TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs

DANIEL LUSTIG, Princeton University
ABHISHEK BHATTACHARJEE, Rutgers University
and MARGARET MARTONOSI, Princeton University

Translation Lookaside Buffers (TLBs) are critical to overall system performance. Much past research has addressed uniprocessor TLBs, lowering access times and miss rates. However, as chip multiprocessors (CMPs) become ubiquitous, TLB design and performance must be re-evaluated. Our paper begins by performing a thorough TLB performance evaluation of sequential and parallel benchmarks running on a real-world, modern CMP system using hardware performance counters. This analysis demonstrates the need for further improvement of TLB hit rates for both classes of application, and it also points out that the data TLB has a significantly higher miss rate than the instruction TLB in both cases.

In response to the characterization data, we propose and evaluate both Inter-Core Cooperative (ICC) TLB prefetchers and Shared Last-Level (SLL) TLBs as alternatives to the commercial norm of private, per-core L2 TLBs. ICC prefetchers eliminate 19% to 90% of data TLB (D-TLB) misses across parallel workloads while requiring only modest changes in hardware. SLL TLBs eliminate 7% to 79% of D-TLB misses for parallel workloads and 35% to 95% of D-TLB misses for multiprogrammed sequential workloads. This corresponds to 27% and 21% increases in hit rates as compared to private, per-core L2 TLBs, respectively, and is achieved this using even more modest hardware requirements.

Because of their benefits for parallel applications, their applicability to sequential workloads, and their readily-implementable hardware, SLL TLBs and ICC TLB prefetchers hold great promise for CMPs.

Categories and Subject Descriptors: B.3.2 [**Hardware**]: Memory Structures—*Design Styles*; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Translation Lookaside Buffer, Shared Last-Level TLB, TLB Prefetching, Simulation, Performance Evaluation

ACM Reference Format:
ACM Trans. Architec. Code Optim. V, N, Article A (January 2013), 31 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Translation Lookaside Buffers (TLBs) are performance-critical structures used to cache address translation information for virtual memory systems. Since every instruction requires at least one translation (for the instruction fetch itself), it is essential that these structures be designed to operate quickly and efficiently in order to avoid placing them into the critical path. The primary way of achieving this goal is to increase the TLB hit rate as much as possible, thereby avoiding costly TLB miss penalties. While previous work has explored TLB placement [Chen et al. 1992; Qui and Dubois 1998], size and associativity [Chen et al. 1992], and enhancements such as superpaging [Qui and Dubois 1998] and prefetching [Kandiraju and Sivasubramaniam 2002b; Saulsbury et al. 2000], these proposals generally focus on traditional uniprocessors. However, as chip multiproces-

Authors' addresses: D. Lustig, email: dlustig@princeton.edu; A. Bhattacharjee, email: abhib@cs.rutgers.edu; M. Martonosi, email: mrm@princeton.edu.
Extension of Conference Papers: (1) Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors", *17th International Symposium on High Performance Computer Architecture*, Feb. 2011. (2) Abhishek Bhattacharjee and Margaret Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors", *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.
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.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Table I. System parameters used to collect statistics using hardware performance counters.

Property	Real System
System	8-core (2x HT) x86 (Core i7) 64-bit
OS	Ubuntu Desktop 10.04.2
Private L1 TLBs	Separate I & D, 7-entry fully-assoc TLB, 64-entry 4-way TLB
L2 TLBs	Unified I & D, 512-entry, 4-way TLB

sors (CMPs) are now the dominant paradigm, it is critical to explore TLB design and performance for this particular setting.

Contemporary chip multiprocessors only maintain TLBs that are private to a particular core. These TLBs are often organized in a multilevel hierarchy, with a smaller L1 TLB close to the core and a larger L2 TLB farther away, although even in this case all levels remain per-core private. Furthermore, although previous work has shown that there is often significant redundancy and predictability in TLB misses across cores [Bhattacharjee and Martonosi 2009], there has been little work to exploit this knowledge.

We propose two new TLB enhancements. First, we explore *Inter-Core Cooperative (ICC) TLB prefetchers*, which communicate information about strided access patterns among cores in order to predict future references. Second, we demonstrate the benefits of replacing the set of private L2 TLBs with a single *Shared Last-Level* (SLL) TLB for both parallel and multiprogrammed sequential workloads. Finally, we present a combined solution by adding a simple prefetcher to an SLL TLB to measure the added benefits. Our specific contributions are as follows:

— We perform a comprehensive characterization of instruction and data TLB miss rates for parallel workloads and multiprogrammed combinations of sequential workloads. On a modern multicore system, we show that at the extreme end of each set, `Canneal` of PARSEC and `mcf` of SPEC CPU2006 show overall miss rates of 19.7 and 51.9 data TLB (D-TLB) misses per thousand instructions, respectively, while instruction TLB (I-TLB) misses are universally orders of magnitude smaller.

— We demonstrate that two forms of ICC prefetching, *Leader-Follower* prefetching and *Distance-based* prefetching can individually eliminate up to 57% and 89% of total D-TLB misses, respectively. As we will discuss, these prefetchers have a variety of hardware and/or software implementations, and are beneficial for a variety of parallel workloads.

— We propose *SLL TLBs* as a replacement for the current standard of per-core private TLBs, and we show that this new design leads to an average reduction in D-TLB miss rate of 27% for parallel workloads and 21% for sequential workloads. Using a performance model of cycles per instruction (CPI), we translate these miss-rate improvements into performance savings of up to 0.25 and 0.4 cycles per instruction, respectively.

The rest of the paper is structured as follows. Section 2 motivates the work by presenting the results of real-system workload characterization. Section 3 presents background and related work. Sections 4 and 5 describe the implementations of ICC prefetchers and SLL TLBs, respectively. The experimental methodology is described in Section 6. Section 7 presents the results of our experiments with ICC TLB prefetchers. Sections 8 and 9 show results for SLL TLBs for parallel and multiprogrammed sequential workloads, respectively. Finally, Section 10 concludes the paper.

2. MOTIVATION

In order to motivate the need for improvement in the performance of TLBs on modern systems, we begin with a thorough characterization of TLB performance across a wide range of benchmarks on a modern CMP. To achieve this, we use hardware performance counters to measure SPEC CPU2006 and PARSEC TLB miss rates without otherwise interfering with the normal operation of the CPU.

We measure the TLB statistics on the system described in Table I over the full run of the SPEC benchmarks and over the region of interest (as defined in each workload) for PARSEC benchmarks[1]. To reduce variability, we each experiment five times and present results as means with standard devi-

[1]Due to compilation issues on our platform, we do not present results for `freqmine`, `perlbench`, `deall`, and `wrf`

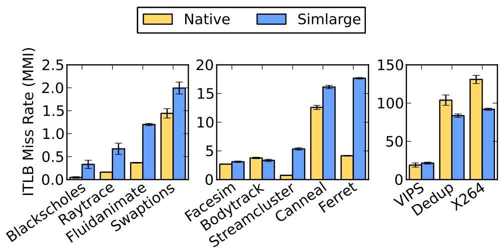

Fig. 1. I-TLB misses per million instructions (MMI) for PARSEC workloads, comparing *native* to *simlarge*, as measured using hardware performance counters.

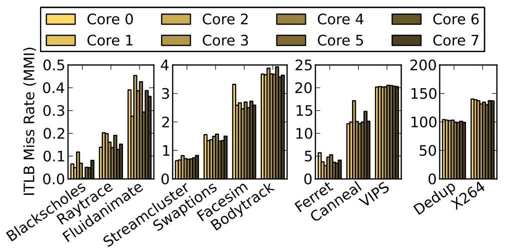

Fig. 2. I-TLB misses per million instructions (MMI) for PARSEC workloads, using *native* inputs, comparing across each core, as measured using hardware performance counters.

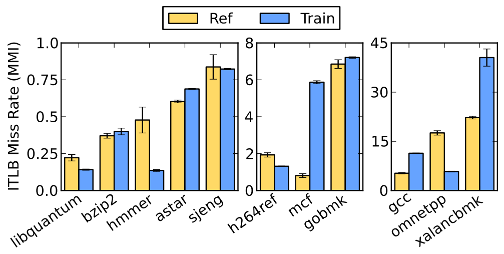

Fig. 3. I-TLB misses per million instructions (MMI) for SPEC CPU2006 INT workloads, comparing *ref* to *train*, as measured using hardware performance counters.

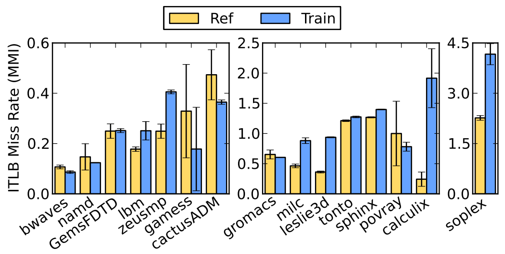

Fig. 4. I-TLB misses per million instructions (MMI) for SPEC CPU2006 FP workloads, comparing *ref* to *train*, as measured using hardware performance counters.

ation error bars[2]. Lastly, we disable address space layout randomization (ASLR), as it dramatically increases the variability of the results when enabled.

2.1. Instruction TLB Performance

We start by characterizing the instruction TLB (I-TLB) performance. Figure 1 presents the number of misses per million instructions (MMI) seen by the I-TLB for the PARSEC benchmark suite. For each program, we plot results for the two largest input sets. Unlike the D-TLB data to follow, I-TLB miss rates are generally quite small across all benchmarks, with even `x264` (the upper extreme) only missing in the I-TLB roughly once every 7700 instructions.

The PARSEC I-TLB results can be further analyzed on a core-by-core basis, as depicted in Figure 2. This shows the miss rate for each individual core for the *native* input. Most of the workloads see fairly consistent rates among the cores, although in cases such as `Blackscholes` and `Fluidanimate` there are visible differences. Again, this diversity is very workload-dependent, as some benchmarks can inherently load-balance their threads better than others. Nevertheless, as PARSEC was designed to represent a set of workloads for CMPs, the load distribution is generally very balanced.

Figures 3 and 4 display the miss rates for the SPEC CPU2006 workloads. The *ref* input set contains the complete inputs used for real-system measurements, and *train*, which is a scaled-down alternative. These benchmarks also show rather low I-TLB miss rates. In fact, the highest overall I-TLB miss rate of all SPEC benchmarks, roughly 23 MMI for `xalancbmk`, is still almost a full order of magnitude lower than the `x264` workload of PARSEC. The SPEC FP workloads have I-TLB miss rates which are lower still.

In general, for both benchmark suites, the I-TLB miss rates are already low enough that only limited benefits would be derived from improving them. For this reason, the remainder of our work will focus on the data TLB.

2.2. Data TLB Performance

Although I-TLB miss rates are generally low enough to be almost negligible, data TLB (D-TLB) miss rates are orders of magnitude higher and therefore would benefit greatly from improved TLB

[2]We also ran tests at different operating system runlevels, but the changes in the results were small or even negligible, so we omit the comparison here.

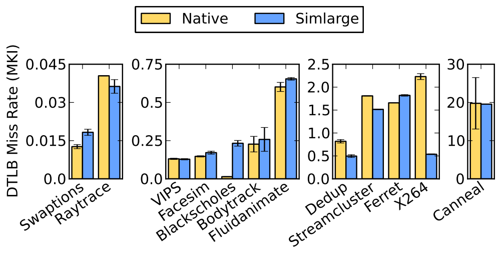

Fig. 5. D-TLB misses per thousand instructions (MKI) for PARSEC workloads, comparing *native* to *simlarge*, as measured using hardware performance counters.

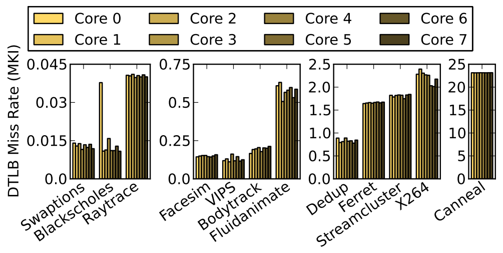

Fig. 6. D-TLB misses per thousand instructions (MKI) for PARSEC workloads, using *native* inputs, comparing across each core, as measured using hardware performance counters.

performance. Figure 5 plots D-TLB miss rates for PARSEC with input sets *native* and *simlarge*. (Note that since D-TLB miss rates are significantly higher than I-TLB, we are now plotting misses per 1000 instructions, or MKI.) Two differences from the I-TLB results are immediately apparent. First, D-TLB miss rates are much higher than I-TLB miss rates. Second, the D-TLB miss rates for the PARSEC workloads vary by multiple orders of magnitude from each other: around 0.15 MKI for `Blackscholes`/*native* to 20 MKI for `Canneal`/*native*. This is useful for our research as it allows us to use the suite to explore a wide range of D-TLB behaviors.

As before, the choice of input set significantly affects the D-TLB miss rate for many of the benchmarks. In particular, `Blackscholes` has a much higher miss rate for *simlarge* than for *native*, while `Dedup` and `x264` decrease just as drastically. Furthermore, `Bodytrack`, `Ferret`, `Dedup`, and `x264` all follow the pipeline-parallel programming model, indicating that the input sets for this model are harder to scale in size than for strictly data-parallel programs [Bienia and Li 2010].

The D-TLB miss rate results for the *native* input set are broken down core-by-core in Figure 6. Even more so than for the I-TLB results, the miss rates among the different cores are generally very consistent within a particular workload, even for the pipeline-parallel programs. For a pipeline to be correctly balanced, a similar amount of data must pass from stage to stage, and this indeed what is reflected in the core-by-core miss rates. One notably inconsistent benchmark, however, is `Blackscholes`. In each trial, most (five to seven) of the cores show a roughly consistent miss rate, while a small number show a higher rate. Furthermore, although the example shows the spike occurring in core 0, this is not consistent; rather, different cores show the spike in different trials.

In order to visualize the relative cost of hits and misses at each level of the TLB, we introduce the *weighted misses per thousand instructions* (WMKI) metric. This combines both kinds of miss into a single value in which each additive component is weighted proportional to its cost in cycles. This approach is needed due to the fact that the two categories cannot be compared directly, as their costs are very different. Using performance counters, we measured the cost of a TLB miss and subsequent page walk to vary between 20 and 40 cycles.[3] Therefore, for this analysis, we assume an average L2 hit penalty of 7 cycles and an average L2 miss penalty of 30 cycles. This leads to the definition

$$\text{WMKI} = \frac{\text{L2 Misses}}{\text{1K Insts.}} + \left(\frac{7}{30} \times \frac{\text{L2 Hits}}{\text{1K Insts.}}\right).$$

These numbers represent typical average values in the system we used for our measurement. Certainly, the cycle cost of individual events may vary, but we use average values in order to assign a valid relative weight to each type of event.

Figure 7 shows the weighted misses per thousand instructions for the PARSEC workloads. Clearly, even though the L2 TLB eliminates a significant number of misses, there is still a non-negligible penalty for L2 D-TLB hits. In fact, this demonstrates that miss rates alone are not the only metric of interest, since that would ignore these L2 hit penalties.[4] Therefore, for example, our performance analysis of proposed SLL TLBs (Sections 8.6 and 9.3), shows that we not only improve the TLB hit rate, but also overcome the added penalty of accessing the SLL TLB.

[3]Because the i7 processor we used for our study did not have hardware counters for page walk cycles, we performed the TLB miss cost analysis on a similar chip which did contain the necessary counters.
[4]The corresponding analysis for the I-TLB showed almost all of the cycle penalty being derived from L2 misses.

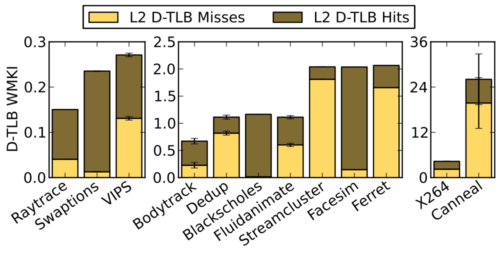

Fig. 7. D-TLB weighted misses per thousand instructions (WMKI) for PARSEC workloads, using *native* inputs, with L1 Miss/L2 Hit rates weighted at 20%.

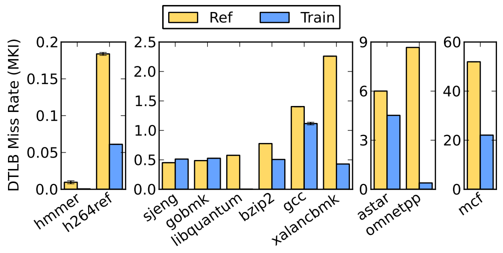

Fig. 8. D-TLB misses per thousand instructions (MKI) for SPEC CPU2006 INT workloads, comparing *ref* to *train*, as measured using hardware performance counters.

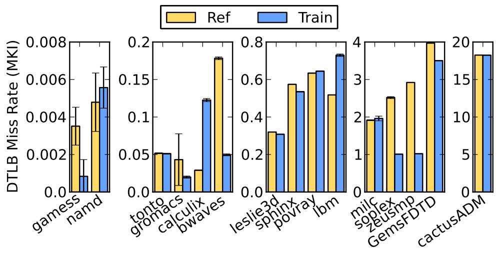

Fig. 9. D-TLB misses per thousand instructions (MKI) for SPEC CPU2006 FP workloads, comparing *ref* to *train*, as measured using hardware performance counters.

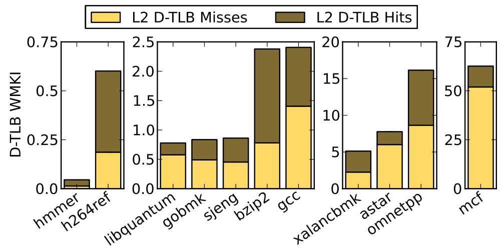

Fig. 10. D-TLB weighted misses per thousand instructions (WMKI) for SPEC INT workloads, using *ref* inputs.

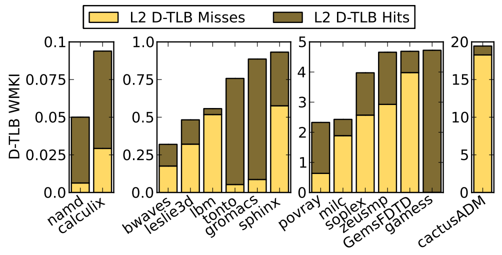

Fig. 11. D-TLB weighted misses per thousand instructions (WMKI) for SPEC FP workloads, using *ref* inputs.

Figures 8 and 9 show the D-TLB miss rates for SPEC CPU2006 *INT* and *FP* benchmarks, respectively. Similarly to PARSEC, the miss rates span orders of magnitude from `gamess` to `mcf`, and in this case the upper D-TLB miss rate limit for the two suites is similar. Again, a large number of the workloads show very different behavior between input sets. There is often a large increase in miss rate from *ref* to *train*; in others there is a large decrease. We therefore use the *ref* for future studies in order to maintain full fidelity. This observation also highlights the importance of PARSEC providing a full and well-characterized collection of input set sizes [Bienia and Li 2010].

Figures 10 and 11 shows the weighted miss rates for SPEC INT and SPEC FP workloads, respectively. Similarly to the PARSEC workloads, the L2 TLB accounts for a large number of hits and introduces a non-negligible access penalty. In fact, FP workloads such as `gromacs` and `gamess` have almost no overall TLB misses, but there is still an impact on performance due to the TLB.

2.3. Key Observations

From this section, we draw three conclusions about TLB behavior, which we use to guide the remainder of our work. First, the miss rates for the data TLB are orders of magnitude higher than those of the instruction TLB, as summarized in Figure 12. We therefore focus on the D-TLB for the rest of this paper. Second, within the multithreaded workloads of PARSEC, the miss rates in both TLBs are very similar across cores. Both of our proposed improvements, ICC prefetching and SLL TLBs, allow the TLBs at each core to share information and resources in a globally beneficial way. Finally, for the D-TLB in particular, even L2 hits incur a penalty and account for significant or even majority portions of the TLB penalty accrued during the run of each benchmark. We therefore create a performance model to show how our proposed improvements overcome this penalty as well.

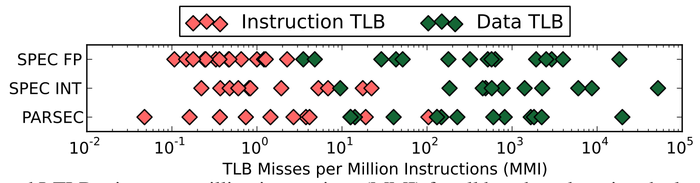

Fig. 12. D-TLB and I-TLB misses per million instructions (MMI) for all benchmarks using the largest input set and. There is a clear distinction between the behavior of the D-TLB and the I-TLB, often by multiple orders of magnitude.

3. BACKGROUND AND RELATED WORK

Contemporary architectures typically maintain private, per-core TLBs placed in parallel with first-level caches [Drongowski 2008; Intel 2012]. Numerous past studies measured TLBs as comprising 5% to 10% of system runtime [Clark and Emer 1985; Kandiraju and Sivasubramaniam 2002b; Nagle et al. 1993; Rosenblum et al. 1995] with extreme cases at 40% [Huck and Hays 1993]. In response, a number of enhancement techniques were proposed. Early work addressed hardware characteristics such as TLB size and associativity [Chen et al. 1992] and superpaging [Talluri and Hill 1994] 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. However, researchers have only very recently started to consider TLBs in the CMP context. UNITD [Romanescu et al. 2010] proposes a mechanism by which TLBs participate in the cache coherence protocol alongside the caches. Synergistic TLBs [Srikantaiah and Kandemir 2010] propose a mechanism by which TLBs on different cores can share entries that might be useful. Their work, however, does not consider a fully-shared structure. The qTLB framework [Tickoo et al. 2007] demonstrates that context switching and contention between processes have an effect on TLB performance on CMPs, and one study has consequently proposed tagging TLB entries with process-specific identifiers [Venkatasubramanian et al. 2009], for architectures which do not already do so. Lastly, the overhead of TLB coherence and shootdowns is also important for many benchmarks [Villavieja et al. 2011].

TLB prefetching schemes have also been explored. For example, recency-based prefetching [Saulsbury et al. 2000] exploits the observation that pages referenced around the same time in the past will be referenced around the same time in the future. In this approach, two sets of pointers are added to each page table entry to track virtual pages referenced in temporal proximity to the current virtual page. While effective, this strategy leads to a larger page table. In response, Kandiraju and Sivasubramaniam [Kandiraju and Sivasubramaniam 2002b] adapt cache prefetching techniques such as Sequential, Arbitrary-Stride and Markov prefetching [Chen and Baer 1995], [Dahlgren et al. 1993], [Joseph and Grunwald 1997]. They propose a distance-based TLB prefetcher which tries to detect repetitive strides as well as the patterns that Markov and Recency prefetching provide, using a modest amount of hardware. Specifically, the distance-based approach tracks the difference or distance between successive TLB miss virtual pages and attempts to capture repetitive distance pairs in the miss stream. On every TLB miss, the goal is to use the distance between the last miss virtual page and current miss virtual page to predict the next expected distance and hence, the next miss virtual page. A prefetch is then initiated for this virtual page.

Recognizing the increasingly critical role of TLBs to system performance, processor vendors have extended the concept of multilevel hierarchies from caches to TLBs. Since the turn of the decade, microarchitectures such as AMD's K7, K8, and K10, Intel's i7, and the HAL SPARC64-III have embraced two-level TLB hierarchies [Drongowski 2008; Intel 2012; Sun 2003]. 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 independent structures private to each core. 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.

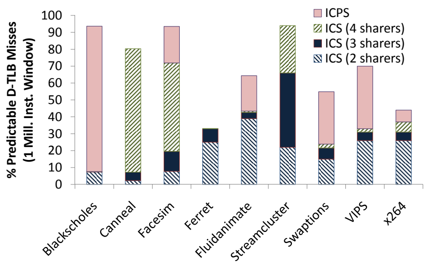

Fig. 13. Number of inter-core shared (ICS) D-TLB misses, per number of sharers, and inter-core predictable stride (ICPS) D-TLB misses. Summing these categories and normalizing to the total misses represents the potential for ICC prefetching to help.

4. TWO INTER-CORE COOPERATIVE TLB PREFETCHERS

As we show, both ICC prefetchers and SLL TLBs drastically eliminate TLB misses by exploiting sharing in parallel programs and allocating resources gracefully among sequential applications. We describe ICC prefetchers first in this section, before covering SLL TLBs in the following section.

4.1. Motivation and Background Data

Effective prefetching must exploit well-characterized and predictable inter-core TLB miss patterns. Previous characterizations [Bhattacharjee and Martonosi 2009] indicate that for parallel workloads, significant commonality exists in TLB miss patterns across cores of a CMP. This leads to two types of *predictable* TLB misses in the system.

Inter-Core Shared (ICS) TLB Misses: In an N-core CMP, a TLB miss on a core is ICS if it is caused by access to a translation entry with the same virtual page, physical page, context ID (process ID), protection information, and page size as the translation accessed by a previous miss on any of the other N-1 cores, within a 1 million instruction window. The number of cores that see this translation is defined as the *number of sharers*. These misses occur often in parallel programs; for example, previous work mentions that 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 [Bhattacharjee and Martonosi 2009]. 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.

Inter-Core Predictable Stride (ICPS) TLB Misses: In an N-core CMP, a TLB miss is ICPS with a stride of S if its virtual page V+S differs by S from the virtual page V of the preceding matching miss (context ID and page size must also match). We require this match to occur within a 1 million instruction window, and the stride S must be repetitive and prominent to be categorized as ICPS. Overall, some benchmarks can see many ICPS misses [Bhattacharjee and Martonosi 2009]. 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.

Figure 13 summarizes the prevalence of these types of predictable D-TLB misses across parallel PARSEC benchmarks, assuming 64-entry D-TLBs. The stacked bars represent the number of ICS D-TLB misses (with separate contributions for different sharer counts) and ICPS D-TLB misses as a percentage of total D-TLB misses. Misses simultaneously in both categories are categorized as ICS misses. As shown, a significant number of TLB misses across the benchmarks are predictable by either ICS misses (e.g., `Canneal`, `Facesim`, and `Streamcluster`) or through ICPS misses caused by a few prominent strides (e.g., over 85% of the D-TLB misses on `Blackscholes` are covered by strides of ±4 pages).

Given these trends, we develop low-overhead techniques to study the behavior of TLB miss patterns on individual cores, gauge whether they are predictable across cores under the ICS or ICPS categories, and then prefetch appropriate TLB entries.

4.2. Prefetching Challenges

Despite the potential benefits of inter-core cooperative prefetching, key challenges remain. First, it is difficult to create a single prefetching scheme that can adapt to diverse D-TLB miss patterns. For example, while PARSEC benchmarks `Canneal` and `Streamcluster` see many shared ICS misses,

Table II. Prominent stride patterns for evaluated benchmarks. Diverse stride patterns mean that distance predictors are likely to outperform simple stride prefetching. The three benchmarks not suited to stride prefetching show good potential for Leader-Follower prefetching.

Benchmark	Strides	Benchmark	Strides	Benchmark	Strides
`Blackscholes`	±4 pages	`Ferret`	None	`Swaptions`	±1, ±2 pages
`Canneal`	None	`Fluidanimate`	±1, ±2 pages	`VIPS`	±1, ±2 pages
`Facesim`	±2, ±3 pages	`Streamcluster`	None	`x264`	±1, ±2 pages

`Blackscholes` is particularly reliant on strided ICPS misses. Moreover, the actual strides among the benchmarks also vary significantly. To see this in greater detail, Table II summarizes the prominent stride values employed by the different benchmarks.

In addition to diverse strides, their distribution among cores may vary. For example, in `Blackscholes` core N+1 misses on virtual page V+4 if core N misses on virtual page V. In contrast, in `VIPS` core 0, 1, and 3 consistently miss with a stride of 1 or 2 pages from core 2. Our implementation must dynamically adapt to these scenarios while also maintaining some level of design simplicity.

A second challenge involves the timeliness of prefetching. On one hand, our scheme requires sufficient time between detecting a TLB miss pattern on one core and using this pattern on another core in order for our prefetchers to react and prefetch the desired entry before use. On the other hand, we must avoid overly-early prefetching which may displace current TLB mappings before they stop being useful. To study this, we have tracked the time between the occurrence of a predictable TLB miss on one core and the subsequent predictable TLB miss on another core. For a 4-core CMP with 64-entry TLBs, this time is between 16K and 4M cycles for 70% of the predictable TLB misses. While this indicates that sufficient time exists for our prefetchers to react to TLB miss patterns, we must be careful that we do not prefetch too early.

Finally, prefetching by its nature causes an increase in memory traffic, and this in turn can effectively lower the available bandwidth for normal requests. However, TLB misses occur at a much lower frequency than do normal cache misses, and so the amount of extra traffic introduced by ICC prefetchers is minimal as compared to normal cache traffic. As a consequence, the performance overhead of the additional memory traffic coming from the ICC prefetchers will be minimal.

4.3. Leader-Follower Prefetching

We now introduce two TLB prefetchers targeting inter-core shared and inter-core predictable stride TLB misses. We begin with the Leader-Follower prefetcher, aimed at eliminating ICS TLB misses.

Leader-Follower prefetching exploits the fact that in ICS-heavy benchmarks, if a core (the *leader*) TLB misses on a particular virtual page entry, other cores (the *followers*) will also typically TLB miss on the same virtual page eventually. Since the leader would already have found the appropriate translation, we can prevent the followers from missing on this entry by pushing it into the followers' TLBs. Key challenges lie in identifying miss patterns and in avoiding pushing mappings onto uninterested cores.

4.3.1. Algorithm. Figure 14 illustrates the algorithm necessary for Leader-Follower prefetching assuming an N-core CMP with per-core D-TLBs. Like many uniprocessor TLB prefetching studies, we do not prefetch entries directly into the TLB, but instead insert them into a small, separate *Prefetch Buffer* (PB) which is looked up concurrently with the TLB. This helps mitigate the challenge of prefetching into the TLB too early and displacing useful information.

Each PB entry maintains a *Valid* bit and a *Prefetch Type* bit (to indicate whether the entry arose from Leader-Follower or Distance-based Cross-Core prefetching) in addition to the *translation entry* (virtual page, physical page, context ID etc.). On a PB entry hit, the particular entry is removed from the PB and inserted into the TLB. The PB uses a FIFO replacement policy; if an entry has to be evicted to accommodate a new prefetch, the oldest PB entry is removed. If a newly prefetched entry's virtual page matches the virtual page of a current PB entry, the older entry is removed and the new prefetch is added to the PB as the newest entry of the FIFO.

Figure 14 separates the Leader-Follower algorithm into two example cases. These cases are independent and can happen in any order. We detail the cases below:

Case 1: Suppose we encounter a D-TLB miss but PB hit on core 0 (step 1a). In response (step 1b), we remove the entry from core 0's PB and add it to its D-TLB.

Case 2: Suppose instead that core 1 sees a D-TLB and PB miss (step 2a). In response, the page table is walked and the translation is located and refilled into the D-TLB. In step 2b, this translation

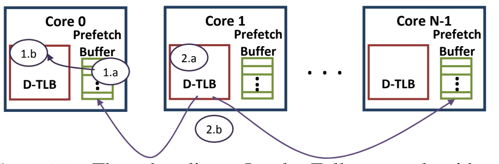

Fig. 14. The baseline Leader-Follower algorithm prefetches a TLB miss translation seen on one core (the leader) into the other cores (the followers) to eliminate inter-core shared TLB misses.

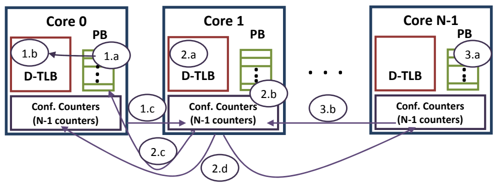
Core 0
PB
1.b
1.a
D-TLB
Conf. Counters (N-1 counters)
1.c
Core 1
PB
2.a
D-TLB
2.b
Conf. Counters (N-1 counters)
2.c
2.d
. . .
3.b
Core N-1
PB
3.a
D-TLB
Conf. Counters (N-1 counters)

Fig. 15. Algorithm for incorporating confidence estimation with saturating confidence counters in Leader-Follower prefetching scheme.

is also prefetched or *pushed* into PBs of the other cores, with the aim of eliminating future ICS misses on the other cores.

In step 2b, at PB insertion time, a check is made to see if the pushed entry already exists. If so, the entry is brought to the head of the PB. However, we do not probe the follower TLBs, and as a result it is possible that the entry may exist in both the D-TLB and the PB simultaneously. In practice, however, we find that this redundancy occurs rarely.

4.3.2. Integrating Confidence Estimation. The baseline Leader-Follower prefetching scheme prefetches a translation into *all* the follower cores every time a TLB and PB miss occurs on the leader core. However, this approach may be over-aggressive and cause *bad* prefetches.

As with standard cache prefetching taxonomy [Srinivasan et al. 2004], we classify a prefetch as bad if it is evicted from the PB without being used. This could happen either because the item was prefetched incorrectly and would never have been referenced even in an infinite PB, or because the finite size of the PB prompts the item to be evicted before its use.

For the Leader-Follower approach, bad prefetching arises due to blind prefetching from the leader to the follower, even if the follower does not share the particular entry. For example, in `Streamcluster`, 22% of the D-TLB misses are shared by 2 cores, 45% by 3 cores, and 28% by all 4 cores. However, for each miss, the baseline approach aggressively pushes the translation into all follower PBs. This can result into two types of bad prefetches, which we classify by extending cache prefetch taxonomy [Srinivasan et al. 2004]. First, the bad prefetch may be *useless* in that it will be unused. Second, the prefetch may be *harmful* in that it will not only be unused, but will also render existing PB entries useless by evicting them too early.

We mitigate harmful and useless prefetches by incorporating confidence estimation. To do so, we add a *CPU Number* field to each PB entry. The CPU Number tracks the leader core responsible for the prefetch of each entry. In addition, as shown in Figure 15, each core maintains confidence counters, one for every other core in the system. Therefore, in our example with an N-core CMP, core 0 has saturating counters for cores 1 to N-1. The figure illustrates three cases of operation for confidence-based Leader-Follower prefetching:

Case 1: Suppose that core 0 sees a PB hit (step 1a). As in the baseline case, step 1b removes the PB entry and inserts it into the D-TLB. In addition, we check, with the Prefetch Type bit, if the entry had been prefetched based on the Leader-Follower scheme. If so, we identify the initiating core (from the CPU number). In our example, this is core 1. Therefore, in step 1c, a message is sent to increment core 1's confidence counter corresponding to core 0 since we are now more confident that prefetches where core 1 is the leader and core 0 is the follower are indeed useful.

Case 2: Suppose instead (step 2a) that core 1 sees a D-TLB and PB miss. In response, the page table is walked and the D-TLB refilled. Then, in step 2b, core 1's confidence counters are checked to decide which follower cores to push the translation to. We prefetch to a follower if its B-bit confidence counter is greater or equal to 2^{B-1}. In our example, core 1's counter corresponding to core 0 is above this value, and hence step 2c pushes the translation into core 0's PB. At the same time, since core 1 itself missed in its PB, we need to increase the rate of prefetching to it. Step 2d therefore sends messages to all other cores so that core 1's confidence counters in the other cores are incremented.

Case 3: Consider the third case in which a PB entry is evicted from core N-1 without being used (step 3a). Since this corresponds to a bad prefetch, we send a message to the core that initiated this entry (step 3b), in this case core 1. There, core 1's counter corresponding to core N-1 is decremented, decreasing bad prefetching.

In step 1c, we send messages from follower to leader immediately after a PB hit so that the leader can learn confidence levels as quickly as possible. As an alternative implementation, one could aim to cut down on message traffic by merging such messages in with the normal eviction process of the original TLB entry. However, such a time delay would likely negate the benefits of confidence tracking. Section 7.4 presents results showing that our implementation of confidence estimation gives dramatic performance improvement for modest hardware.

4.3.3. Key Attributes. In summary, Leader-Follower prefetching has the following key properties. First, the scheme is *shootdown-aware*. If a translation mapping or protection information is changed, initiating a shootdown, TLBs are sent an invalidation signal for the relevant entry. In our scheme, this message is relayed to the PB to invalidate any matching entries. Second, our scheme performs *single-push* prefetches in that a TLB miss on one core results in that single requested translation being inserted into follower PBs. Third, the Leader-Follower mechanism prefetches translations into followers only after the leader walks the page table to find the appropriate translation entry. Therefore, all the translation information is already present when inserted into the follower PBs. Fourth, our scheme does not rely on any predesignation of which cores are leaders or followers. Any core can be a leader or follower for any TLB entry at a time.

4.4. Distance-Based Cross-Core Prefetching

To capture ICPS misses, our solution draws from a uniprocessor distance-based prefetcher [Kandiraju and Sivasubramaniam 2002b], and extends it for cross-core behavior. As an initial example, assume that two CMP cores have the following TLB miss virtual page streams with all of core 0's misses occurring before core 1:

Core 0 TLB Miss Virtual Pages: 3, 4, 6, 7
Core 1 TLB Miss Virtual Pages: 7, 8, 10, 11

Here, a stride of 4 pages repeats between the missing virtual pages on the two cores. But due to timing interleaving and global communication, cross-core patterns are hard to detect and store directly. Instead, our approach focuses on the differences, or *distances*, between successive missing virtual pages on the *same* core, and then makes distance patterns available to *other* cores. For example, the first distance on core 0 is 1 page (page 4 - page 3). Overall, the distances are:

Core 0 Distances: 1, 2, 1
Core 1 Distances: 1, 2, 1

The key to our approach is that although the cores are missing on different virtual pages, they both have the same distance pattern in their misses, and this can be exploited. We therefore design a structure to record repetitive *distance-pairs* - in this case, the pairs *(1, 2)* and *(2, 1)*. Then, on a TLB miss from a core, the *current distance* (current missing virtual page minus last missing virtual page) is used to scan the observed distance pairs. From this, we find the next *predicted distance*, and hence the next virtual page miss. The matching translation entry is then prefetched. In our example, core 0 experiences all its misses, recording the distance-pairs *(1, 2)* and *(2, 1)*. Then, once core 1 misses on pages 7 and 8 (current distance 1), the distance-pair *(1, 2)* reveals that the next virtual page is predicted to be 2 pages away. A subsequent prefetch therefore eliminates the miss on page 10. Similarly, the TLB miss on page 11 is also eliminated (using the *(2, 1)* pair).

4.4.1. Algorithm. Figure 16 shows how Distance-based Cross-Core prefetching works. We again assume an N-core system with prefetches placed into per-core PBs. The approach is as follows:

Step 1: On a D-TLB access, the PB is scanned concurrently to check for the entry. If there is a PB hit, we go to step 2, otherwise we skip directly to step 3.

Step 2: On a PB hit, the entry is removed from the PB and inserted into the D-TLB (in our example, for core 0). We then move to step 3 and follow the same steps as the PB miss case.

Step 3: We now check if the context ID of the current TLB miss is equal to the context ID of the last TLB miss (held in the *Last Ctxt. Reg.*). If so, the current distance is calculated by subtracting the current TLB miss virtual page from the last TLB miss virtual page (held in the *Last VP Reg.*) and we move to step 4. If there is no match, we skip directly to step 8.

Step 4: The core (in our example, core 0) sends the current distance, the last distance (from the *Last Dist. Reg.*), the CPU number, and the current context to the *Distance Table* (DT), which caches frequently used distance-pairs and is shared by all the cores. Our scheme places the DT next to the shared L2 cache.

Step 5: The DT uses the current distance to extract predicted future distances from the stored distance-pairs. It also updates itself using the last distance and current distance.

Step 6: A maximum of *P* predicted distances (the current distance may match with multiple distance-pairs) are sent from the DT back to the requesting core (core 0 in our example), where they

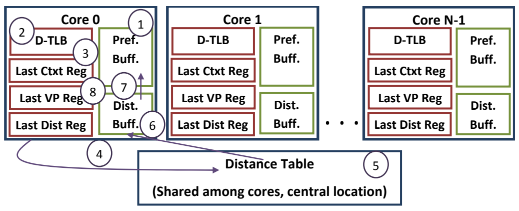

Fig. 16. Distance-based Cross-Core prefetching uses a central, shared Distance Table to store distance pairs and initiates prefetches based on these patterns whenever a TLB miss occurs on one of the cores (for both PB hits and misses). Note that the prefetches on a core may be initiated by a distance-pair initially seen on a different core.

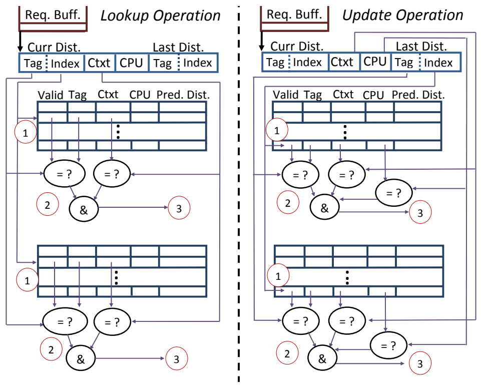

Fig. 17. The Distance Table uses the current distance as the address in the lookup operation and also requires a context match for a lookup hit. The last distance is used as the address for updating with context and CPU number matches also required.

are entered into the *Distance Buffer* (DB). The DB is a FIFO structure with size *P* to hold all newly predicted distances.

Step 7: The predicted distances in the DB are now used by the core (core 0 in our case) to calculate the corresponding virtual pages and walk the page table. When these prefetched translations are found, they are inserted or *pulled* into the PB (unlike the Leader-Follower case, this is a pull mechanism since the core with the TLB miss prefetches further items to *itself* rather than the others).

Step 8: The *Last Ctxt.*, *Last VP*, and *Last Dist.* registers are updated with the current context, current virtual page, and current distance.

A number of options exist for the page table walk in step 7; a hardware-managed TLB could use its hardware state machine without involvement from the workload, which could execute in parallel. In contrast, a software-managed TLB may execute the page table walk within the interrupt caused by the initiating TLB miss. We will compare these approaches in Section 7.7.3.

4.4.2. Distance Table Details. Figure 17 further clarifies DT operations such as lookups (left diagram) and updates (right diagram). Requests are initially enqueued into a *Request Buffer*, global to all cores. Each request is comprised of the current distance, the context, the core number initiating the request, and the last distance value. Moreover, each DT entry has a *Valid* bit, a *Tag* (to compare the distance used to address into the DT), *Ctxt* bits for the context ID of the stored distance-pair, the *CPU* number from which this distance-pair was recorded, and the *Pred. Dist.* or next predicted distance. We now separately detail the steps involved in DT lookup and update.

DT Lookup: For the lookup operation, the low-order bits of the *current distance* index into the appropriate set. Figure 17 shows a 2-way set associative DT, but the associativity could be higher. Second, for all indexed entries, the valid bit is checked and if the tag matches the current distance tag and the *Ctxt* bits match the current context, we have a DT hit. Multiple matches are possible since the same current distance may imply multiple future distances. Third and finally, on a DT hit, the *Pred. Dist.* field of the entry is extracted. Clearly, this DT line may have been allocated by a core different from the requesting core, allowing us to leverage inter-core TLB miss commonality. The maximum number of prefetches is equal to the DT associativity.

DT Update: In contrast to the lookup, DT update uses the low-order bits of the *last distance* to index into the required set. Second, for each line, the valid bit is checked, the tag is compared against the last distance tag portion, and the *Ctxt* bits are compared against the current context. Also, since distances are calculated relative to TLB misses from the same core, we check that the CPU bits of the lines match with the requesting CPU.

If a matching entry (*Valid*, *Tag*, *Ctxt*, and *CPU*) is found, we next check check if updating the *Pred. Dist.* entry with the current distance will result in multiple lines in the set having the same *Tag, Pred. Dist.* pair (this might happen when multiple cores see the same distance-pairs). If true, we avoid storing redundant distance-pairs by not updating the line. If no duplicates exist, we update the *Pred. Dist.* entry with the current distance.

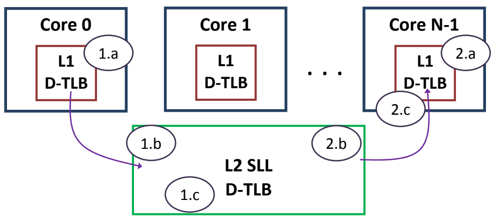

Fig. 18. The basic structure of a shared last-level TLB involves a CMP with private, per-core L1 TLBs and a larger, shared L2 TLB. Cases 1 and 2 detail instances of SLL TLB misses and hits respectively.

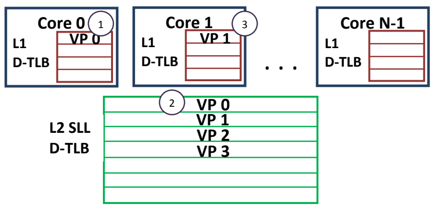

Fig. 19. Enhancing the basic SLL TLB algorithm with simple stride prefetching. When an L1 and SLL TLB miss occur, both the requested translation and other translations a stride of 1, 2, and 3 pages away are inserted into the SLL TLB.

On the other hand, if no matching entry is found, a new line in the set is allocated with the tag, context, and CPU bits set appropriately. For this purpose, the DT uses an LRU replacement policy.

4.4.3. Key Attributes. Like Leader-Follower prefetching, Distance-based Cross-Core prefetching is *shootdown-aware*; PB entries can be invalidated when necessary. Since the DT only maintains distance-pairs and not translations, it is agnostic to TLB shootdowns. Second, this scheme is *multiple-pull*. That is, prefetches for translations are pulled only into the core which experienced the initial TLB miss. Furthermore, multiple prefetches (limited by the associativity of the DT) may be initiated by a single miss. Third, the DT predicts future distances but the corresponding translations need to be found. This differs from the Leader-Follower scheme, in which the leader directly pushes the required translation into the PBs of the other cores. The actual translation search may be accomplished differently for hardware and software-managed TLBs and will be further studied in future sections. Fourth, since the DT induces additional page table walks, we must account for page faults. Our scheme assumes *non-faulting* prefetches in which the page walk is aborted without interrupting the OS if the entry is not found.

5. SHARED LAST-LEVEL TLBS

We now detail our proposed shared last-level TLB. We introduce the concept of SLL TLBs and describe their operation and implementation. We then discuss augmenting SLL TLBs with prefetching mechanisms as well.

5.1. Concept

Figure 18 presents a CMP with private, per-core L1 TLBs backed by an SLL L2 TLB. While this example uses just one level of per-core private TLBs, further levels may be readily 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 of the L1 TLBs. The SLL TLB strives for inclusion with the L1 TLB, so that entries that are accessed by one core are available to others. Figure 18 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 5.2.

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 provide more flexible caching space in which entries can be placed. Eliminations arising from this benefit both parallel and sequential workloads.

5.2. Implementation Options

Having detailed basic SLL TLB operation, 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. Furthermore, we store the full context or process ID with each entry. Space could be saved with fewer bits but our SLL TLB is small, making such optimizations unnecessary. Entries also may or may be pinned in the TLB by the operating system, as is done in the SF3800 described in Section 6.3.

Replacement Policies: To leverage inter-core sharing in parallel programs, the SLL TLB and L1 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 [Hinton 2001]. Instead, we 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. Nevertheless, SLL TLBs could easily be ported to a fully-inclusive hierarchy as well if desired.

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 of 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: Many SLL TLB placement options exist. 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 6), which enjoy short hit times (2 cycles for 45nm technology from CACTI experiments [Muralimanohar et al. 2009]). If future SLL TLBs are considerably larger and require longer hit times, they could be distributed similarly to NUCA caches [Kim et al. 2003].

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 based on CACTI [Muralimanohar et al. 2009] simulations. While this does mean that 22 cycles are spent even on an SLL TLB hit, as we will show, this still vastly improves performance by eliminating a page table walk that could take hundreds of cycles [Jacob and Mudge 1998a; 1998b]. Techniques that reduce the communication latency to reach the SLL TLB will only amplify the SLL TLB benefits.

Finally, since the SLL TLB is centrally shared among all of the cores, they 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 shorter communication time. To ensure that this additional time does not annul the gains from higher SLL hit rates, we assess SLL TLB performance versus private L2 TLBs, which are faster to access 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 the OS page table handler; our design therefore assumes blocking SLL TLBs. Nevertheless, non-blocking SLL TLBs would likely provide even more performance benefits.

5.3. Adding Simple Stride Prefetching to the Baseline Shared Last-Level TLB Operation

As detailed, SLL TLBs provide benefits for parallel programs by capturing inter-core sharing. They also improve multiprogrammed sequential workloads by more efficiently allocating TLB resources to match the varying needs of different sequential workloads. However, we also consider simple stride prefetching extensions to the baseline scheme to further increase TLB hit rates. For example, on a TLB miss, we can insert the requested translation into the SLL TLB and also prefetch entries for virtual pages consecutive to the current one. Figure 19 describes SLL TLBs with prefetching integrated for the following steps:

Step 1: First, we assume that a TLB miss has occurred in both the L1 and SLL L2 TLBs. After 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. At this point, prefetching is activated. To capture potential intra-core and inter-core strides, we now prefetch entries for virtual pages located near the one just missed upon.

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.

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

Benchmark	**Model**	**Wkg. Set**	**Benchmark**	**Model**	**Wkg. Set**
`Streamcluster`	Data-parallel	16MB	`Ferret`	Pipeline-parallel	64MB
`Canneal`	Unstructured	256MB	`VIPS`	Data-parallel	16MB
`Facesim`	Data-parallel	256MB	`Swaptions`	Data-parallel	512KB
`Fluidanimate`	Data-parallel	64MB	`Blackscholes`	Data-parallel	2MB
`x264`	Pipeline-parallel	16MB	`Dedup`	Pipeline-parallel	2MB
`Bodytrack`	Data-parallel	512KB	`Raytrace`	Data-parallel	N/A[6]

Because cache line sizes are larger than translation entries, a single line will maintain multiple translation entries. With 64-byte cache lines and 16-byte TLB entries (see Section 6), entries for three other translations will also reside on the same cache line. Therefore, we prefetch these entries into the SLL TLB, with no additional page walk requirements. Moreover, we permit only *non-faulting* prefetches.

Continuing our example from step 1, after virtual page 0 has been missed upon, we prefetch translations for pages 1, 2, and 3, as these translations reside on the same cache line and therefore arrive for free.

Step 3: Suppose now 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 L2 structure. Fortunately, based on the stride prefetching scheme used, we find that the entry does exist 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.

6. METHODOLOGY AND BENCHMARK CHARACTERIZATION

6.1. Workloads and Input Sets

6.1.1. Parallel Workloads. For parallel applications we use PARSEC, a suite of next-generation shared-memory programs for CMPs [Bienia et al. 2008]. Table III lists the workloads used in this study. Of the 13 available workloads, we are able to compile nine for our simulator[5] and 12 for our real system. The workloads use diverse parallelization strategies (unstructured, data-parallel, and pipeline-parallel) and are run with a thread pinned to each CMP core.

We also classify the benchmarks into groups based on their behavior. Figure 20 arranges the workloads in terms of TLB miss sharing by plotting them with the percentage of ICS misses (at least 2 sharers) on the x-axis and percentage of ICPS misses on the y-axis. Based on this, we form the following categories:

ICPS-h: Stride-reliant workloads with high ICPS misses and low ICS sharing. Only `Blackscholes` is in this category.

ICS/ICPS-m: Moderate but roughly similar contributions from ICS and ICPS misses. `Fluidanimate`, `Swaptions`, and `VIPS` are in this category

ICS-m: Moderate ICS misses and few ICPS misses. `Ferret` and `x264` comprise this category.

ICS-h/ICPS-m: Heavy ICS sharing with moderate ICPS. Only `Facesim` is in this category.

ICS-h: ICS-sharing exclusively, forming a high proportion of the total D-TLB misses. `Canneal` and `Streamcluster` fall in this category.

Specifically, we expect that ICS-high categories particularly benefit from Leader-Follower prefetching while ICPS-high benchmarks exploit Distance-based Cross-Core prefetching.

6.1.2. Sequential Workloads. We run sequential applications from the widely-used SPEC CPU2006 [SPEC 2006] benchmark suite to form our multiprogrammed workloads. For the simulation experiments, we choose to evaluate the workloads designated as capturing the overall performance range of the SPEC CPU2006 suite [Phansalkar et al. 2007]. 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. We therefore draw from the methods and data in [Phansalkar et al. 2007] to form seven workloads of four SPEC CPU2006 applications each.

[5]These are also the PARSEC workloads that are studied in [Bhattacharjee and Martonosi 2009] and hence, serve as a point of reference for our results.

[6]Because `raytrace` is a newer addition to PARSEC, its characterization data is not available in [Bienia et al. 2008].

Table IV. 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.

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 V. System parameters used to collect statistics using hardware performance counters and using simulation.

Property	Simulated Hardware
System	4-16 core SPARC
L1 cache	Private, 32 KB, (4-way)
L2 cache	Shared, 16 MB (4-way)
L3 cache	None
LLC roundtrip	40 cc (uncontested)
Private L1 TLBs	16-entry fully-assoc (locked/unlocked pgs.), 64-entry, 2-way (unlocked pgs.)
L2 TLBs	(see Table VII)
OS	Sun Solaris 10

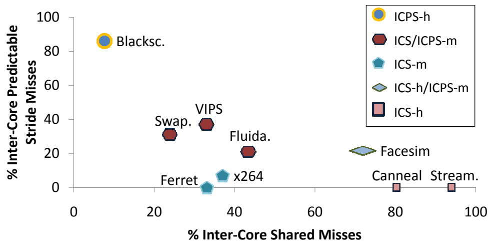

Fig. 20. Based on inter-core sharing, we separate the workloads into ICPS-h, ICS/ICPS-m, ICS-m, ICS-h/ICPS-m, and ICS-h categories.

As shown in Table IV, 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. In this case, `mcf` and `cactusADM` serve as our 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`.

Lastly, we also note that many workloads will consist of multiprogrammed combinations of both sequential and parallel applications. Such a combination leads to interesting questions about partitioning, sharing, interference, etc., and we hope to study such mixes in the future.

6.2. Simulation Infrastructure

We use the Multifacet GEMS simulator [Martin et al. 2005] from Table V. Our simulator uses Virtutech Simics [Virtutech 2007] as its functional model to simulate a 4-16 core CMP based on Sun's UltraSPARC III Cu with SunFire's MMU architecture [Sun 2003]. 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. These sizes are similar to the L1 TLBs of contemporary processors such as Intel's i7 (64-entry) and AMD's K10 (48-entry).

6.3. ICC Prefetcher Evaluation

To evaluate the ICC prefetcher, we consider a variety of MMU configurations, shown in Table VI. Since the simulated MMUs are software-managed, the OS receives an interrupt on every TLB miss. Furthermore, each MMU has a distinct TLB architecture. The SF280R is representative of Sun's entry-level servers with typical TLB sizes, whereas the SF3800 contains one of the largest TLB

Table VI. Simulated SunFire MMUs with software-managed TLBs.

MMU Type	D-TLBs
SF280R	64-entry (2-way)
Intermediate	512-entry (2-way)
SF3800	16-entry, fully-assoc. (locked/unlocked pages) 2 × 512-entry, 2-way (unlocked pages)

Table VII. TLB enhancements evaluated in this work. SLL TLB and private, per-core L2 TLB sizes match those of the ICC prefetchers.

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)	512-entry, 4-way, 22 cc roundtrip (interconnect: 20 cc, access: 2 cc)
ICC Prefetching (Our Strategy)	16-entry PB per core, 512-entry DT, 28 cc DT roundtrip (interconnect: 20 cc, access: 8 cc)

organizations to date. In all cases, we keep the L1 TLB size constant so as not to increase the latency of hits, which are by far the common case.

We develop and evaluate the two prefetching schemes in the following steps:

In Section 7, we evaluate the Leader-Follower and Distance-based Cross-Core prefetching schemes on a 4-core CMP system with the SF280R MMUs (64-entry TLBs). We show the benefits of each scheme individually and then combine them. In the Leader-Follower scheme, we assume that it takes 40 cycles for the leader core to push a translation into the follower core (this is equal to the L2 latency, which may be considerably longer than the actual time taken on interconnection networks with 4-16 cores today). Furthermore, in Distance-based Cross-Core prefetching, we place the DT next to the L2 cache, and hence assume that a DT access is equal to an L2 access latency. Finally, we assume that, as with hardware-managed TLBs, a hardware state machine walks the page table on predicted distances from the DT. In this section, the state machine is assumed to locate the desired translation with an L1 access (subsequent sections address longer page table walks).

After this analysis, we then study the performance implications of these approaches for multiple core counts and TLB sizes. Lastly, we investigate hardware/software prefetcher implementation tradeoffs and assess the benefits and overheads of each approach.

Since TLB misses occur less frequently than cache misses, we use the largest available input data set feasible for simulation, the *simlarge* set. Due to slow full-system timing simulation speeds, we present results observed with 1 billion instructions.

6.4. SLL TLB Evaluation

To assess the benefits of SLL TLBs, we compare them against both per-core, private L2 TLBs and ICC prefetchers with the same total hardware. Based on the ICC prefetchers detailed in Table VII, an equally-sized SLL TLB requires 512 entries. This 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 [Wilton and Jouppi 1994; Muralimanohar et al. 2009] 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).

We again use the full-system 4-core CMP simulator of Table V. For parallel workloads, we again present results for 1 billion instructions of execution. For sequential workloads, we use an approach similar to previous studies [Ebrahimi et al. 2010; Kandiraju and Sivasubramaniam 2002a; Sharif and Lee 2009]: 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 approach allows us to capture larger swaths of execution, and it allows us to use the full *ref* datasets to more fully exercise the TLB than would be possible with smaller input sets. In addition, these multiprogrammed sequential workloads are not as heavily influenced as the parallel ones by inter-thread timing interactions. 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.

7. INTER-CORE COOPERATIVE PREFETCHER RESULTS

We now focus on the benefits of the prefetchers and explore the hardware parameters involved. In Section 7.1, we quantify the benefits of Leader-Follower prefetching and then in Section 7.2, do the same for Distance-based Cross-Core prefetching. Both these cases assume an aggressive

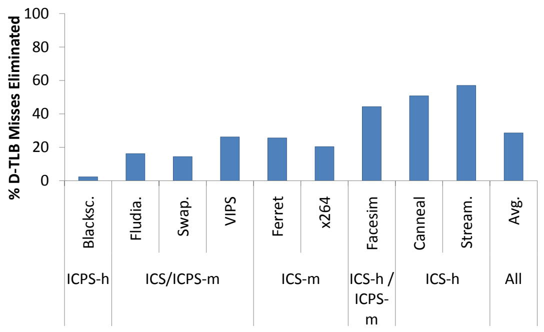

Fig. 21. Percentage of D-TLB misses eliminated with Leader-Follower prefetching with infinite PBs. This scheme performs well for high-ICS benchmarks such as `Canneal`, `Facesim`, and `Streamcluster` but poorly for ICPS-reliant `Blackscholes`.

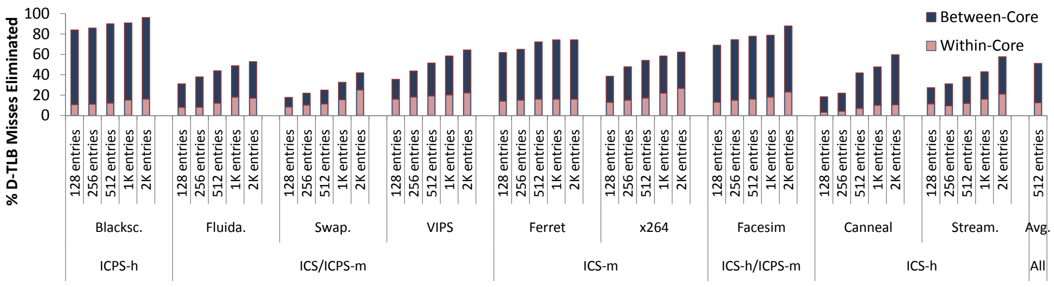

Fig. 22. Percentage of D-TLB misses eliminated with Distance-based Cross-Core prefetching assuming infinite PBs for various sizes of the DT. Note that a high number of misses are eliminated consistently across benchmarks, primarily from *between-core* prefetches.

implementation with infinite PBs and no confidence estimation. In Section 7.3, we then combine both approaches for feasible PB sizes. Subsequently, Section 7.4 shows how confidence estimation reduces bad prefetches for better performance. Finally, Section 7.5 compares our approach against increasing TLB sizes.

7.1. Leader-Follower Prefetching

Figure 21 shows the percentage of total D-TLB misses eliminated using Leader-Follower prefetching, assuming infinite PBs for now. From this, we observe the following:

First, ICS-h and ICS-h/ICPS-m benchmarks `Canneal`, `Facesim`, and `Streamcluster` enjoy particularly high benefits. For example, `Streamcluster` eliminates as much as 57% of its misses.

Second, even benchmarks from the ICS-m and ICS/ICPS-m categories see more than 14% of their D-TLB misses eliminated. For example, `VIPS` eliminates 26% of its D-TLB misses. This means that even moderate amounts of ICS sharing can be effectively exploited by Leader-Follower prefetching.

Unlike their ICS-heavy counterparts, ICPS-reliant benchmarks see fewer benefits. For example, `Blackscholes` sees roughly 3% of its D-TLB misses eliminated. Nonetheless an average of 28% miss reduction occurs across all applications.

7.2. Distance-Based Cross-Core Prefetching

Next, Figure 22 presents results for Distance-based Cross-Core prefetching. It shows D-TLB misses eliminated for various DT sizes with infinite PBs. Assuming a 4-way set-associative DT (therefore, the maximum number of prefetches is 4 and the DB is also set to this value), we vary the size of the DT from 128 to 2K entries. Each bar is further separated into D-TLB misses eliminated from two types of prefetches:

1. *Between-Core* prefetches in which a core prefetches based on a distance-pair in the DT that was recorded from a different core. This is the category that exploits inter-core commonality.

2. *Within-Core* prefetches in which a core prefetches based on a distance-pair in the DT that was recorded from itself.

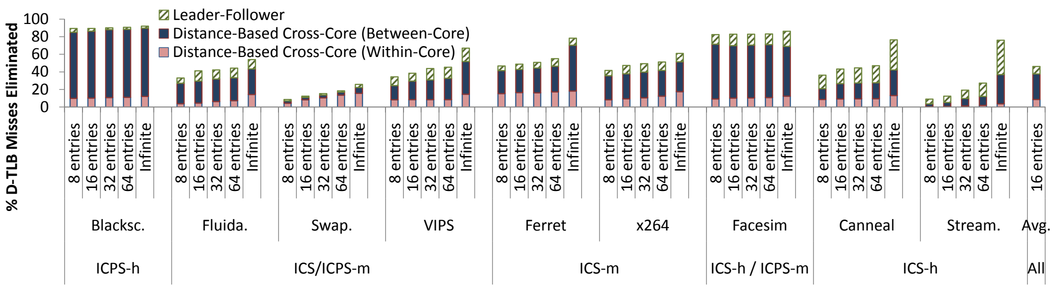

Fig. 23. Effect of combining the two prefetching schemes with finite PBs. Even with as few as 16 entries in the PB, these techniques eliminate an average of 46% of the D-TLB misses.

Figure 22 indicates that miss eliminations rise with bigger DTs. Benchmarks with ICPS TLB misses enjoy particular improvements from this approach. For example, `Blackscholes` (ICPS-h) consistently eliminates more than 80% of its TLB misses.

Second, Figure 22 shows that streaming benchmarks employing regular distance-pairs derive great benefits from Distance-based Cross-Core prefetching. For example, `Facesim`, which employs an iterative Newton-Raphson algorithm over a sparse matrix, sees over 70% of its D-TLB misses eliminated even at the smallest DT. Similarly, `Ferret's` working set is made up of an image database that is scanned linearly by executing threads; hence, regular distance-pairs exist, eliminating above 60% of D-TLB misses.

Third, Distance-based Cross-Core prefetching aids even ICS benchmarks from ICS-m, ICS-h/ICPS-m, and ICS-h categories. For example, `Canneal` enjoys roughly 60% D-TLB miss elimination at 2K entry DTs. ICS-heavy workloads typically benefit most from increased DT size because they have less prominent strides and hence a higher number of unique distance-pairs.

Finally, the high contribution of between-core prefetches demonstrates that the DT actively exploits inter-core commonality. Even in cases where this is less prominent however, the DT can capture within-core distance-pairs, and use them for better performance. For example, `Swaptions` makes particular use of this with half of its D-TLB eliminations arising from within-core prefetches.

Clearly, the bulk of eliminated D-TLB misses across the workloads arises from behavior seen across CMP cores. While uniprocessor distance schemes [Kandiraju and Sivasubramaniam 2002b] may be able to capture some of these patterns, they would take longer to do so, eliminating fewer misses. Moreover, since our scheme uses a single DT to house all distance-pairs across cores, we eliminate the redundancy of a scheme with per-core DTs.

Based on Figure 22, we assume a DT of 512 entries from now on (with an average of 54% of the D-TLB misses eliminated). Moreover, we have experimented with a range of associativities and found that there is little benefit beyond a 4-way DT. Therefore, we assume an associativity, and hence maximum number of simultaneous predictions and DB size, of 4.

Based on this, each DT entry uses a Valid bit, 25 Tag bits, 2 CPU bits (for a 4-core CMP), 13 context bits (from UltraSPARC specifications), and 32 bits for the next predicted distance, amounting to a 4.56 KB DT for 4 cores, or 4.81 KB at 64 cores. Compared to the neighboring L2 cache, the DT is orders of magnitude smaller, making for modest and scalable hardware.

7.3. Combining the ICC Approaches

Since the Leader-Follower and Distance-based Cross-Core schemes target distinct application characteristics, we now evaluate the benefits of both approaches together in a combined ICC TLB prefetcher. Both schemes may be implemented as before, with the PB now shared between both strategies.

Figure 23 shows the benefits of the combined prefetcher for finite PBs of 8 to 64 entries and infinite PBs. In all cases, a 4-way, 512-entry DT with 4-entry DBs is assumed. As expected, the combined ICC prefetcher eliminates 26% to 92% of the D-TLB misses for infinite PBs. Moreover, in every case, the combined approach outperforms either of the approaches individually.

Figure 23 also shows that ICC prefetchers offer notable benefits even for small PB sizes. For example, even modest 16-entry PBs eliminate 13% (for `Swaptions`) to 89% (for `Blackscholes`) of the D-TLB misses, with an average of 46%. Moreover, benchmarks like `Canneal` and `Ferret`, which suffer from a high number of D-TLB misses [Bhattacharjee and Martonosi 2009], see more than 44% of their misses eliminated, translating to significant performance savings.

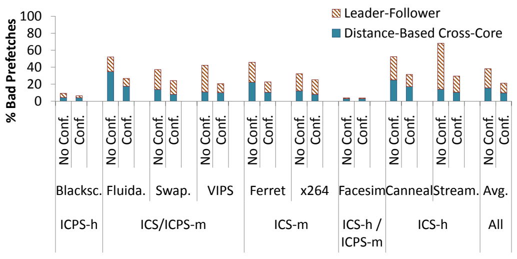

Fig. 24. Percentage of total prefetches that are bad because they are never used or are prematurely evicted from the PB due to its finite size. Without confidence there are many bad prefetches, particularly from the Leader-Follower scheme. However, 2-bit confidence counters fix this, leading to a 2× decrease in bad prefetches.

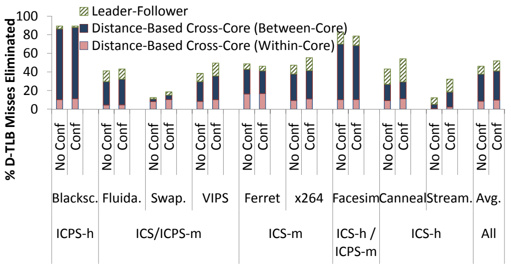

Fig. 25. Percentage of D-TLB misses eliminated with the inclusion of confidence estimation. Not only does confidence estimation reduce bad prefetches, it also improves prefetcher performance by retaining useful information for longer in the PB. On average, 6% additional D-TLB misses are eliminated by incorporating confidence estimation.

Interestingly, Figure 23 shows that ICS-h benchmarks `Canneal` and `Streamcluster` suffer most from decreasing PB sizes. Section 7.4 shows how confidence estimation can mitigate this effect.

Based on Figure 23, we assume a combined ICC prefetcher with a modest PB size of 16 entries for the rest of our evaluations. This represents the smallest of the PB sizes deemed feasible by Kandiraju and Sivasubramaniam [Kandiraju and Sivasubramaniam 2002b].

7.4. Integrating Confidence Estimation

Our results so far assume the absence of confidence estimation described in Section 4.3.2. However, as previously noted, there may be instances of over-aggressive prefetching, especially for the Leader-Follower case in benchmarks like `Streamcluster` in which not all cores share the all the TLB miss translations. Confidence estimation is crucial to the performance of these workloads.

Figure 24 profiles the percentage of total prefetches from our prefetcher without confidence estimation (i.e. the version presented until now) that are bad, and compares this to the case of using confidence with 2-bit counters. Each bar in the graph is divided into Leader-Follower and Distance-based Cross-Core contributions. Without confidence, benchmarks like `Canneal` and `Streamcluster`, which particularly suffer from lowered PB sizes, have the most bad prefetches. Even in other cases without confidence, there are high bad prefetch counts (an average of 38%). Moreover, it is clear that a large proportion of the bad prefetches are initiated by over-aggressive Leader-Follower prefetching. For example, this scheme causes roughly 80% of `Streamcluster's` bad prefetches, with 60% on average across applications.

Figure 24 shows that using just 2-bit confidence counters cuts bad prefetches from an average of 38% to 21% across the workloads. In fact, we see that `Streamcluster's` bad prefetches are halved while `Canneal` also sees substantial benefits. Moreover, while bad prefetches from Leader-Follower prefetching decrease, Distance-based Cross-Core prefetching also benefits because fewer prefetches from this scheme are prematurely evicted due to bad Leader-Follower prefetches. This means that not only are useless prefetches decreased, so too are harmful prefetches.

Figure 25 shows that the decrease in bad prefetches from confidence estimation translates into notable performance improvements. For example, `Canneal` and `Streamcluster` eliminate 10% and 20% more misses with confidence. This is because harmful prefetches are decreased and thus useful information is not prematurely evicted from the PB. At the same time, benchmarks like `Facesim` and `Ferret` see a slight drop of 2% to 3% in D-TLB miss elimination due to the reduced prefetching; however, since the average benefit is a 6% increase in D-TLB miss elimination, we incorporate confidence estimation into our ICC prefetcher.

7.5. Cooperative Prefetching Versus Larger TLBs

To fairly quantify the benefits of prefetching, we must compare our techniques against just enlarging the TLB. Specifically, since we require 16-entry PBs to be checked concurrently with the D-TLBs, we need to compare this approach to adding 16 TLB entries.

Figure 26 plots the benefits of ICC prefetching over blindly adding 16 entries for the 64-entry TLBs (SF280R MMU), 512-entry TLBs (Intermediate MMUs), and 1024-entry TLBs (SF3800 MMUs). For these TLB sizes, we plot the difference between percent D-TLB misses eliminated using ICC prefetching with the baseline size versus adding 16 TLB entries to the baseline case.

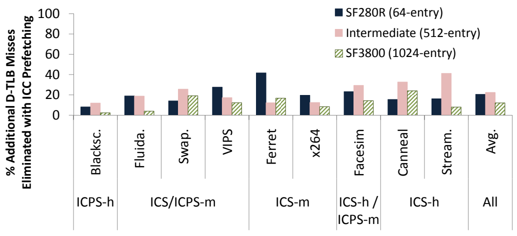

Fig. 26. Percentage additional misses eliminated using ICC prefetching with 16-entry PBs versus just enlarging TLBs by 16 entries. ICC prefetching consistently outperforms enlarged TLBs.

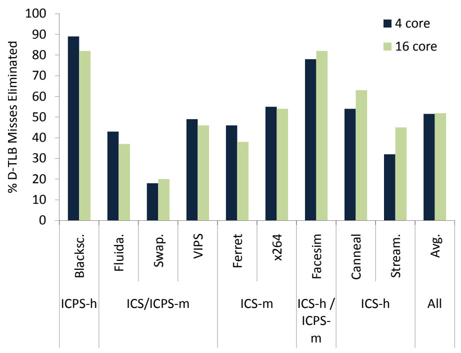

Fig. 27. TLB miss elimination rates using ICC prefetching for 4-core and 16-core CMPs with SF280R MMUs. Note that higher core counts increase benefits for benchmarks like `Canneal` and `Streamcluster` which see more sharing with more cores.

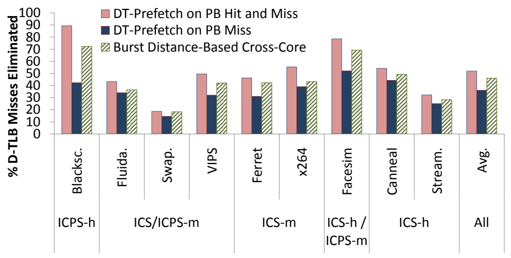

Fig. 28. Burst Distance-based Cross-Core prefetching eliminates almost as many D-TLB misses as the fully-hardware case. Results assume that Leader-Follower prefetching remains unaffected.

Figure 26 shows that ICC prefetching notably outperforms blindly increasing TLB sizes across all sizes and benchmarks. At 64-entry and 512-entry baseline sizes, ICC prefetching outperforms larger TLBs by over 20%. At 1024-entry baseline TLB sizes, benefits are slightly reduced to roughly 12% since TLB misses occur less often, lessening the impact of prefetching. Nevertheless, ICC prefetching outperforms larger TLBs notably even for 1024-entry TLBs. Therefore, prefetching strategies with modest hardware can yield significant gains beyond just enlarging TLBs.

7.6. Moving to Greater Core Counts

When analyzing the benefits of our prefetchers, it is important to gauge their performance in the presence of increasing core counts. While we have so far assumed a 4-core CMP, we now quantify the performance benefits on a 16-core CMP.

Figure 27 compares TLB miss elimination rates for the 4-core CMP against a 16-core CMP for SF280R MMUs. We assume the fully-hardware implementation with 16-entry PBs, hardware Leader-Follower prefetching, and hardware Distance-based Cross-Core prefetching with a 512-entry, 4-way DT.

Figure 27 shows that ICC prefetching improves performance even at greater core counts. However, the exact benefits vary with the benchmarks. Some benchmarks like `Canneal` and `Streamcluster` see benefits rising by about 8% from the 4-core to the 16-core case. This may be attributed to the fact that at higher core counts, inter-core shared TLB misses increase. Overall, these results indicate that prefetching strategies will likely become even more pertinent as CMPs scale to higher core counts.

7.7. Hardware/Software Implementation Tradeoffs

Having assessed the basics of our proposed prefetcher designs, we now discuss a number of hardware/software implementation possibilities for them. Our goal here is to provide insight into implementation issues that hardware and operating system designers will face when integrating ICC prefetching. Our focus here is on a qualitative understanding of these implementations and their impact on performance TLB miss elimination and associated performance. While the specific performance implications will vary based on a number of architectural features (eg. whether we use hardware or software-managed TLBs, whether the page table walk process is an x86-based radix page table walk or an inverted page table, which hardware caches can safely store page table entries), our high-level qualitative analysis will hold across a range of architectures.

7.7.1. Fully-Hardware Implementation. The highest-performance ICC prefetcher implements the prefetcher components (PBs, DT, DBs, and confidence estimation) entirely in hardware. While PB access would require no additional penalty due to its small size and placement next to the per-core TLBs, accessing the DT would incur a penalty similar to the L2 cache.

An additional key issue is page table walk times. While Leader-Follower prefetching pushes the already-available translation into cores, Distance-based Cross-Core prefetching requires page table walks for each DT prediction. A fully-hardware, high-performance prefetching strategy would be possible assuming hardware-managed TLBs, where per-core hardware state machines walk the page table. This means that DT-induced translation searches proceed without OS or program intervention.

7.7.2. Hardware Prefetchers with Software Page Table Walks. While fully-hardware ICC prefetching could be readily accommodated for hardware-managed TLBs, we must also consider implementation possibilities for SW-managed TLBs. In this section, we consider the case where the prefetchers remain fully-hardware units, but page table walks are carried out by dedicated OS interrupt handlers rather than hardware state machines.

While Leader-Follower prefetching remains unaffected for SW-managed TLBs, there are two cases to consider for Distance-based Cross-Core prefetching. In the first case, a core misses in both the D-TLB and PB, causing an OS interrupt. When this happens, the interrupt handler assumes responsibility for conducting page table walks for the suggested distances from the DT. In the second case, a PB hit occurs, and there is no interrupt. At the same time, the DT suggests predicted distances for which page table walks are needed.

A solution is to limit Distance-based Cross-Core prefetches to instances when both the D-TLB and PB miss, because in these cases the OS will be interrupted anyway. In particular, we implement *Burst Distance-based Cross-Core* prefetching. Our scheme performs DT prefetches only when both the D-TLB and PB miss; however, instead of prefetching just the predicted distances relative to the current distance, we use these predicted distances to re-index into the DT and predict future distances as well. Suppose, for example, that a current distance *curr* yields the predicted distances $pred_0$ and $pred_1$. In our scheme, $pred_0$ then re-indexes the DT to find its own set of predicted distances (eg. $pred_3$ and $pred_4$). Similarly, $pred_1$ is then used to index the DT. In effect, our scheme limits prefetches to PB misses but compensates by aggressively prefetching in bursts at this point.

Figure 28 showcases the effectiveness of Burst Distance-based Cross-Core Prefetching in eliminating D-TLB misses, assuming a maximum of 8 DT-induced prefetches for every PB miss. For each workload, we compare this scheme against the conventional Distance-based Cross-Core approach. We also show our benefits versus the option of performing DT prefetches only on PB misses, but prefetching based on just the distances predicted from the current distance. In all cases, a 4-core CMP with SF280R MMUs also using Leader-Follower prefetching is assumed.

Restricting DT prefetches on a PB miss to distances based on the current distance severely reduces ICC prefetching gains. This is especially true for ICPS-heavy benchmarks like `Blackscholes` and `Facesim` which particularly exercise the DT. On average, there is a 15% reduction in benefits against the fully-hardware case where DT prefetches occur for both PB hits and misses.

Fortunately, Figure 28 also shows that Burst Distance-based Cross-Core prefetching addresses this problem effectively for every workload. On average, we eliminate just 5% fewer D-TLB misses than the fully-hardware approach making this a valuable technique for SW-managed TLBs.

In terms of performance implications, designers will need to account for the fact that since all DT-based prefetching will be initiated within the interrupt handler, there will be some modest performance overheads as compared to the fully-hardware case. Nevertheless, high TLB miss elimination counts indicate that this scheme will provide significant benefits.

7.7.3. Hardware/Software Prefetch with Software Page Table Walks. We now discuss the benefits and overheads of also moving prefetcher components into software.

We first decide which components to leave in hardware. Hardware PBs must be retained for concurrent scans with D-TLBs. Furthermore, since Leader-Follower prefetching operates without software intervention, it too can remain a purely hardware operation.

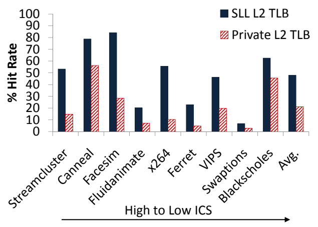

Fig. 29. SLL TLB versus private, per-core L2 TLB hit rates. While private L2 TLBs do provide benefits, they are consistently outperformed by SLL TLBs (by 27% on average).

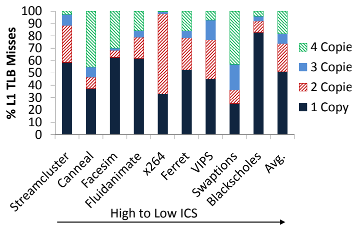

Fig. 30. Copy counts for private L2 TLBs. For every evicted L1 line, we record how many L2 TLBs hold this entry. Heavy replication of entries exists, which SLL TLBs mitigate.

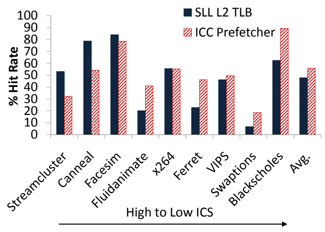

Fig. 31. SLL TLB hit rate versus ICC prefetcher hit rate. Benchmarks with high inter-core sharing like `Canneal`, `Facesim`, and `Streamcluster` benefit the most from SLL TLBs.

In contrast, designers may wish to place the DT purely in software. Since we use Burst Distance-based Cross-Core prefetching, we may access the DT from the interrupt handler and burst-prefetch translations every time a D-TLB and PB miss occurs. Moreover, care must be taken to ensure that the DT, now in software, is pinned in physical memory so that a DT access cannot itself result in a TLB miss.

With the DT held in software, we must not only perform page table walks within the interrupt but also DT lookups as well. This in turn would add modest performance overheads. For the DT organization we consider (512-entry, 4-way), each DT entry requires 73 bits. A 64-byte cache line can easily accommodate 4 DT entries where 4 equals the associativity. Therefore, after the first DT reference, which brings a set into the L1 cache, every access in the set results in an L1 cache hit. For burst-prefetching, in the worst case, we need to access 8 independent sets of the DT, amounting to 8 L2 accesses. However, this would occur rarely since multiple predictions usually arise from the same set. Therefore, while performance may fall short of the hardware prefetchers, substantial performance improvements will be seen using this approach as well.

8. SHARED LAST-LEVEL TLBS: RESULTS FOR PARALLEL WORKLOADS

We now study SLL TLBs for parallel workloads. First, Section 8.1 compares SLL TLBs against commercial per-core, private L2 TLBs. Second, Section 8.2 compares SLL TLBs with ICC prefetching. Section 8.3 analyzes sharing patterns of entries in SLL TLBs while Section 8.4 considers the benefits of enhancing the baseline SLL TLB operation with stride prefetching. Section 8.5 then studies the benefits of the SLL TLB with increasing core counts. Finally, Section 8.6 focuses on the performance implications of our results.

8.1. Shared Last-Level TLBs versus Private L2 TLBs

Figure 29 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 [Bhattacharjee and Martonosi 2009]. 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, we observe that 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 29 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 30 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.

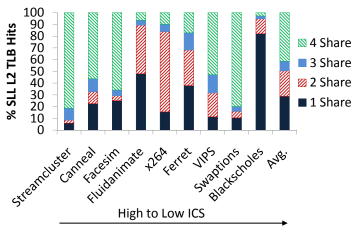

Fig. 32. Sharing characteristics of each SLL L2 TLB hit entry. Note that high-ICS applications like `Canneal`, `Facesim`, and `Streamcluster` see high SLL inter-core sharing.

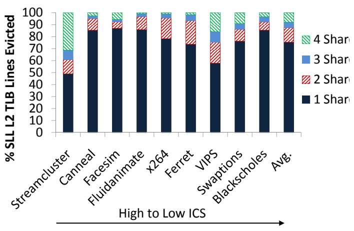

Fig. 33. Sharing patterns of SLL TLB entries evicted. As shown, most evicted entries are unshared; inter-core sharing increases priority in replacement algorithm, decreasing eviction likelihood.

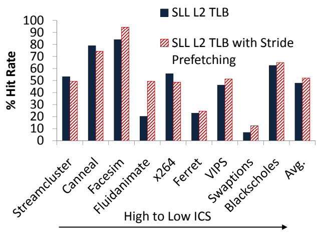

Fig. 34. Including simple stride prefetching improves SLL TLB hit rates by an average of 5% across the evaluated workloads. This is because it captures repetitive inter-core distance pairs.

Figure 30 shows that heavy replication exists across the benchmarks. As expected, high-ICS applications see heavy replication. 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.

8.2. Shared Last-Level TLBs versus Inter-Core Cooperative Prefetching

We now consider the benefits versus ICC prefetching (which includes both Leader-Follower and Distance-based Cross-Core prefetching). Both strategies aim to catch requests that have missed in the L1 TLB, albeit in different ways and with different latencies. In this section we present hit rates; a performance analysis is presented in Sections 8.6 and 9.3.

Figure 31 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, though the exact benefits vary across benchmarks.

On average, SLL TLBs see merely a 4% drop in hit rate compared to ICC prefetchers. Moreover, Figure 31 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 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.

8.3. Shared Last-Level TLB Sharing Characteristics

Having quantified the benefits of SLL TLBs, it is also useful to understand their sharing patterns. Figure 32 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 are sharers. On our 4-core CMP, there can be 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 33 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 prioritize 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.

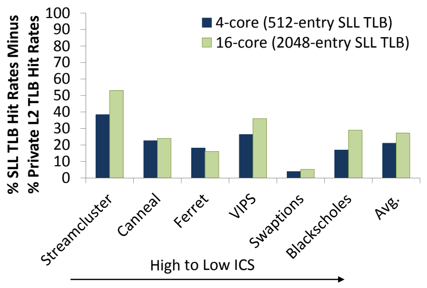

Fig. 35. Increase in hit rate that SLL TLBs provide versus private, per-core L2 TLBs for 4-core and 16-core CMPs. Since private L2 TLBs are 128-entry, the SLL TLB is 512-entry and 2048-entry for 4-core and 16-core CMPs respectively. Note the *increased* hit rates at higher core counts.

8.4. Shared Last-Level TLBs with Simple Stride Prefetching

Having studied the hit rates of the baseline SLL TLB, we now consider low-complexity enhancements. In particular, we augment SLL TLBs by adding simple stride prefetching for translations residing on the same cache line as the currently missing entry. While this cannot capture the sophistication of ICC prefetching techniques, it does offer some of its benefits while retaining implementation simplicity. As covered in Section 5.3, prefetched candidates are 1, 2, and 3 pages away from the currently missing page.

Figure 34 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. For example, `Blackscholes`, which has repetitive 4-page strides [Bhattacharjee and Martonosi 2009], 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 [Bhattacharjee and Martonosi 2009]. Similarly, even `Facesim` sees an additional 10% hit rate since it exploits 2 and 3 page strides.

Figure 34 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.

8.5. 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 35 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 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 35 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 [Bhattacharjee and Martonosi 2009], 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.

8.6. 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. Since ICC prefetchers have already been established as overly-complex for implementation, we compare SLL TLB performance against the commercial norm of private L2 TLBs. As previously discussed, full-run cycle-level simulations would take weeks per datapoint to complete and are simply never done for TLB studies. Instead we use a CPI analysis inspired by [Saulsbury et al. 2000].

Table VIII. 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.

Type	**Type 1**	**Type 2**	**Type 3**	**Type 4**
Description	Flush ROB Setup insts. TSB Hit in L1$ Cleanup code	Flush ROB Setup insts. TSB Hit in L2$ 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

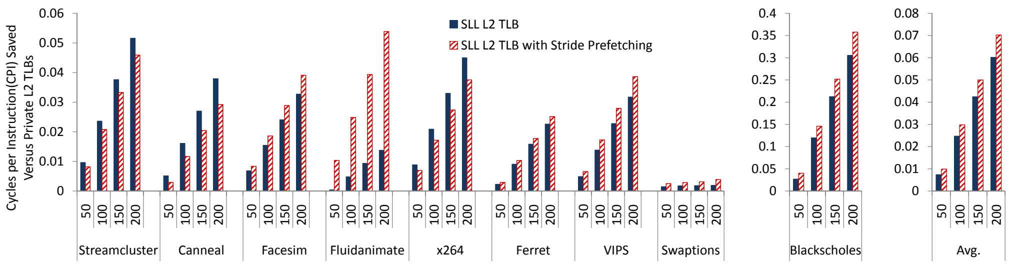

Fig. 36. CPI saved by SLL TLBs against private L2 TLBs. Every application benefits from SLL TLBs with exact gains increasing with miss penalties.

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 VIII 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 data structure, may be cached. A TSB hit in the L1 cache minimizes the total handler penalty to roughly 50 cycles (Type 1), while an L1 miss results 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.

Figure 36 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 unrealistic assumption that all handlers are L1-TSB hits 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 investigate SLL TLBs for multiprogrammed sequential workloads.

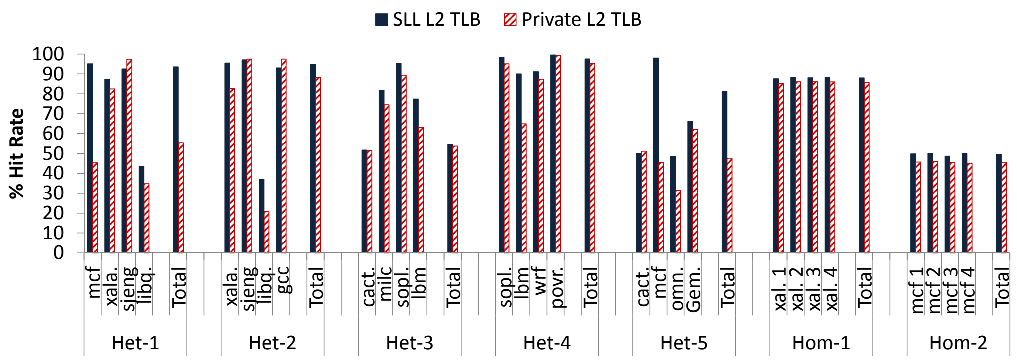

Fig. 37. Hit rates for the multiprogrammed workloads for both the SLL L2 TLB and the private L2 TLBs. SLL TLB hit rates in total for each heterogeneous workload combination are substantially higher than private for L2 TLBs (on average, by 21%). Furthermore, high-stress applications like `mcf` see vast improvements without noticeably degrading lower-stress applications. Even homogeneous workload combinations see hit rate increases with SLL TLBs.

9. SHARED LAST-LEVEL TLBS: RESULTS FOR MULTIPROGRAMMED WORKLOADS

We now focus on SLL TLBs for workloads comprised of sequential applications, running one per core in a multiprogrammed fashion. First, Section 9.1 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, the focus is on understanding how effectively a single shared last-level TLB adapts to simultaneously executing applications with different memory requirements. In contrast, for the homogeneous workloads, we study SLL TLB benefits when multiple programs of similar nature execute.

After studying application hit rates for programs with processes pinned to cores, Section 9.2 analyzes the effect of process migration among cores. Then, Section 9.3 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.

9.1. Multiprogrammed Workloads with One Application Pinned per Core

Figure 37 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) previously 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 vast 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 37 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).

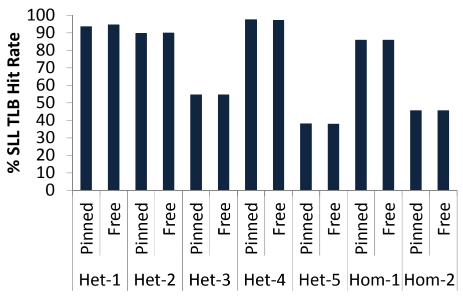

Fig. 38. L2 D-TLB Hit Rate for the multiprogrammed workloads. This shows that very little change is seen between the case of processes being pinned to specific cores and the case in which the OS is left to manage the migrations.

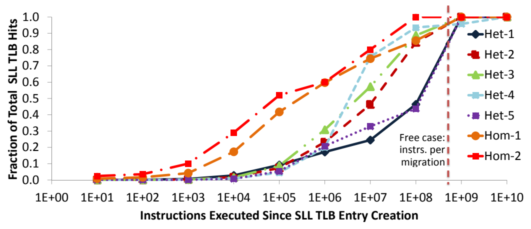

Fig. 39. SLL *TLB Creation to Hit* (CTH) counts, recorded as the number of instructions executed since the creation of every hit entry. These results show that TLB entries generally have very short CTH counts relative to the overall execution of the program.

Third, Figure 37 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, dividing the entries equally 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.

9.2. Multiprogrammed Workloads with Process Migration

Our multiprogramming studies up to this point have pinned one application to each core of our evaluated 4-core CMP. Therefore, the benefits extracted for these multiprogrammed workloads have been due to the SLL TLB's ability to intelligently allocate its resources to multiple simultaneous applications with differing memory demands. However, contemporary systems typically run operating systems which often employ process migration in which applications can often switch cores through their execution time. Furthermore, process migration is likely to become even more prevalent in future CMPs as a mechanism to cope with issues like dynamic thermal management techniques [Choi et al. 2007; Donald and Martonosi 2006]. In fact, recent work suggests that future CMPs are likely to provide support for fast process migration [Rangan et al. 2009]. It is therefore important to consider the effects of process migrations on SLL TLBs.

To test migration in our workloads, we show SLL L2 TLB hit rates for two scenarios. First, we considered the *pinned* case for every workload combination, where one application is pinned to each core. This corresponds to the results already presented in Section 9.1. Second, we considered the *free* case, where the applications are left unpinned and the Solaris scheduler is free to migrate processes. We expect that process migration would actually introduce inter-core sharing for SLL TLBs to exploit. Specifically, when a process migrates, it sees new L1 TLB misses and if the L2 TLBs are private, suffers from additional page table walks. The SLL TLB, however, mitigates this problem by giving the process on its new core L2 access to its previous translations, reducing TLB misses. Figure 38 details the SLL TLB hits rates for our pinned and free experiments. The numbers shown are the hit rates for the total L2 accesses across all the sequential applications constituting each workload. It is clear from the hit rates that the free case introduces little additional inter-core sharing for SLL TLBs to exploit over the pinned case. Therefore, hit rates increase only marginally for the free case. The reason for this surprisingly small increase is that operating systems traditionally attempt to minimize migrations to avoid cold caches, TLBs, and migration code overheads. In fact, we find that in the free case, each sequential application migrates no more than twenty times over

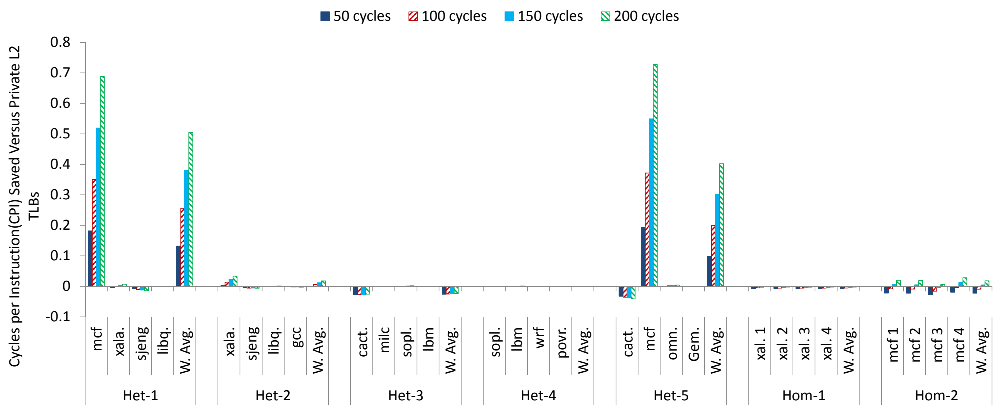

Fig. 40. CPI saved using SLL TLBs versus private L2 TLBs for individual applications and per-workload averages. Higher TLB miss penalties result in greater performance gains.

a ten billion instruction window. Therefore, any benefits SLL TLBs extract from inter-core sharing are amortized over a very long period, reducing the observed benefits

Nevertheless, as future CMPs may provide support for fast process migration [Choi et al. 2007; Rangan et al. 2009], it is useful to consider the benefits that SLL TLBs would provide for these platforms as migration frequencies increase. As a guideline to understand SLL TLB benefits at various migration speeds, we now present *Creation to Hit (CTH)* instruction counts for our workloads. CTH counts are gathered by recording, for each SLL TLB hit, the number of instructions ago that the corresponding translation entry was brought into the SLL TLB. The larger the CTH counts, the higher the chances that SLL TLBs will retain these entries across migrations.

Figure 39 shows the cumulative distribution function plotting the probability of SLL TLB entries of a particular CTH count being accessed in each of the workloads. The x-axis shows entry CTH values (measured in instruction counts) varying on a log scale. The y-axis shows the probability that SLL TLB entries of that CTH count (or less) result in a hit. From this data, it becomes clear that very few TLB entries have CTH counts long enough to be exploited with high payoff for the migration rates in the free case. In fact, Hom-2 sees that all its hits are to entries created at most a hundred million instructions ago, far too short a lifetime for migrations that occur over billion-instruction ranges. Nevertheless, Figure 39 does show the benefits that more rapid migrations on future CMPs may glean from SLL TLBs. As such inter-core sharing will increase greatly and SLL TLBs should be a significant help in these cases.

9.3. Performance Analysis

As previously described, SLL TLBs are, by construction, aimed at capturing inter-core shared misses and hence, aiding parallel programs. To make SLL TLBs a viable option however, they must also not substantially degrade sequential applications. 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 8.6 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 [Saulsbury et al. 2000].

Figure 40 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 40 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 40 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 experience unchanging hit rates even as TLB reach is increased [Korn and Chang 2007; Woo et al. 2010]. Therefore, our larger SLL TLB only marginally increases its hit rate (see Figure 37) 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 [Korn and Chang 2007; Woo et al. 2010]; 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 all individual benchmarks in these workloads equally stress the SLL TLB, none sees a significant increase in TLB entries available to it. Therefore, performance is marginally decrease due to the additional SLL TLB access time, even though 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.

10. CONCLUSION

This paper shows the benefits of ICC prefetchers and SLL TLBs for both parallel and multiprogrammed sequential workloads. We find that ICC prefetching, which combine the benefits of both leader-follower prefetching and distance-based cross-core prefetching, eliminates an average of 46% of D-TLB misses across a wide range of parallel programs. Meanwhile, SLL TLBs exploit parallel program inter-core sharing to eliminate 7% to 79% of L1 TLBs misses, providing comparable benefits to ICC prefetchers, but use simpler hardware that is possible to implement on commercial systems today. They also outperform conventional per-core, private L2 TLBs by an average of 27%, leading to runtime improvements of as high as 0.25 CPI. Finally, a combined approach of integrating stride prefetching into SLL TLBs provides further increases in 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 by designers of future CMP systems to augment existing TLB hardware and thereby improve overall performance. This study points to a range of potential designs that include different combinations of SLL TLBs with prefetchers. Our results provide guidance to both sequential and parallel software developers on the benefits they can expect from this approach, using only readily-implementable and low-complexity hardware.

REFERENCES

BHATTACHARJEE, A. AND MARTONOSI, M. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *PACT*.

BIENIA, C. ET AL. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *PACT*.

BIENIA, C. AND LI, K. 2010. Fidelity and Scaling of the PARSEC Benchmark Inputs. *ISWC*.

CHEN, J. B., BORG, A., AND JOUPPI, N. 1992. A Simulation Based Study of TLB Performance. *ISCA*.

CHEN, T. AND BAER, J. 1995. Effective Hardware-based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*.

CHOI, J. ET AL. 2007. Thermal aware task scheduling at the system software level. *ISLPED*.

CLARK, D. AND EMER, J. 1985. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Trans. on Comp. Sys. 3,* 1.

DAHLGREN, F., DUBOIS, M., AND STENSTRÖM, P. 1993. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. *International Conference on Parallel Processing*.

DONALD, J. AND MARTONOSI, M. 2006. Techniques for Multicore Thermal Management: Classification and New Exploration. *ISCA*.

DRONGOWSKI, P. 2008. Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors. http://developer.amd.com/Assets/Basic_Performance_Measurements.pdf.

EBRAHIMI, E. ET AL. 2010. Fairness via Source Throttling: a Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. *ISCA*.

HINTON, G. 2001. The Microarchitecture of the Pentium 4. *Intel Technology Journal*.

HUCK, H. AND HAYS, H. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. *ISCA*.

INTEL. 2012. Intel 64 and IA-32 Architectures Software Developer's Manual. http://download.intel.com/products/processor/manual/325462.pdf.

JACOB, B. AND MUDGE, T. 1998a. A Look at Several Memory Management Units: TLB-Refill, and Page Table Organizations. *ASPLOS*.

JACOB, B. AND MUDGE, T. 1998b. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching Using Markov Predictors. *International Symposium on Computer Architecture*.

KANDIRAJU, G. AND SIVASUBRAMANIAM, A. 2002a. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. *Sigmetrics*.

KANDIRAJU, G. AND SIVASUBRAMANIAM, A. 2002b. Going the Distance for TLB Prefetching: An Application-Driven Study. *ISCA*.

KIM, C., BURGER, D., AND KECKLER, S. 2003. NUCA: A Non-Uniform Cache Architecture for Wire-Delay Dominated On-Chip Caches. *IEEE Micro Top Picks*.

KORN, W. AND CHANG, M. 2007. SPEC CPU2006 Sensitivity to Memory Page Sizes. *ACM SIGARCH Comp. Arch. News 35,* 1.

MARTIN, M. ET AL. 2005. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Comp. Arch. News*.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2009. CACTI 6.0: A Tool to Model Large Caches. *HP Labs Technical Report HPL-2009-85*.

NAGLE, D. ET AL. 1993. Design Tradeoffs for Software Managed TLBs. *ISCA*.

PHANSALKAR, A. ET AL. 2007. Subsetting the SPEC CPU2006 Benchmark Suite. *ACM SIGARCH Comp. Arch. News 35,* 1.

QUI, X. AND DUBOIS, M. 1998. Options for Dynamic Address Translations in COMAs. *ISCA*.

RANGAN, K., WEI, G., AND BROOKS, D. 2009. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. *ISCA*.

ROMANESCU, B., LEBECK, A., SORIN, D., AND BRACY, A. 2010. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. *HPCA*.

ROSENBLUM, M. ET AL. 1995. The Impact of Architectural Trends on Operating System Performance. *Trans. on Mod. and Comp. Sim.*.

SAULSBURY, A., DAHLGREN, F., AND STENSTRÖM, P. 2000. Recency-Based TLB Preloading. *ISCA*.

SHARIF, A. AND LEE, H.-H. 2009. Data Prefetching Mechanism by Exploiting Global and Local Access Patterns. *Journal of Instruction-Level Parallelism Data Prefetching Championship*.

SPEC. 2006. The Standard Performance Evaluation Corporation. SPEC CPU2006 Results. http://www.spec.org/cpu2006.

SRIKANTAIAH, S. AND KANDEMIR, M. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. *MICRO 43*.

SRINIVASAN, V., DAVIDSON, E., AND TYSON, G. 2004. A Prefetch Taxonomy. *IEEE Transaction on Computers 53,* 2.

SUN. 2003. An Overview of UltraSPARC III Cu. http://www.sun.com/processors/UltraSPARC-III/USIIICuoverview.pdf.

TALLURI, M. AND HILL, M. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. *ASPLOS*.

TICKOO, O. ET AL. 2007. qTLB: Looking Inside the Look-aside Buffer. *HiPC*.

VENKATASUBRAMANIAN, G. ET AL. 2009. TMT - a TLB Tag Management Framework for Virtualized Platforms. *SBAC-PAD 21*.

VILLAVIEJA, C. ET AL. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. *PACT*.

VIRTUTECH. 2007. Simics for Multicore Software.

WILTON, S. AND JOUPPI, N. 1994. An Enhanced Access and Cycle Time Model for On-Chip Caches. *West. Res. Lab. Res. Report 93/5*.

WOO, D. H. ET AL. 2010. An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth. *HPCA*.