applications

4.1.1 Simulation Infrastructure

We study SLL TLBs with parallel programs using the Multifacet GEMS simulator [18] from Table 1. Our simulator uses Virtutech Simics [30] as its functional model to simulate 4-core and 16-core CMPs based on Sun's UltraSPARC III Cu with SunFire's MMU architecture [27]. As shown, this uses two L1 TLBs that are looked up concurrently. The OS uses a 16-entry, fully-associative structure primarily to lock pages. A second 64-entry TLB is used for unlocked translations. Our L1 TLB sizes match the ICC prefetcher studies from [4]. Furthermore, these sizes are similar to the L1 TLBs of contemporary processors such as Intel's i7 (64-entry) and AMD's K10 (48-entry).

System	4-core SPARC
Issue/Commit Width	4-instruction
Reorder Buffer	64-entry
Instruction Window	32-entry
L1 cache	Private, 32 KB (4-way)
L2 cache	Shared, 16 MB (4-way)
L2 roundtrip	40 cc (uncontested)
Private L1 TLBs	16-entry fully-assoc TLB (locked/unlocked pages), 64-entry, 2-way TLB (unlocked pages)
OS	Sun Solaris 10

Table 1: Simulation parameters used to evaluate SLL TLB benefits for parallel workloads.

Strategy	Description
Per-Core Private L2 TLBs (Conventional case)	128-entry, 4-way 16 cc roundtrip interconnect: 14 cc, access: 2 cc
Shared Last-Level L2 TLB (Our strategy)	4-core case: 512-entry, 4-way 16-core case: 2048 entries, 4-way 22 cc roundtrip (interconnect: 20 cc, access: 2 cc)
ICC Prefetching (For comparison)	16-entry PB per core 512-entry DT 28 cc DT roundtrip (interconnect: 20 cc, access: 8 cc)

Table 2: TLB enhancements evaluated in this work. SLL TLB and private, per-core L2 TLB sizes match the ICC prefetchers designed in [4].

To assess the benefits of SLL TLBs, we compare them against both per-core, private L2 TLBs and ICC prefetchers (including both Leader-Follower and Distance-based Cross-Core prefetching) with the same total hardware. We first compare our SLL TLBs against the ICC prefetchers from [4] which assume a 4-core CMP with the configuration detailed in Table 2. As shown, based on these configurations, an equally-sized SLL TLB requires 512 entries. This in turn means that for a 4-core CMP, we compare SLL TLBs to private L2 TLBs of 128 entries. Finally, TLB access times are assigned from CACTI [19] assuming a 45nm node. These penalties include time to traverse the on-chip network as well as time to scan the TLB array. We find that the TLB scan times for both approaches remain the same (2 cycles); however since the private L2 TLBs are placed closer to the cores than the L2 SLL TLB, they have quicker network traversal (by 6 cycles).

After comparing the benefits of SLL TLBs with ICC prefetchers and private, per-core L2 TLBs on a 4-core CMP, we study the impact of core counts on SLL TLBs. For these experiments, we model a 16-core CMP as shown in Table 2. In order to fairly compare a 16-core CMP with private, per-core TLBs of 128 entries, we model a 2048 SLL TLB for these studies.

Benchmark	Model	Working Set
`Streamcluster`	Data-parallel	16MB
`Canneal`	Unstructured	256MB
`Facesim`	Data-parallel	256MB
`Fluidanimate`	Data-parallel	64MB
`x264`	Pipeline-parallel	16MB
`Ferret`	Pipeline-parallel	64MB
`VIPS`	Data-parallel	16MB
`Swaptions`	Data-parallel	512KB
`Blackscholes`	Data-parallel	2MB

Table 3: Summary of PARSEC benchmarks used to evaluate SLL TLBs. Note the diversity in parallelization models and working set sizes.

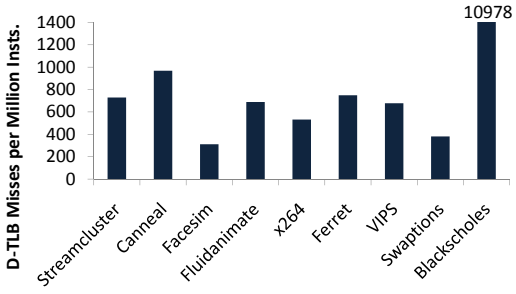

Figure 3: D-TLB misses per million instructions (MMI) for PARSEC workloads. `Canneal`, `Ferret`, and `Streamcluster` see high D-TLB misses, consistent with their working sets. While `Blackscholes` has a smaller working set, its access pattern results in the highest MMI.

Due to slow full-system timing simulation speeds, we present results for 1 billion instructions. Our instruction windows are chosen such that under 5% of the total D-TLB misses are cold misses. Historically, TLB studies [2, 3] focus on miss elimination rates rather than performance as it is infeasible to run applications with long enough durations on timing simulators to provide practical runtime performance numbers. We go beyond previous work and investigate performance (in addition to miss eliminations) by carefully considering TLB miss handling strategies and analytically modeling our benefits appropriately.

4.1.2 Parallel Benchmarks and Input Sets

We use the PARSEC benchmarks, a suite of next-generation shared-memory programs for CMPs [5]. Table 3 lists the workloads used in this study. Of the 13 available workloads, we are able to compile the 9 listed for our simulator.[1] The workloads use diverse parallelization strategies (unstructured, data-parallel, and pipeline-parallel) and are run with a thread pinned to each CMP core. Since TLB misses occur less frequently than cache misses, we use the largest available input data set feasible for simulation, the *Simlarge* set.

[1]These are also the PARSEC workloads that are studied in [3, 4] and hence, serve as a point of reference for our results.

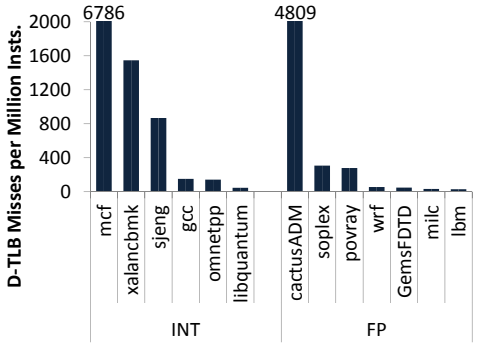

Figure 4: D-TLB misses per million instructions (MMI) for the SPEC CPU2006 workloads used in this paper. The workloads exhibit diverse miss rates, with `mcf` and `cactusADM` showing particularly high MMIs.

Figure 3 presents the workload D-TLB misses per million instructions (MMIs). As expected, benchmarks such as `Canneal`, and `Ferret`, which have large working sets, see high MMIs. Interestingly, `Blackscholes` sees the highest MMI due to its access pattern, despite a relatively modest working set size. These MMI numbers present a useful foundation to better understand our subsequent results.

4.2 Multiprogrammed Workloads of Sequential Applications

4.2.1 Simulation Infrastructure

We also provide results for multiprogrammed sequential workloads. As for parallel workloads, our sequential applications use the full-system 4-core CMP simulator of Table 1. Using a similar approach to previous studies [8, 14, 25], we advance simulation by four billion instructions and evaluate performance over a window of ten billion instructions. Unlike the parallel workload experiments, we evaluate the multiprogrammed workloads using functional simulation only. This is in part because these multiprogrammed sequential workloads are not as heavily influenced as the parallel ones by inter-thread timing interactions. In addition, our functional approach allows us to capture larger swaths of execution, which is important because of the large *Ref* datasets we use to fully exercise the TLB. Since TLB effects occur over such long timescales, the key is for the window to be sufficiently large, to observe and contrast the behavior of the various workloads. Our functional simulation also includes OS effects, which are naturally quite important to our study. Finally, as with parallel workloads, while we cannot present raw full-program runtime performance numbers, we do provide performance intuition through careful analysis of TLB miss handling overheads and analytical models.

We use sequential applications from the SPEC CPU2006 [26] suite to form our multiprogrammed workloads. We choose to evaluate the workloads designated by [21] as capturing the overall performance range of the SPEC CPU2006 suite. Figure 4 provides an initial characterization of these benchmarks which include 6 integer and 7 floating-point applications. As shown, the applications see varying D-TLB MMIs for the 64-entry TLBs simulated in this system. In particular, we find that `mcf` and `cactusADM` most severely stress our TLBs with MMIs of 6786 and 4809 respectively.

ID	Stress	SPEC Benchmarks
Het-1	Inter.	`mcf, xalancbmk, sjeng, libquantum`
Het-2	Low	`xalancbmk, sjeng, libquantum, gcc`
Het-3	Inter.	`cactusADM, milc, soplex, lbm`
Het-4	Low	`soplex, lbm, wrf, povray`
Het-5	High	`cactusADM, mcf, omnetpp, GemsFDTD`
Hom-1	High	4 copies of `mcf`
Hom-2	Low	4 copies of `xalancbmk`

Table 4: The multiprogrammed workloads used in this paper. Five of the workloads are constructed to be heterogeneous (Het-1 to Het-5) while two are homogeneous (Hom-1 and Hom-2). The workloads are designed to show varying degrees of TLB stress.

While a fully-comprehensive analysis of multiprogrammed workloads comprised of four applications would involve simulation of all $\binom{29}{4}$ combinations of benchmarks, this is practically infeasible. In conjunction with Figure 4, we therefore draw from the methods and data in [21] to form seven workloads of four SPEC CPU2006 applications each. Table 4 lists these combinations in detail.

As shown in Table 4, these combinations stress the TLBs to varying degrees. We separate them into five heterogeneous workloads (Het-1 to Het-5) and two homogeneous workloads (Hom-1 and Hom-2). The heterogeneous workloads provide insight into how well SLL TLBs adapt to programs with different memory requirements. In contrast, the homogeneous ones model scenarios where no single application overwhelms the others.

We construct the workloads as follows. First, we design two heterogeneous workloads with intermediate levels of TLB stress by combining one high-stress application with three lower-stressed ones. Here, `mcf` and `cactusADM` serve as high-stress benchmarks and therefore are used to create intermediate-stress workloads Het-1 and Het-3 along with three other lower-stress applications. Second, for comparison, we create a pair of low-stress workloads, Het-2 and Het-4. Finally, our last heterogeneous workload is designed to be very high-stress. Therefore, in this case we combine both `mcf` and `cactusADM` along with two other workloads in Het-5.

For the homogeneous workloads, we once again focus on a high-stress and low-stress case. The high-stress workload is constructed using four copies of `mcf` while the low-stress workload uses four copies of `xalancbmk`.

5 SLL TLBs: Parallel Workload Results

We now study SLL TLBs for parallel workloads. First, Section 5.1 compares SLL TLBs against commercial per-core, private L2 TLBs. Second, Section 5.2 compares SLL TLBs with ICC prefetching. Section

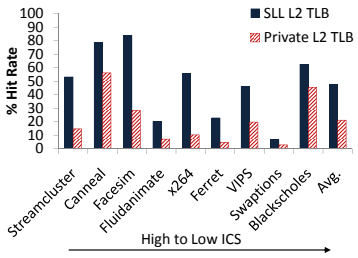

Figure 5: SLL TLB versus private, per-core L2 TLB hit rates. While private L2 TLBs do provide benefits, they are consistently outperformed by SLL TLBs (by 27% on average).

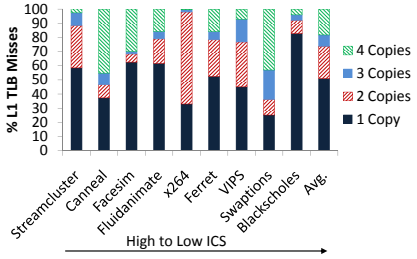

Figure 6: Copy counts for private L2 TLBs. For every evicted L1 line, we record how many L2 TLBs hold this entry. Heavy replication of entries exists, which SLL TLBs mitigate.

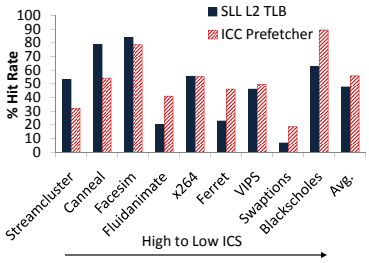

Figure 7: SLL TLB hit rate versus ICC prefetcher hit rate. Benchmarks with high inter-core sharing like `Canneal`, `Facesim`, and `Streamcluster` benefit the most from SLL TLBs.

5.3 evaluates the benefits of enhancing the baseline SLL TLB operation with stride prefetching. Section 5.4 then studies the benefits of the SLL TLB with increasing core counts. Finally, Section 5.5 conducts a detailed performance analysis of our approach.

5.1 SLL TLBs versus Private L2 TLBs

Figure 5 shows the hit rates of a single 512-entry SLL TLB and per-core, private 128-entry L2 TLBs in a 4-core CMP. The benchmarks are ordered from highest to lowest inter-core sharing [3]. The overriding observation is that SLL TLBs eliminate significantly more misses than private L2 TLBs using the same total hardware for every single application. On average the difference in hit rates is 27%.

Second, high-ICS applications like `Canneal`, `Facesim`, and `Streamcluster` see especially high hit rate increases as compared to the private L2 case (by 23%, 57%, and 38% respectively). This occurs because SLL TLBs deliberately target inter-core shared misses.

Figure 5 also shows that `x264` sees the biggest improvement using SLL TLBs versus private L2 TLBs. As we will show, this is because many entries in each private L2 are replicated for this application; in contrast, the SLL TLB eliminates this redundancy, allowing for more TLB entries to be cached for the same hardware.

Figure 6 explores this issue of replication in greater detail. To analyze this, on every L1 TLB miss, we scan all the private L2 TLBs to look for the number of existing copies of the missing translation entry. Then, as a percentage of the total L1 misses that exist in at least one L2 TLB, we show separately the number of misses that have a single or multiple copies. Higher copy-counts are indicative of applications which would gain even more from SLL TLBs that remove redundancy and use the extra hardware to cache more unique translations.

Figure 6 shows that heavy replication exists across the benchmarks. For example, `Canneal` sees that 45% of its L1 evictions are replicated across all 4 cores. As mentioned, `x264` suffers from an extremely high copy-count, which SLL TLBs eliminate. In fact, even lower-ICS benchmarks like `Ferret` and `Swaptions` see high replication rates. Therefore, it is clear that maintaining separate and private L2 TLBs results in wasted resources as compared to a unified SLL TLB.

5.2 SLL TLBs versus ICC Prefetching

We now consider benefits versus previously-proposed ICC prefetching (which includes both Leader-Follower and Distance-based Cross-Core prefetching). Figure 7 shows the hit rate of a 512-entry SLL TLB compared to the ICC prefetcher. On average, SLL TLBs enjoy a hit rate of 47%. These hit rates rival those of ICC prefetchers, but SLL TLBs achieve them with simpler hardware.

On average, SLL TLBs see merely a 4% drop in hit rate compared to ICC prefetchers. Moreover, Figure 7 shows that in many high-ICS workloads like `Canneal`, `Facesim`, and `Streamcluster`, SLL TLBs actually outperform ICC prefechers. In fact, SLL TLBs eliminate an additional 24%, 6%, and 21% TLB misses for these workloads. However, applications like `Blackscholes` which are highly ICPS see lower benefits than from ICC prefetching. Nevertheless, SLL TLBs still manage to eliminate a high 62% of the TLB misses for `Blackscholes`. Overall, SLL TLBs eliminate a highly successful 7% to 79% of baseline TLB misses across all applications, while requiring simpler hardware than ICC prefetchers.

Apart from the benefits of SLL TLBs, it is also useful to understand their sharing patterns. Figure 8 plots, for every L1 TLB miss and SLL TLB hit, the number of distinct cores that eventually use this particular SLL entry. We refer to these distinct cores as sharers. On a 4-core CMP, there are up to 4 sharers per entry.

High-ICS benchmarks enjoy high SLL TLB entry sharing. For example, 81% of `Streamcluster's` hits are to entries shared among all 4 cores. Less intuitive but more interesting is the fact that even benchmarks with lower inter-core sharing such as `x264`, `VIPS`, and `Swaptions` see high sharing counts for their SLL hit entries. This is because the SLL TLB effectively prioritizes high-ICS entries in its replacement algorithm; hence, these entries remain cached longer. On average, roughly 70% of all hits are to entries shared among at least two cores.

We also consider sharing patterns of evicted translations. Figure 9 illustrates the number of sharers for every evicted SLL TLB entry. The vast majority (on average, 75%) of the evictions are unshared. This reaffirms our previous hypothesis that the SLL structure helps priori-

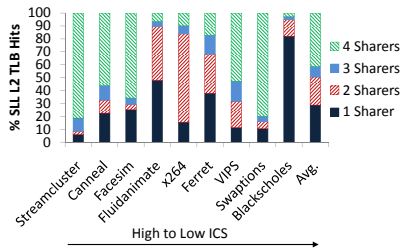

Figure 8: Sharing characteristics of each SLL L2 TLB hit entry. Note that high-ICS applications like `Canneal`, `Facesim`, and `Streamcluster` see high SLL inter-core sharing.

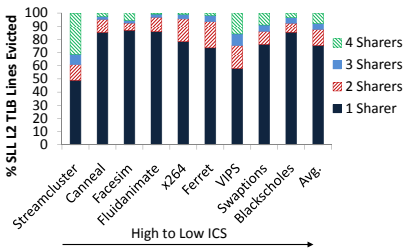

Figure 9: Sharing patterns of SLL TLB entries evicted. As shown, most evicted entries are unshared; inter-core sharing increases priority in replacement algorithm, decreasing eviction likelihood.

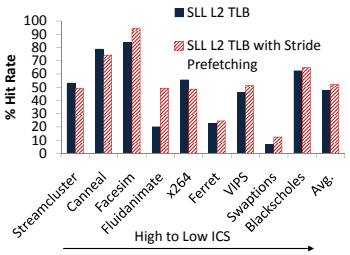

Figure 10: Including simple stride prefetching improves SLL TLB hit rates by an average of 5% across the evaluated workloads. This is because it captures repetitive inter-core distance pairs.

tize shared TLB entries in parallel applications. Namely, entries accessed by multiple cores are frequently promoted to the MRU position, while those accessed by a single core are more likely to become LRU and therefore prime candidates for eviction. Since our parallel workloads have many ICS misses, SLL TLBs cache translations that will be used frequently by multiple cores.

5.3 SLL TLBs with Simple Stride Prefetching

Having studied the hit rates of the baseline SLL TLB, we now consider low-complexity enhancements. Specifically, we now add simple stride prefetching for translations residing on the same cache line as the currently missing entry. This offers the benefits of prefetching without the complexity of ICC techniques. As covered in Section 3.4, prefetched candidates are 1, 2, and 3 pages away from the currently missing page.

Figure 10 compares the proposed SLL TLB alone, versus an SLL TLB that also includes stride prefetching. First, we see that the benefits of this approach vary across applications. `Blackscholes`, which has repetitive 4-page strides [4], sees little benefit since the only strides being exploited here are 1, 2, and 3 pages. However, `Fluidanimate` and `Swaptions` enjoy greatly improved hit rates since they do require strides of 1 and 2 pages [4]. Similarly, even `Facesim` sees an additional 10% hit rate since it exploits 2 and 3 page strides.

Figure 10 also shows that applications lacking prominent strides (eg. `Canneal` and `Streamcluster`) can actually see slightly lower hit rates. This is because the useless prefetches can displace useful SLL TLB entries.

Overall, while stride prefetching provides benefits for most applications, one may also consider the prospect of combining ICC prefetchers with SLL TLBs. While this is certainly possible, the main motivation of SLL TLBs is to achieve similar performance to ICC prefetchers but with much lower hardware complexity. Thus, SLL TLBs with simpler stride prefetching are an elegant and effective alternative.

5.4 SLL TLBs at Higher Core Counts

Our results indicate that SLL TLBs are simple yet effective at 4 cores. It is also important, however, to quantify their benefits at higher core counts. To this end, we now compare the benefits of SLL TLBs against private, per-core L2 TLBs at 16 cores.

Figure 11 plots the increase in hit rate that SLL TLBs provide over 128-entry private, per-core L2 TLBs (higher bars are better) for 4-cores and 16-cores. Since

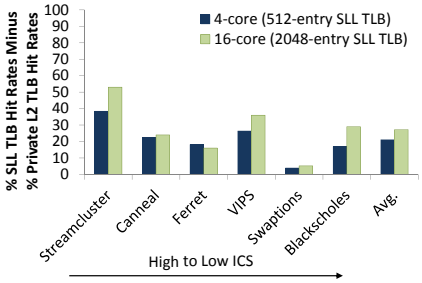

Figure 11: Increase in hit rate that SLL TLBs provide versus private, per-core L2 TLBs for 4-core and 16-core CMPs. Since private L2 TLBs are 128-entry, the SLL TLB is 512-entry and 2048-entry for 4-core and 16-core CMPs respectively. Note the *increased* hit rates at higher core counts.

each private L2 TLB is 128 entries, equivalently-sized SLL TLBs are 512-entry for the 4-core case and 2048-entry for the 16-core case.

Figure 11 demonstrates that not only do SLL TLBs consistently outperform private L2 TLBs (each bar is greater than zero), the benefits actually tend to *increase* at higher core counts. For example, `Streamcluster` and `VIPS` for 16-core CMPs enjoy an additional 10% increase in hit rate over the 4-core case. In fact, the benefits increase by 6% on average.

There are two primary reasons for these improvements. First, higher core counts tend to see even higher inter-core sharing [3], which the SLL TLB exploits. Furthermore, since greater core counts have more on-chip real estate devoted to the TLB, an aggregated SLL TLB has even more entries in a 16-core case than in a 4-core case (2048 entries versus 512 entries). The net effect is that SLL TLBs will be even more useful in future CMP systems with higher core counts.

5.5 Performance Analysis

Up to this point, we have focused purely on TLB hit rates; however, the ultimate goal of our work is to achieve performance benefits. This section sketches a cost-benefit analysis to estimate the performance gains from SLL TLBs against the alternatives. For these experiments, we compare SLL TLB performance against the commercial norm of private L2 TLBs. As previously detailed, our performance analysis is conducted assuming a 4-core CMP. Moreover, since full-run cycle-level simulations would take weeks per datapoint to complete. We instead use a CPI analysis inspired by [24].

Type	Type 1	Type 2	Type 3	Type 4
Description	Flush ROB, Setup insts. TSB Hit in L1 cache Cleanup code	Flush ROB, Setup insts. TSB Hit in L2 Cache Cleanup code	Flush ROB, Setup insts. TSB Hit in DRAM Cleanup code	Flush ROB, Setup insts. TSB Miss 3-level page table walk Cleanup code
Penalty	50 cycles	80 cycles	150 cycles	Beyond 200 cycles

Table 5: Typical TLB miss handler times. After a TLB miss, the reorder buffer (ROB) is flushed, handler setup code is executed, the TSB is accessed and if needed, the page table walk is conducted, followed by cleanup code.

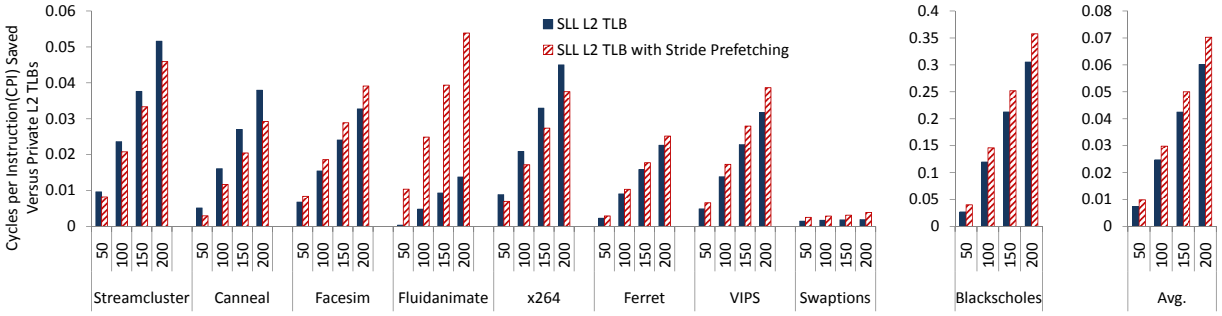

Figure 12: CPI saved by SLL TLBs against private L2 TLBs. Every application benefits from SLL TLBs with exact gains increasing with miss penalties.

While SLL TLBs do provide substantially better hit rates than private L2 TLBs, they also require longer network traversal times. Therefore, it is important to carefully weigh these benefits with access costs. We use Cycles per Instruction (CPI) to assess the performance of SLL TLBs by focusing on CPI saved on TLB miss handling time versus private L2 TLBs. This metric will hold regardless of actual program CPI, which may change across architectures. To compute CPI saved, we need to consider the various costs associated with a TLB miss, how we mitigate them, analytically model these savings and finally produce a range of possible performance benefits. We begin by considering the steps in a typical TLB miss handler. We focus on Solaris TLB handlers in this analysis; however these same steps and strategies are applicable to other miss handling strategies too.

Table 5 details typical TLB miss handler steps, breaking them into four categories. For all the handlers, the reorder buffer (ROB) is flushed upon the interrupt, and handler setup code is executed. In Solaris, this is followed by a lookup in the Translation Storage Buffer (TSB), a software data structure that stores the most recently accessed page table elements. The TSB, like any software data structure, may be cached. A TSB access that hits in the L1 cache minimizes the total handler penalty to roughly 50 cycles (Type 1), while others miss in the L1 resulting in lookups in the L2 cache (Type 2) or DRAM (Type 3), with progressively larger penalties. In the worst case, the requested translation will be absent in the TSB and a full-scale three-level page table walk must be conducted, which takes hundreds of cycles. The exact TLB miss handling times per application will vary depending on the mix of these miss types. Therefore, rather than focusing on a single miss handler value, we now analyze SLL TLB performance across a range of possible average handler times. We vary from the optimistic case of 50 cycles to the more realistic of 100-150 cycles and beyond to 200 cycles.

We note here that handling TLB misses in software is one of a number miss handling strategies currently employed commercially. In particular, a number of commercial systems employ hardware-managed TLBs [12]. In these systems, a TLB miss is handled by a dedicated finite state machine which walks the page table on a miss. The additional hardware provides for faster page table walks than software miss handlers but has less flexible OS management of page tables [13]. These systems enjoy TLB miss handling latencies around the 50 cycle mark [12]. Therefore, our subsequent analysis which shows CPI numbers for TLB miss latencies of 50 cycles also provides information about SLL TLB performance for hardware-managed TLB walks.

Figure 12 plots the CPI saved by our approach versus the commercial norm of private L2 TLBs when using the baseline SLL TLB and its prefetching-augmented counterpart. For each application, CPI counts are provided for TLB miss penalties ranging from 50 to 200 cycles in increments of 50. As shown, *every* parallel benchmark benefits with the SLL TLB, even under the assumption that all handlers are executed in 50 cycles. Assuming a more realistic average miss penalty of 150 cycles, the average benefits are roughly 0.05 CPI, and as high as 0.25 CPI for `Blackscholes`. The exact benefits also vary for the scheme used; for example, `Fluidanimate` particularly benefits with the prefetcher-augmented SLL TLB. Moreover, the gains become more substantial as miss penalties increase.

Therefore, even with optimistically low TLB miss penalties, our SLL TLB outperforms private L2 TLBs, despite using merely the same total hardware. As such, SLL TLBs are an effective and elegant alternative to private L2 TLBs. To further show their utility, we now focus on multiprogrammed sequential workloads.

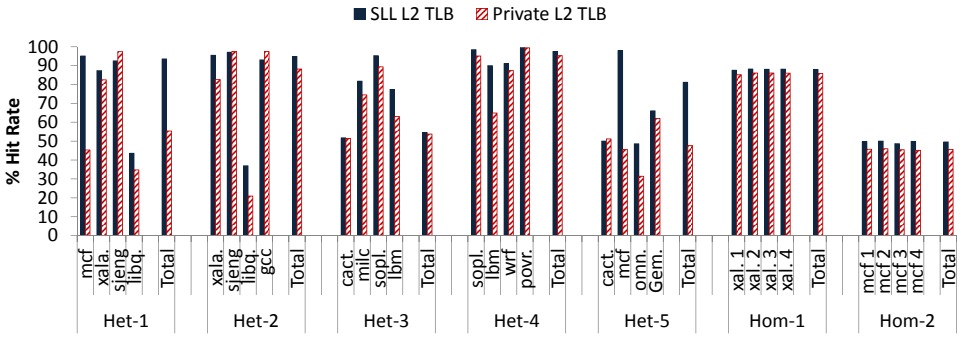

Figure 13: Hit rates for the multiprogrammed workloads for both the SLL L2 TLB and the private L2 TLBs. SLL TLB hit rates in total for each heterogeneous workload combination are substantially higher than private for L2 TLBs (on average, by 21%). Furthermore, high-stress applications like `mcf` see vast improvements without noticeably degrading lower-stress applications. Even homogeneous workload combinations see hit rate increases with SLL TLBs.

6 SLL TLBs: Multiprogrammed Workload Results

We now study SLL TLBs for workloads comprised of sequential applications, running one per core in a multiprogrammed fashion. Section 6.1 quantifies L2 TLB hit rates for the five heterogeneous and two homogeneous workloads. Compared to private, per-core L2 TLBs, we show both per-application and across-workload benefits. For the heterogeneous workloads, we study how effectively a single shared last-level TLB adapts to simultaneously-executing applications with different memory requirements. We also use homogeneous workloads to study SLL TLB benefits when multiple programs of similar nature execute.

After studying application hit rates, Section 6.2 details the performance gains derived from SLL TLBs versus private L2 TLBs. As with parallel workloads, this section performs a cost-benefit analysis and quantifies CPI saved using our approach.

6.1 Multiprogrammed Workloads with One Application Pinned per Core

Figure 13 quantifies SLL L2 and private L2 TLB hit rates for the five heterogeneous (Het-1 to Het-5) and two homogeneous workloads (Hom-1 and Hom-2) described. For every workload combination, we separately plot TLB hit rates for each sequential application, and also show total TLB hit rates across all applications.

First, we study hit rates for the heterogeneous workloads. As shown, both SLL TLBs and per-core, private L2 TLBs eliminate a large fraction of the L1 TLB misses (35% to 95% for the SLL TLBs on average). Furthermore, we find that for every workload combination, total SLL TLB hit rates are higher than the private L2 hit rates. On average, the SLL TLB eliminates 21% additional L1 misses over private L2 TLBs for heterogeneous workloads, a substantial improvement. These increases occur because the SLL L2 TLB is able to allocate its resources flexibly among applications differing in memory requirements. In contrast, the private, per-core L2 TLBs provide fixed hardware for all applications, regardless of their actual needs.

Second, and more surprisingly, Figure 13 shows that SLL TLBs do not generally degrade hit rates for lower-stress application when running with high-stress ones. One might initially expect high-stress benchmarks to capture a larger portion of the SLL TLB, lowering other applications hit rates significantly. However, for example in Het-1, while `mcf` hit rates for SLL TLBs increase by 50% over the private TLB, `xalancbmk` and `libquantum` still enjoy hit rate increases of 5% and 9% respectively. This behavior is also seen across all the other workload combinations, particularly in Het-5, where `mcf` on the SLL TLB enjoys a 52% hit rate increase while *every* other application in the workload also sees a hit rate increase. This occurs because the low-stress applications experience short bursts of TLB misses. Therefore, while the SLL TLB generally provides more mapping space to high-stress applications like `mcf`, it also rapidly adapts to these bursty periods, providing the lower-stress applications with the TLB space they require. The result is that SLL TLBs show notable improvement over private L2 TLBs for the workload combinations in general, improving high-stress applications without substantially degrading lower-stress ones (and usually improving them too).

Third, Figure 13 also compares the SLL TLB hit rates versus private L2 TLB hit rates for the homogeneous workloads, showing 2% to 4% improvements. As expected, the hit rates are consistent for all four cores. Because each core now places an equal demand on the SLL TLB, essentially equally dividing the entries among them, we expect little benefit from this approach. However, even in this case, we find that SLL TLBs marginally increase hit rates over the private L2 TLBs. This occurs because the four benchmarks do not run in exact phase; therefore, the short-term needs of each program vary enough to take advantage of the flexibility that SLL TLBs provide in allocating entries among applications. Moreover, the OS may occupy proportionally less space in the SLL TLB than it does in each of the private L2 TLBs, giving more overall room for the benchmarks to operate. These effects result in the improvement of SLL TLBs against private TLBs for both homogeneous workloads.

Therefore, our results strongly suggest that the SLL TLB demonstrates far greater flexibility in tailoring the total hardware that private L2 TLBs use to the demands of various simultaneously executing sequential workloads. The result is that both total workload hit rates and per-application hit rates enjoy increases.

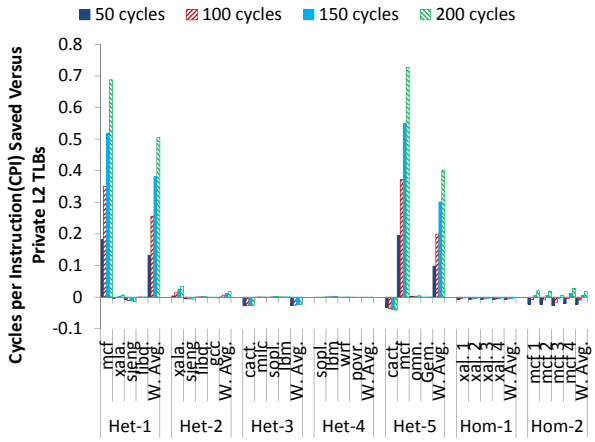

Figure 14: CPI saved using SLL TLBs versus private L2 TLBs for individual applications and per-workload averages. Higher TLB miss penalties result in greater performance gains.

6.2 Performance Analysis

The previous section showed that sequential applications actually benefit from SLL TLBs in terms of hit rate relative to private L2 TLBs. However, since hit penalties for an SLL TLB are higher than for the private L2 TLB, it is important to conduct a cost-benefit analysis of the sources of TLB overhead and how we mitigate them. Therefore, we now extend the parallel program performance analysis based on the TLB handling times described in Section 5.5 to multiprogrammed combinations of sequential workloads. Again, the focus is on understanding CPI saved using our approach for a realistic range of TLB miss penalties, with a methodology inspired by [24].

Figure 14 shows the CPI saved from SLL TLBs relative to private per-core L2 TLBs for individual applications and per-workload averages. While the individual application CPIs may be computed using their particular TLB miss rates, the per-workload averages are based on weighting the L1 TLB miss rates for each constituent sequential program. The results are shown assuming miss penalties ranging from 50 to 200 cycles, in increments of 50 cycles.

Figure 14 shows that across the heterogeneous workloads, higher hit rates typically correspond to increased performance for the per-workload averages. In particular, Het-1 and Het-5 see notable CPI savings. The SLL TLB also provides CPI savings to Het-2, albeit more muted, while Het-4 sees little change. These trends can be better understood by the nature of the application mixes. The SLL TLB typically provides the most benefit in workload mixes where a high-stress application runs with lower-stress ones. In this case, the private L2 TLBs allocate unused resources to the low-stress applications, while the high-stress application suffers. SLL TLBs, on the other hand, can better distribute these resources among the sequential applications, aiding the high-stress workload without hurting the lower-stress ones. This behavior is particularly prevalent for Het-1 and Het-5, in which `Mcf` suffers in the private L2 TLB case. In the presence of the SLL TLB, however, `Mcf` increases in performance without hurting the other applications in Het-1 and only marginally degrades `cactusADM` in Het-5. This leads to a CPI savings approaching 0.2, even at the smallest TLB penalty of 50 cycles. As expected, benefits become even more pronounced at more realistic TLB miss penalties around 100 to 150 cycles.

Figure 14 also shows that `cactusADM` sees lowered performance in Het-3 and Het-5. This is surprising since `cactusADM` is a high-stress TLB application; one may therefore have expected that an SLL TLB would be highly beneficial. In reality, `cactusADM` has been shown to have extremely poor TLB reuse and hence experiences unchanging hit rates even as TLB reach is increased [17, 31]. Therefore, our larger SLL TLB only marginally increases its hit rate (see Figure 13) and is unable to overcome the additional access penalty relative to private L2 TLBs. This means that `cactusADM` suffers a marginal performance degradation. Nevertheless, `cactusADM` is a well-known outlier in this regard [17, 31]; the large majority of applications show better TLB reuse characteristics, making them likely to improve performance with SLL TLBs.

Finally, as expected, Hom-1 and Hom-2 change little with the SLL TLB. Since individual benchmarks in these workloads equally stress the SLL TLB, none sees a significant increase in available entries. Therefore, performance is only marginally decreased due to the additional access time, and these homogeneous workloads are likely to represent the worst-case for SLL TLBs. Overall SLL TLBs provide significant performance improvements for parallel and some heterogeneous sequential workloads, while being largely performance-neutral on others. This makes them an effective and low-complexity alternative to per-core L2 TLBs.

7 Conclusion

This paper shows the benefits of SLL TLBs for both parallel and multiprogrammed sequential workloads. We find that readily-implementable SLL TLBs exploit parallel program inter-core sharing to eliminate 7-79% of L1 TLBs misses, providing comparable benefits to ICC prefetchers. They even outperform conventional per-core, private L2 TLBs by an average of 27%, leading to runtime improvements of as high as 0.25 CPI. Further integrating stride prefetching provides increased hit rates (on average 5%). In addition, SLL TLBs also, somewhat surprisingly, can improve performance for multiprogrammed sequential workloads over private L2 TLBs. In fact, improvements over private L2 TLBs are 21% on average, with higher hit rates also experienced per application in a workload mix. This can lead to as high as 0.4 CPI improvements.

Ultimately, this work may be used to design SLL TLBs in modern processors. Our results provide insight to both sequential and parallel software developers on the benefits expected from this approach, using low-complexity hardware.

8 Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Joel Emer for discussions on this work. This material is based upon work supported by

the National Science Foundation under Grant No. CNS-0627650 and CNS-07205661. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Finally, we acknowledge the support of a research gift from Intel Corp.

References

[1] Advanced Micro Devices. *http://www.amd.com*.
[2] T. Barr, A. Cox, and S. Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ISCA*, 2010.
[3] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *PACT*, 2009.
[4] A. Bhattacharjee and M. Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. *ASPLOS*, 2010.
[5] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *PACT*, 2008.
[6] J. B. Chen, A. Borg, and N. Jouppi. A Simulation Based Study of TLB Performance. *ISCA*, 1992.
[7] D. Clark and J. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Trans. on Comp. Sys.*, 3(1), 1985.
[8] E. Ebrahimi et al. Fairness via Source Throttling: a Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. *ISCA*, 2010.
[9] G. Hinton. The Microarchitecture of the Pentium 4. *Intel Technology Journal*, 2001.
[10] H. Huck and H. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. *ISCA*, 1993.
[11] Intel Corporation. *http://www.intel.com*.
[12] B. Jacob and T. Mudge. A Look at Several Memory Management Units: TLB-Refill, and Page Table Organizations. *ASPLOS*, 1998.
[13] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 1998.
[14] G. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. *Sigmetrics*, 2002.
[15] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *ISCA*, 2002.
[16] C. Kim, D. Burger, and S. Keckler. NUCA: A Non-Uniform Cache Architecture for Wire-Delay Dominated On-Chip Caches. *IEEE Micro Top Picks*, 2003.
[17] W. Korn and M. Chang. SPEC CPU2006 Sensitivity to Memory Page Sizes. *ACM SIGARCH Comp. Arch. News*, 35(1), 2007.
[18] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Comp. Arch. News*, 2005.
[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A Tool to Model Large Caches. *HP Labs Tech Report HPL-2009-85*, 2009.
[20] D. Nagle et al. Design Tradeoffs for Software Managed TLBs. *ISCA*, 1993.
[21] A. Phansalkar et al. Subsetting the SPEC CPU2006 Benchmark Suite. *ACM SIGARCH Comp. Arch. News*, 35(1), 2007.
[22] X. Qui and M. Dubois. Options for Dynamic Address Translations in COMAs. *ISCA*, 1998.
[23] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. *Trans. on Mod. and Comp. Sim.*, 1995.
[24] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-Based TLB Preloading. *ISCA*, 2000.
[25] A. Sharif and H.-H. Lee. Data Prefetching Mechanism by Exploiting Global and Local Access Patterns. *Journal of Instruction-Level Parallelism Data Prefetching Championship*, 2009.
[26] Standard Performance Evaluation Corporation. *http://www.spec.org/cpu2006*.
[27] Sun. UltraSPARC III Cu User's Manual. 2004.
[28] Sun Microsystems. *http://www.sun.com*.
[29] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. *ASPLOS*, 1994.
[30] Virtutech. Simics for Multicore Software. 2007.
[31] D. H. Woo et al. An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth. *HPCA*, 2010.