Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors

Abhishek Bhattacharjee and Margaret Martonosi

Department of Electrical Engineering
Princeton University
{abhattac, mrm}@princeton.edu

Abstract

Translation Lookaside Buffers (TLBs) are commonly employed in modern processor designs and have considerable impact on overall system performance. A number of past works have studied TLB designs to lower access times and miss rates, specifically for uniprocessors. With the growing dominance of chip multiprocessors (CMPs), it is necessary to examine TLB performance in the context of parallel workloads.

This work is the first to present TLB prefetchers that exploit commonality in TLB miss patterns across cores in CMPs. We propose and evaluate two Inter-Core Cooperative (ICC) TLB prefetching mechanisms, assessing their effectiveness at eliminating TLB misses both individually and together. Our results show these approaches require at most modest hardware and can collectively eliminate 19% to 90% of data TLB (D-TLB) misses across the surveyed parallel workloads.

We also compare performance improvements across a range of hardware and software implementation possibilities. We find that while a fully-hardware implementation results in average performance improvements of 8-46% for a range of TLB sizes, a hardware/software approach yields improvements of 4-32%. Overall, our work shows that TLB prefetchers exploiting inter-core correlations can effectively eliminate TLB misses.

Categories and Subject Descriptors B.3 [*Memory Structures*]: Design Styles; C.1 [*Processor Architectures*]: Parallel Architectures; C.4 [*Performance of Systems*]: Design Studies; D.4 [*Operating Systems*]: Performance

General Terms Design, Experimentation, Performance

Keywords Translation Lookaside Buffer, Parallelism, Prefetching

1. Introduction

To avoid high-latency accesses to operating system (OS) page tables storing virtual-to-physical page translations, processor Memory Management Units (MMUs) store commonly used translations in instruction and data Translation Lookaside Buffers. While past work has addressed various options for TLB placement and lookup [4, 17], most contemporary systems place them in parallel with the first-level cache. Due to their long miss penalties, TLB behavior affects processor performance significantly [6, 13, 16, 18].

Numerous techniques have been proposed to improve TLB performance. On the hardware side, TLB characteristics such as size, associativity, and the use of multilevel hierarchies have been explored [4]. On the software side, the concept of superpaging has been examined [23]. Hardware/software prefetching techniques have also been investigated in detail [14, 19]. While effective, proposed prefetchers are specific to uniprocessors. With the growing dominance of chip multiprocessors (CMPs), it is imperative that we examine TLB performance in the context of parallel workloads.

Recent characterizations of emerging parallel workloads on CMPs show that significant similarities exist in TLB miss patterns among multiple cores [2]. This occurs either in the form of TLB misses caused by identical virtual pages on multiple cores, or in the form of predictable strides between virtual pages causing TLB misses on different cores. These observations point to valuable opportunities for eliminating TLB misses by studying common miss streams across cores.

This paper develops *Inter-Core Cooperative* (ICC) TLB prefetchers to exploit common TLB miss patterns among cores for performance benefits. We propose and evaluate two approaches for TLB prefetching. The first approach, *Leader-Follower* prefetching, exploits common TLB miss virtual pages among cores by pushing TLB mappings from *leader* to other cores, reducing TLB misses. In the second approach, we augment the uniprocessor-centric *Distance-based* prefetching mechanism developed by Kandiraju and Sivasubramaniam [14] to exploit stride-predictable TLB misses across CMP cores. Our specific contributions are as follows:

- Foremost, our work is the first to recognize opportunities for inter-core TLB cooperation and propose mechanisms in response.
- In particular, by pushing TLB mapping information from the initial miss core (leader) to the other cores, Leader-Follower prefetching can eliminate up to 57% of the TLB misses across the surveyed workloads. Confidence mechanisms also help to reduce over-aggressive prefetching.
- Furthermore, we show how Distance-based Cross-Core prefetching captures repetitive TLB miss virtual page stride patterns between cores and within the same core to eliminate up to 89% of the TLB misses across the evaluated workloads.
- We then combine both approaches and show that they can be implemented with modest hardware to eliminate 13-89% of TLB misses across the tested parallel benchmarks.
- Finally, we investigate performance improvements for a range of hardware and software implementations of ICC prefetching. While a fully hardware implementation can yield average performance improvements of 8-46%, even after moving significant components of the prefetcher into software we achieve average improvements of 4-32%.

Overall this work is the first to exploit inter-core TLB miss redundancy in parallel applications. The rest of the paper is structured as follows. Section 2 covers background material. Section 3 then proposes two ICC TLB prefetchers. Section 4 presents our evaluation methodology while Section 5 evaluates the benefits of each TLB prefetching scheme individually and then combines them. Section 6 addresses the performance benefits of incorporating the ICC TLB prefetchers for a range of hardware/software implementations. Section 7 discusses system issues related to prefetching and finally, Section 8 offers conclusions.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03...$10.00

2. Background and Related Work

Since TLBs are usually placed in parallel with first-level caches, CMPs maintain per-core instruction and data TLBs, which are largely oblivious to the behavior of other TLBs, except for shootdowns used for coherence. These TLBs are either *hardware-managed* or *software-managed*. Hardware-managed TLBs use a hardware state machine to walk the page table, locate the appropriate mapping, and insert it into the TLB on every miss. Because the page-table walk is initiated by a hardware structure, there is no need for expensive interrupts and the pipeline remains largely unaffected. Moreover, the handling state machine does not pollute the instruction cache. Past studies have shown the performance benefits of hardware-managed TLBs [10], with typical miss latencies ranging from 10 to 50 cycles [11, 13].

Although hardware-managed TLBs do offer performance benefits, they also imply a fixed page table organization. As such, the OS cannot employ alternate designs. In response, RISC architectures such as MIPS and SPARC often use software-managed TLBs [9, 16]. Here, a TLB miss causes an interrupt, and the OS executes a miss handler which walks the page table and refills the TLB. Since the OS controls the page table walk, the data structure design is flexible. This flexibility, however, comes with an associated performance cost. First, precise interrupts prompt pipeline flushes, removing a possibly large number of instructions from the reorder buffer. Second, the miss handler tends to be 10 to 100 instructions long and may itself miss in the instruction cache [10]. In addition, the data cache may also be polluted by the page table walk. All these factors contribute to TLB miss latencies that can span hundreds of cycles [9, 10].

Numerous studies in the 1990s investigated the performance overheads of TLB management in uniprocessors. Studies placed TLB handling at 5-10% of system runtime [6, 13, 16, 18] with extreme cases at 40% of runtime [8]. Anderson showed that software-managed TLB miss handlers are among the most commonly executed primitives [1] while Rosenblum et al. found that these handlers can use 80% of the kernel's computation time [18].

To tackle TLB management overheads, early work addressed hardware characteristics such as TLB size, associativity, and multi-level hierarchies [4]. More recently, TLB prefetching schemes have also been explored. For example, Saulsbury et al. [19] introduce *Recency-based* prefetching to exploit 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 [14] adapt cache prefetching techniques such as *Sequential*, *Arbitrary-Stride* and *Markov* prefetching [5, 7, 12]. 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.

While these prefetchers exhibit performance benefits, they all target uniprocessors. As CMPs become ubiquitous, it becomes necessary to re-evaluate the role of TLBs in the performance of emerging parallel workloads. There has been surprisingly little work done in this context although our prior work indicates that emerging parallel workloads can severely stress current TLB designs, with a worst-case CPI of 0.7 devoted to D-TLB management on a 4-core AMD Opteron [2]. Fortunately, this work also indicates that significant commonality exists in TLB miss patterns across cores of a CMP. In particular, a large number of TLB misses are predictable in that they are caused by virtual page accesses seen on multiple cores or by virtual pages that experience repetitive inter-core strides. As such, these observations present a valuable opportunity to eliminate the rising costs of TLB misses in parallel workloads. This work uses this insight to develop novel CMP-targeted TLB prefetchers for performance improvements of parallel applications.

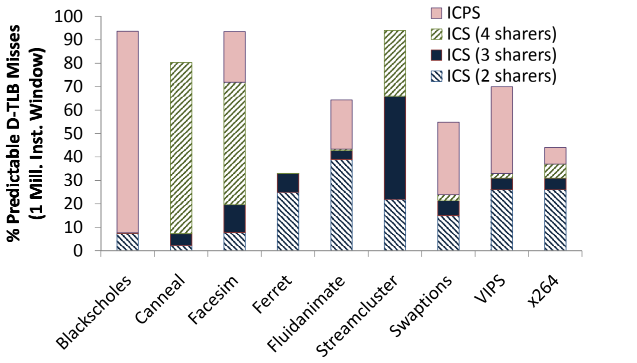


Figure 1. Number of inter-core shared (ICS) D-TLB misses, per number of sharers, and inter-core predictable stride (ICPS) D-TLB misses. Summing these categories and normalizing to the total misses represents the potential for ICC prefetching to help.

While the techniques we develop in this work may be applied to both I-TLBs and D-TLBs, this study focuses on D-TLBs because of their far greater impact on system performance [2, 6, 19]. Our approaches, however, are likely to reduce I-TLB misses as well.

3. Two Inter-Core Cooperative TLB Prefetchers

3.1 Motivation and Background Data

To develop effective prefetching mechanisms exploiting redundant inter-core TLB miss patterns, predictable TLB miss types must be understood. In [2], we classified predictable TLB misses in CMPs into two categories:

1. *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*.

2. *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.

Figure 1 summarizes the prevalence of these types of predictable D-TLB misses across the parallel benchmarks from PARSEC surveyed in [2], 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. 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). Note that the methodology and benchmarks used for this plot are described in detail in Section 4.

In this work, we exploit these predictable misses with ICC prefetching techniques that detect inter-core TLB behavior commonality and eliminate TLB misses. Our strategy is to 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.

3.2 Prefetching Challenges

Despite the potential benefits of inter-core cooperative prefetching, key challenges remain. First, it is difficult to create a single

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

Table 1. 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.

prefetching scheme that can adapt to diverse D-TLB miss patterns. For example, while PARSEC benchmarks `Canneal` and `Streamcluster` see many shared ICS misses, `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 1 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, for our prefetchers to react and prefetch the desired entry before use. At the same time, 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 to the subsequent predictable TLB miss on another core. For a 4-core CMP with 64-entry TLBs, this time is between 16K to 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.

3.3 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.

3.3.1 Concept

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.

3.3.2 Algorithm

Figure 2 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

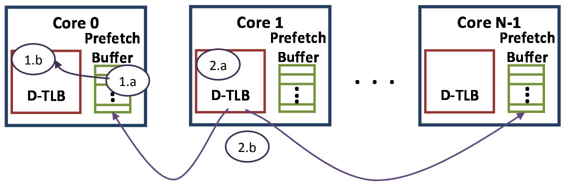


Figure 2. The baseline Leader-Follower algorithm prefetches a TLB miss translation seen on one core (the leader) into the other cores (the followers) to eliminate inter-core shared TLB misses.

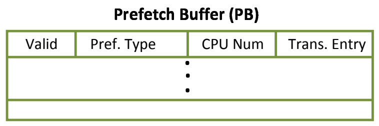


Figure 3. Each prefetch buffer entry has a Valid bit, a Prefetch Type bit (to indicate whether the entry arose from Leader-Follower or Distance-based Cross-Core prefetching), CPU Number (indicating prefetch-initiating core number), and the translation information.

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 2 separates the Leader-Follower algorithm into two example cases. While these cases are numbered, there is no implied ordering between them. 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, the translation is located and refilled into the D-TLB. In step 2b, this translation is also prefetched or *pushed* into PBs of the other cores, with the aim of eliminating future ICS misses on the other cores.

3.3.3 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 [20], 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 [20]. 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 alleviate this problem by incorporating confidence estimation. This results in modifications to both the prefetch buffer and the baseline Leader-Follower algorithm.

Figure 3 shows that each PB entry now holds a *CPU Number* field in addition to the baseline information. The CPU Number tracks the leader core responsible for the prefetch of each entry. Figure 4 illustrates the modification to baseline Leader-Follower prefetching. 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.

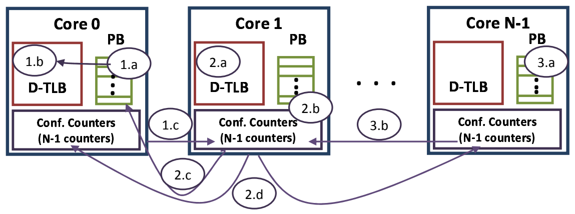


Figure 4. Algorithm for incorporating confidence estimation with saturating confidence counters in Leader-Follower prefetching scheme.

Figure 4 also details three types of operations 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.

Section 5.4 presents results showing that confidence estimation gives dramatic performance improvement for modest hardware.

3.3.4 Key Attributes

We now highlight key properties of Leader-Follower prefetching. First, this 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.

Finally, while later sections in the paper will quantitatively evaluate the benefits of this approach and investigate feasible PB sizes, it is clear that the Leader-Follower technique is advantageous primarily for inter-core shared TLB misses. For instances of inter-core predictable strides, the next section investigates Distance-based Cross-Core prefetching.

3.4 Distance-Based Cross-Core Prefetching

3.4.1 Concept

As detailed in Section 3.2, although many TLB misses are ICPS, creating feasible hardware to detect and adapt to the various stride

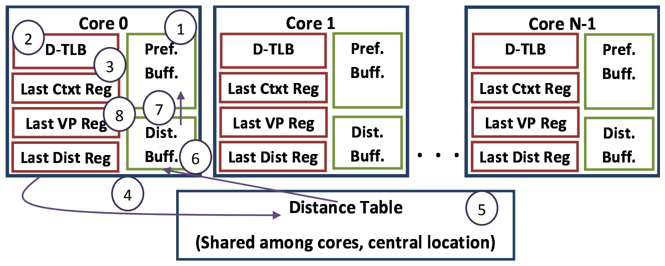


Figure 5. Distance-based Cross-Core prefetching uses a central, shared Distance Table to store distance pairs and initiates prefetches based on these patterns whenever a TLB miss occurs on one of the cores (for both PB hits and misses). Note that the prefetches on a core may be initiated by a distance-pair initially seen on a different core.

patterns is challenging. Therefore, our solution draws from a distance-based approach introduced for uniprocessors [14].

To understand Cross-Core Distance prefetching, assume that two cores in a CMP 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 but makes distance patterns available to *other* cores. For example, the first difference 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 - last missing virtual page) is used to scan the observed distance pairs to 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 TLB 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).

3.4.2 Algorithm

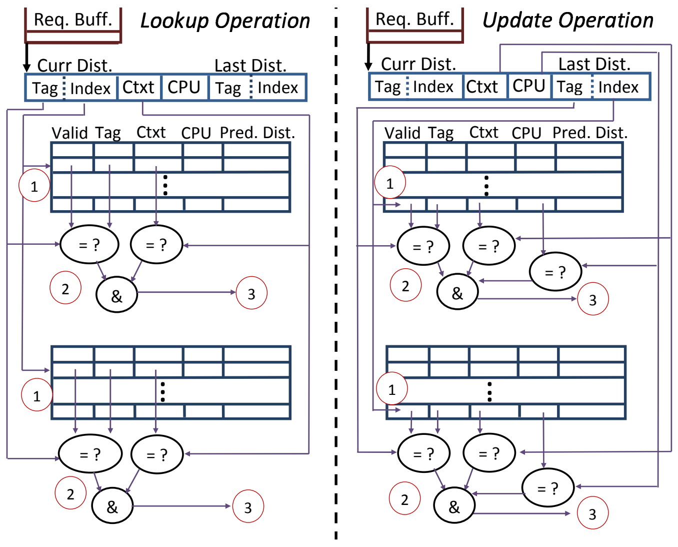
Figure 5 shows how Distance-based Cross-Core prefetching works. We again assume an N-core system with prefetches placed into per-core PBs. The steps of the approach are as follows:

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.



Req. Buff.
Lookup Operation
Curr Dist.
Last Dist.
Tag Index Ctxt CPU Tag Index
Valid Tag Ctxt CPU Pred. Dist.
= ?
&
1
2
3
Req. Buff.
Update Operation
Curr Dist.
Last Dist.
Tag Index Ctxt CPU Tag Index
Valid Tag Ctxt CPU Pred. Dist.

Figure 6. 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.

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 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. Regs 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 6.

3.4.3 Distance Table Details

Figure 6 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

Step 1: The lower-order bits of the *current distance* index into the appropriate set. Figure 6 shows a 2-way set associative DT, but the associativity could be higher.

Step 2: 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.

Step 3: 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

Step 1: In contrast to the lookup, DT update uses the lower-order bits of the *last distance* to index into the required set.

Step 2: 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. We move to step 3 if these comparisons hold; otherwise, we skip to step 4.

Step 3: We now 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 however, no duplicates exist, we update the Pred. Dist. entry with the current distance.

Step 4: 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.

3.4.4 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*; 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.

Finally, while Distance-based Cross-Core prefetching reduces ICPS TLB misses, it can also help with ICS misses and distance-pairs seen on only one core. Hence, some benefits of uniprocessor TLB prefetching are also provided with this approach.

4. Methodology and Characterization

4.1 Simulation Infrastructure

We evaluate our inter-core cooperative TLB prefetchers using the Multifacet GEMS simulation infrastructure with parameters listed in Table 2 [15]. GEMS employs Virtutech Simics [24] as its functional model, which simulates a 4-16 core CMP based on Sun's UltraSPARC III Cu with SunFire's MMU architecture [21]. We instrument the Simics MMU source code to track requested virtual/physical address pairs prompting TLB misses and integrate our ICC prefetchers.

Table 3 shows the modeled MMUs. 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 organizations to date. The SF3800 employs a 16-entry fully-associative L1 D-TLB used primarily by the OS for locking pages. The SF3800 also has two L1 512-entry D-TLBs for unlocked translations. These are accessed in parallel and can be configured by the OS to hold translations for different page sizes. In our simulations, the OS sets both TLBs to the same page size, making them equiva-

Architecture	SPARC (out-of-order)
Core Count	4-16
Fetch/Issue/Commit Width	4
Reorder Buffer Size	64-entry
Instruction Window Size	32-entry
L1 cache	Private, 32 KB (4-way)
L2 cache	Shared, 16 MB (4-way)
L2 roundtrip	40 cycles (uncontested)
OS	Sun Solaris 10
Interconnection Network	Mesh

Table 2. Simulation parameters used to evaluate TLB prefetchers.

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

Table 3. Simulated SunFire MMUs with software-managed TLBs.

	Parallelization		Data Usage	
Benchmark	**Model**	**Granul.**	**Sharing**	**Exchange**
`Blackscholes`	Data-parallel	Coarse	Low	Low
`Canneal`	Unstructured	Fine	High	High
`Facesim`	Data-parallel	Coarse	Low	Medium
`Ferret`	Pipeline	Medium	High	High
`Fluidanimate`	Data-parallel	Fine	Low	Medium
`Streamcluster`	Data-parallel	Medium	Low	Medium
`Swaptions`	Data-parallel	Coarse	Low	Low
`VIPS`	Data-parallel	Coarse	Low	Medium
`x264`	Pipeline	Coarse	High	High

Table 4. Summary of PARSEC benchmarks used to evaluate ICC TLB prefetchers. Note the diversity in parallel models, granularities, and data sharing characteristics.

lent to a single 1024-entry D-TLB. Finally, we evaluate Intermediate MMUs with TLB sizes between the SF280R and SF3800.

Our simulator runs Solaris 10, which can exploit superpaging techniques [23]. However, when tracking MMU activity, we find no use of superpaging and cannot access the necessary source code to initiate this ourselves. Nevertheless, our ICC schemes are equally applicable to scenarios with superpaging.

Finally, due to the slow speeds of full-system simulation, we present results observed with 1 billion instructions rather than full runs. Our instruction windows are chosen such that under 5% of the total D-TLB misses are cold misses across the workloads.

4.2 Benchmarks and Input Sets

We evaluate our prefetchers using PARSEC benchmarks, a suite of next-generation shared-memory programs for CMPs [3]. Table 4 lists the PARSEC workloads we use here. Of the 13 PARSEC workloads available, we are able to compile the 9 listed for our simulator. The workloads come from many application domains and, as shown, use diverse parallelization schemes (unstructured, data, and pipeline-parallel), parallelization granularities, and inter-core communication characteristics.

We run the PARSEC workloads with a number of threads equivalent to the core count of the CMP system, and we use *Simlarge* input data sets. Since TLB misses occur with coarser temporal granularity than cache misses, we must use large input data sets to realistically stress TLB designs. *Simlarge* represents the largest PARSEC input set considered feasible for simulation.

4.3 D-TLB Miss Rates of PARSEC Workloads

Figure 7 plots D-TLB misses per million instructions (MMI) for the workloads across the SF280R, Intermediate, and SF3800 MMUs. As expected, the D-TLB misses decrease with larger TLBs. However, benchmarks like `Canneal`, `Ferret`, and `Streamcluster` consistently suffer from high D-TLB misses, even with larger

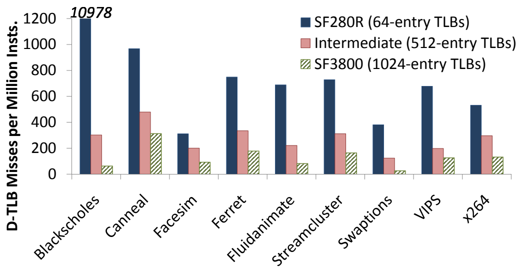


Figure 7. D-TLB misses per million instructions (MMI) for the PARSEC workloads. Note that `Canneal`, `Ferret`, and `Streamcluster` consistently experience the most D-TLB misses. While `Blackscholes` sees the most misses for SF280R MMUs, its performance improves relative to other workloads for larger TLBs.

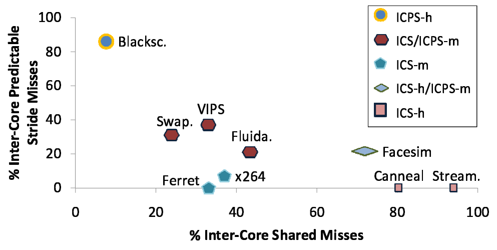


Figure 8. Based on inter-core sharing, we separate the workloads into ICPS-h, ICS/ICPS-m, ICS-m, ICS-h/ICPS-m, and ICS-h categories.

TLBs. Moreover, while `Blackscholes` sees a particularly high MMI for SF280R MMUs, this declines sharply for larger TLBs.

The actual performance implications of D-TLB behavior depend on a number of factors apart from D-TLB MMIs. For example, the time taken for page table walks and the CPI of the benchmark heavily influence how severely the D-TLB MMIs affect performance and the benefits of our ICC prefetchers. Section 6 will explore these issues in more detail.

4.4 Classification of Inter-Core D-TLB Patterns

Figure 8 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: This is for stride-reliant workloads with high ICPS misses and low ICS sharing. Only `Blackscholes` is in this category.

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

ICS-m: These have moderate ICS misses and few ICPS misses. `Ferret` and `x264` comprise this category.

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

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

We will use these classifications to assess the benefits of our prefetchers. Specifically, we expect that ICS-high categories particularly benefit from Leader-Follower prefetching while ICPS-high benchmarks exploit Distance-based Cross-Core prefetching.

4.5 Experimental Approach

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

In Section 5, 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

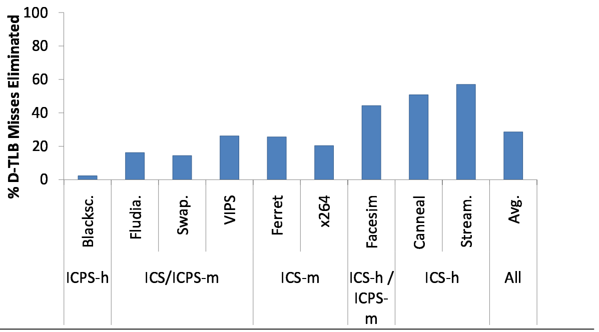


Figure 9. Percentage of D-TLB misses eliminated with Leader-Follower prefetching with infinite PBs. This scheme performs well for high-ICS benchmarks such as `Canneal`, `Facesim`, and `Streamcluster` but poorly for ICPS-reliant `Blackscholes`.

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).

Finally, in Section 6, we investigate hardware/software prefetcher implementation tradeoffs and assess the benefits and overheads of each approach. We then study the performance implications of these approaches for multiple core counts and TLB sizes.

5. Inter-Core Cooperative Prefetcher Results

We now focus on the benefits of the prefetchers and explore the hardware parameters involved. In Section 5.1, we quantify the benefits of Leader-Follower prefetching and then in Section 5.2, do the same for Distance-based Cross-Core prefetching. Both these cases assume an aggressive implementation with infinite PBs and no confidence estimation. In Section 5.3, we then combine both approaches for feasible PB sizes. Subsequently, Section 5.4 shows how confidence estimation reduces bad prefetches for better performance. Finally, Section 5.5 compares our approach against increasing TLB sizes.

5.1 Leader-Follower Prefetching

Figure 9 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.

5.2 Distance-Based Cross-Core Prefetching

Next, Figure 10 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.

Figure 10 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 10 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 through execution.

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 [14] 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 10, 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.

5.3 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 11 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 11 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 [2], see more than 44% of their misses eliminated, translating to significant performance savings.

Interestingly, Figure 11 shows that ICS-h benchmarks `Canneal` and `Streamcluster` suffer most from decreasing PB sizes. Section 5.4 shows how confidence estimation can mitigate this effect.

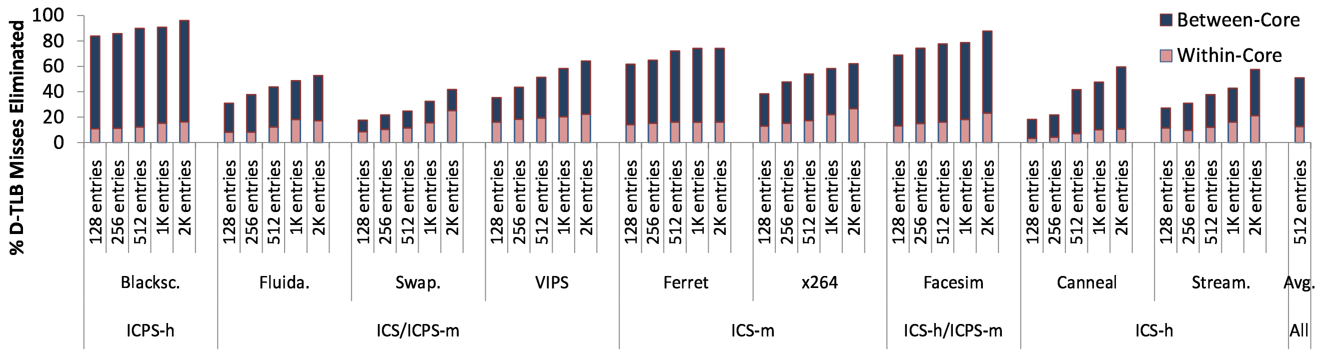


Figure 10. Percentage of D-TLB misses eliminated with Distance-based Cross-Core prefetching assuming infinite PBs for various sizes of the DT. Note that a high number of misses are eliminated consistently across benchmarks, primarily from *between-core* prefetches.

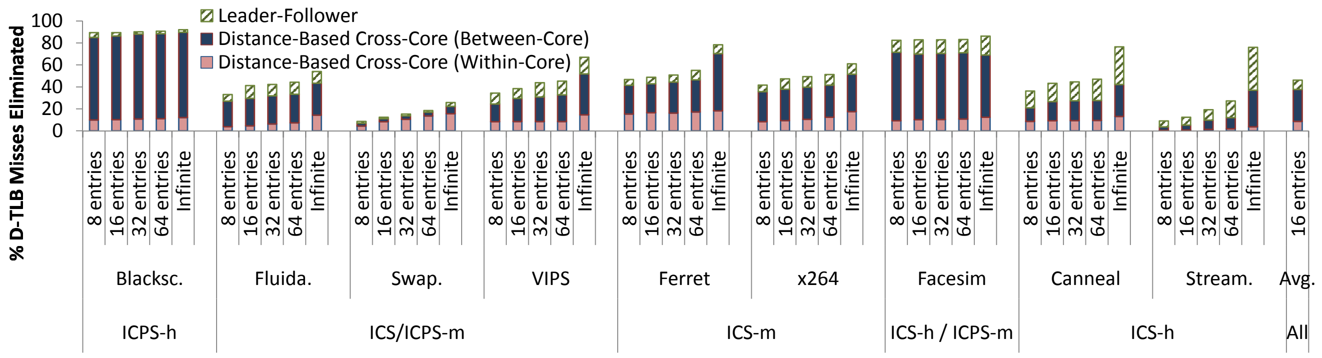


Figure 11. Effect of combining the two prefetching schemes with finite PBs. Even with as few as 16 entries in the PB, these techniques eliminate an average of 46% of the D-TLB misses.

Based on Figure 11, 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 [14].

5.4 Integrating Confidence Estimation

Our results so far assume the absence of confidence estimation described in Section 3.3.3. 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 12 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 12 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 13 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.

5.5 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 14 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.

Figure 14 shows that ICC prefetching notably outperforms blindly increasing TLB sizes across all sizes for all 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% due to the fact that 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.

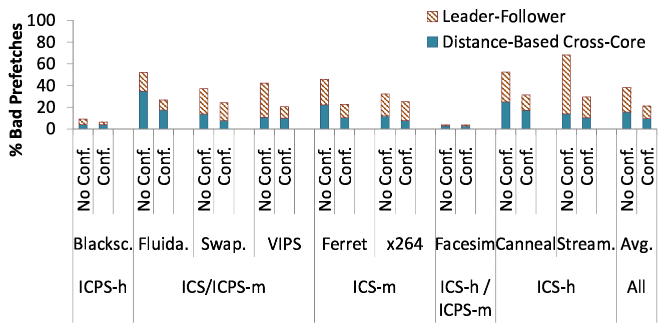


Figure 12. Percentage of total prefetches that are bad because they are never used or are prematurely evicted from the PB due to its finite size. Without confidence there are many bad prefetches, particularly from the Leader-Follower scheme. However, 2-bit confidence counters fix this, leading to a 2× decrease in bad prefetches.

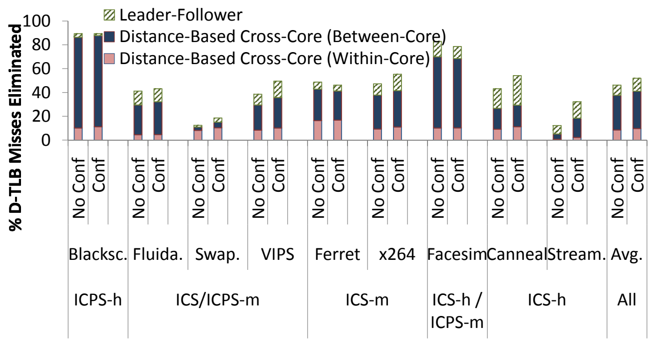


Figure 13. Percentage of D-TLB misses eliminated with the inclusion of confidence estimation. Not only does confidence estimation reduce bad prefetches, it also improves prefetcher performance by retaining useful information for longer in the PB. On average, 6% additional D-TLB misses are eliminated by incorporating confidence estimation.

6. Hardware/Software Implementation Tradeoffs

A number of hardware/software implementations are possible for ICC prefetching. This section discusses implementation possibilities and their impact on performance.

Table 5 presents the three implementation options that we study. First, in Section 6.1, we assess the performance implications of a fully-hardware design, in which both Leader-Follower and Distance-based Cross-Core prefetching are implemented completely in hardware. Moreover, this scheme assumes the presence of per-core state machines to walk the page table, like in hardware-managed TLBs. This is the highest-performance but most resource-hungry of the options (although the hardware remains modest).

In Section 6.2, we then explore a hardware implementation, but without per-core state machines to walk the page table. This represents the case of software-managed TLBs. In this case, we augment Distance-based Cross-Core prefetching to conduct DT-induced page table walks in bursts within OS interrupts. We will explain our burst prefetch algorithm in detail in Section 6.2.

Finally, Section 6.3 develops a hybrid hardware/software approach by moving the structures for Distance-based Cross-Core Prefetching, such as the DT and the Last Ctxt., VP, and Dist. registers into software. We also remove the DB entirely. We do, however, leave the PBs and Leader-Follower prefetching within hardware. This implementation strives for performance benefits close to the full-hardware case but with lower hardware requirements.

6.1 Fully-Hardware Implementation

We first present the performance of a fully-hardware ICC prefetcher with 16-entry PBs, a 512-entry, 4-way DT, 4-entry DBs and confidence estimation. Again, it takes 40 cycles for the leader to push a translation to the follower, and 40 cycles for each DT access.

A key issue affecting performance 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. As with hardware-managed TLBs, we assume that a fully-hardware prefetcher uses hardware state machines to walk the page table. This means that DT-induced translation searches proceed without OS or program intervention.

To assess performance benefits, we need realistic latencies for DT-induced page table walks. Since our simulator does not allow us to implement these walks directly, we instead run separate simulations for two scenarios: cases in which page table walks all hit in the L1 cache, and those in which page table walks all miss in the L1 but hit in the L2 cache. These are the typical cases since Solaris maintains a software data structure known as the Translation Storage Buffer (TSB) which stores frequently accessed page table entries and is typically found in the L1 or L2 cache.

Figure 15 shows the runtime performance improvements of the SF280R, Intermediate, and SF3800 MMUs for a 4-core CMP. For each workload, separate results are shown for DT-induced page table walks that hit in the L1 and L2 cache.

Significant performance benefits exist for all the workloads considered. They are particularly pronounced for SF280R MMUs (over 46% on average), and remain notable for Intermediate MMUs (over 14% on average), and SF3800 MMUs (over 8% on average).

Figure 15 also shows that improvements depend on prefetching accuracy and original D-TLB miss rates. For example, since `Canneal` has one of the highest MMIs, it sees consistently high benefits from 17% to 57% across the MMUs considered.

Figure 15 shows benefits across all D-TLB sizes. Even for the largest D-TLBs, ICC prefetches yield 9% improvements on average. Comparatively, improvements drop for larger TLBs because of lower MMIs. However, high MMI workloads like `Canneal`, `Streamcluster`, and `Ferret` benefit notably for all cases.

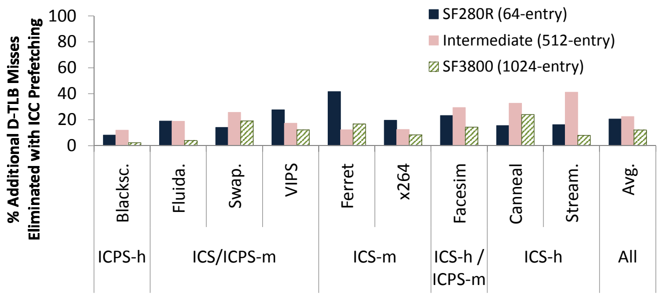


Figure 14. Percentage additional misses eliminated using ICC prefetching with 16-entry PBs versus just enlarging TLBs by 16 entries. ICC prefetching consistently outperforms enlarged TLBs.

Implementation	Description
Fully-Hardware	PB, Leader-Follower scheme in HW. Distance-based Cross-Core in HW (DT, DB, Last Ctxt, Last VP, Last Dist. Regs in HW). Hardware page walking state machine present.
Hardware Prefetch with Software Page Table Walks	PB, Leader-Follower scheme in HW. Distance-based Cross-Core in HW (DT, DB, Last Ctxt, Last VP, Last Dist. Regs in HW). Software page table walks in burst during interrupts.
Hardware/Software Prefetch with Software Page Table Walks	PB, Leader-Follower scheme in HW. Distance-based Cross-Core in SW (DT, Last Ctxt., Last VP, Last Dist. Regs in SW, no DB present). Software page table walks in burst during interrupts.

Table 5. Range of hardware and software ICC implementations considered. For each case, we assess the benefits and overheads.

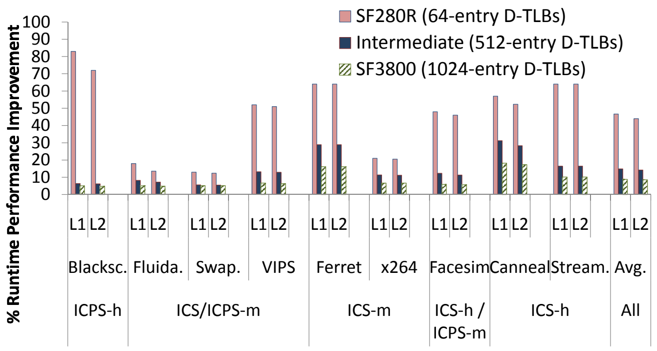


Figure 15. Runtime performance improvements from fully-HW ICC TLB prefetchers for SF280R, Intermediate, and SF3800 MMUs. We show separate performance graphs for the scenario where page table walks for DT predictions hit in the L1 cache and when they hit in the L2 cache. Improvements depend upon prefetching accuracy and frequency of D-TLB misses in the benchmark.

Finally, even the pessimistic assumption of an L2 access for every DT-induced page table walk sees considerable benefits. In this case, ICC prefetching still achieves average improvements of 8% to 43%. This high performance despite potentially expensive L2 accesses occurs because the state machine walks the page table *in parallel* with program execution. Therefore, the L2 access is overlapped with useful work. Furthermore, this delay is seen by prefetches from the Distance-based Cross-Core scheme, only a fraction of the total prefetches. Finally, TLB misses occur on a relatively coarse temporal granularity, allowing sufficient time between lookups for a translation and their subsequent use.

6.2 Hardware Prefetch with Software Page Table Walks

We next consider MMUs with software-managed TLBs. Since SW-managed TLBs rely on the OS for page table walks, we must adapt ICC prefetching accordingly.

While Leader-Follower prefetching remains unaffected for SW-managed TLBs, there are two cases to consider for Distance-based Cross-Core prefetching. When a core misses in both the D-TLB and PB, the OS receives an interrupt. In this case, the interrupt handler may assume responsibility for conducting page table walks for the suggested distances from the DT. If a PB hit occurs, 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, $pred_0$ will then re-index the DT to find its own set of predicted distances (eg. $pred_3$ and $pred_4$). Similarly, $pred_1$ may then be 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 16 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`

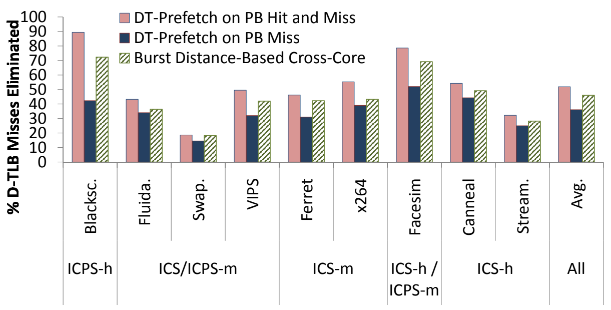


Figure 16. Burst Distance-based Cross-Core prefetching eliminates almost as many D-TLB misses as the fully-hardware case. Results assume that Leader-Follower prefetching remains unaffected.

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

Figure 17 shows the runtime performance gains from ICC prefetching using Leader-Follower and Burst Distance-based Cross-Core prefetching. While we again assume that both schemes include hardware support, we use a burst prefetch approach because of the absence of page walking hardware state machines. Here, we use 16-entry PBs, and a 512-entry, 4-way DT table. Since Burst Distance-based Cross-Core prefetching can yield up to 8 prefetches at a time, we increase DB sizes to 8. We again show separate results for the case when all DT-induced page table walks hit in the L1 cache and when they hit in the L2 cache.

Figure 17 indicates that all the workloads still enjoy significant performance improvements from ICC prefetching, even with software page table walks. For example, SF280R MMUs see 41% and 36% improvements for the L1 and L2 hit cases respectively. Moreover, benchmarks with high D-TLB MMIs like `Canneal`, `Ferret`, and `Streamcluster` still enjoy particularly high benefits consistently across TLB sizes.

While Figure 17 shows that improvements are high, they are lower than the full-hardware case with page-walk-handling state machines. This is because prefetches for DT predictions are now done in the interrupt handler rather than by the state machine in parallel with workload execution. Moreover, since Burst Distance-based Cross-Core prefetching does not prefetch on both PB hits and misses (like in the fully-hardware case), performance benefits decrease. This explains why, for example, `Blackscholes'` performance improvement drops by roughly 12% from the full hardware case. Nevertheless, performance benefits remain substantial (7-41% for SF280R MMUs with L1 access).

Finally, the impact of page table walks missing in the L1 cache and needing to access the L2 cache is still marginal; however, it is more pronounced than for the fully-hardware case. This is because the prefetches are processed by the interrupt handler and therefore, the cost of going to the L2 cache cannot be overlapped with useful program execution. Nevertheless, the L2 hit case still sees average improvements of 36%, 11%, and 6% for the SF280R, Intermediate, and SF3800 MMUs.

6.3 Hardware/Software Prefetch with Software Page Table Walks

The previous section addressed performance implications of moving page table walks into software but leaving the prefetchers in hardware. We now assess 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. Fur-

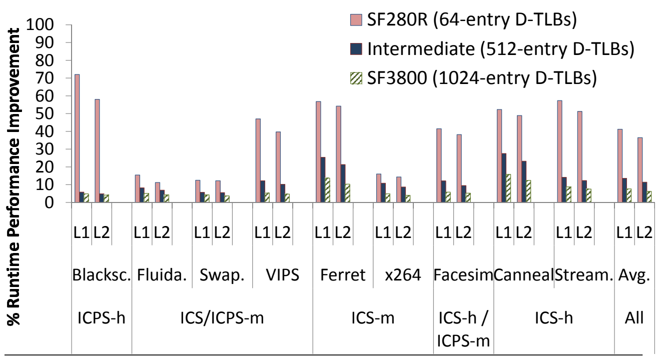


Figure 17. Runtime performance improvements from Leader-Follower and Burst Distance-based Cross-Core prefetchers for SF280R, Intermediate, and SF3800 MMUs. The components are all in hardware but page table walks performed in software. Significant improvements are seen across all benchmarks with an average of 6% to 41%.

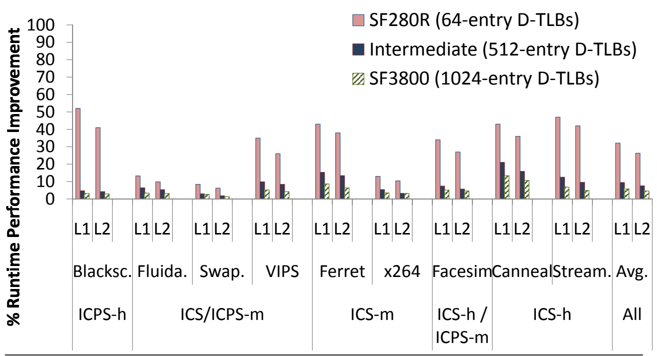


Figure 18. Runtime performance improvements from hardware Leader-Follower and software Burst Distance-based Cross-Core prefetchers for SF280R, Intermediate, and SF3800 MMUs. Significant improvements are seen across all benchmarks with an average of 4% to 32%.

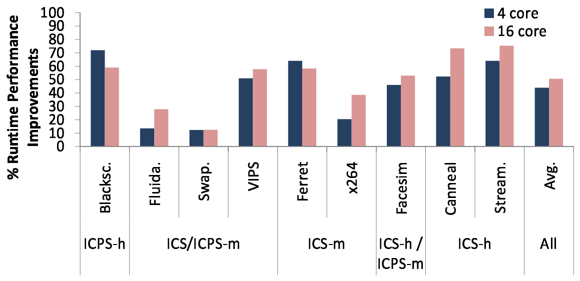


Figure 19. Runtime performance improvements from ICC prefetching for 4-core and 16-core CMPs with SF280R MMUs. Note that higher core counts increase benefits on average from 43% for 4 cores to 49% for 16 cores. ICS-h `Canneal` and `Streamcluster` benefit particularly.

thermore, since Leader-Follower prefetching operates without software intervention, it too can remain a purely hardware operation.

In contrast, we now place the DT purely in software. Since we use Burst Distance-based Cross-Core prefetching, we access the DT from the interrupt handler and burst-prefetch up to 8 translations every time a D-TLB and PB miss occurs. We again assume a 512-entry, 4-way DT, but this time pin the structure 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. We assume that the first DT lookup in every interrupt hits in the L2 cache. For the DT organization we consider, 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 case rarely occurs since multiple predictions usually arise from the same set.

After the predictions are extracted from the DT, we must perform the associated page table walks. We again separately consider performance results assuming L1 and L2 page table hits.

As described, this scheme has minimal hardware and software requirements. Modest 16-entry PBs are assumed and a software table maintains the DT. Even the DB structures are not required in this scheme because of the removal of the hardware DT.

Figure 18 shows the performance improvements of ICC prefetching when using this combined hardware/software approach. Again, notable performance improvements exist for all considered workloads. In fact, SF280R MMUs see an average of 31% and 26% improvements for the L1 and L2 hit cases respectively. Even for the largest TLBs from SF3800, we see average improvements of 4% to 6%. Moreover high MMI benchmarks like `Canneal`, `Ferret`, and `Streamcluster` also enjoy consistently high performance gains across TLB sizes.

Figure 18 does show lower performance gains however, than the hardware schemes of Section 6.1 and 6.2. One reason for this is that, compared to the fully-hardware case, no state handling machine exists to walk the page table for DT predictions in parallel with program execution. Moreover, not only do we now have to conduct page table walks in the interrupt handler, we must also perform DT lookups in the handler. This further serves to increase runtime, particularly when DT lookups require accesses to multiple sets that do not fit in the same cache line.

Nevertheless, Figure 18 indicates that significant scope remains for performance improvements using this modest combination of hardware and software. Especially in cases when even moderate hardware overheads are undesirable, this software approach provides a valuable and effective alternative.

7. Discussion

7.1 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 our results up to now have assumed a 4-core CMP, we now quantify the performance benefits on a 16-core CMP.

Figure 19 compares the runtime performance improvements from ICC prefetching 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. DB sizes are set to 4 and a hardware state machine is used to walk the page table for predictions from the DT. Moreover, we show performance results assuming that the page table walks for DT suggestions all miss in the L1 cache and hit in the L2 cache.

Figure 19 shows that ICC prefetching improves performance even at greater core counts; on average the benefits rise from 43% for 4 cores to 49% for 16 cores. This can be attributed to the fact that in many benchmarks, D-TLB MMIs increase at higher core counts. This is because while the instruction counts per thread decrease, the number of D-TLB misses do not decrease commensurately. In addition, ICS and ICPS TLB misses actually increase for higher cores since there are more cores which potentially have correlating miss patterns. This explains why ICS-h workloads `Canneal` and `Streamcluster` see the most benefits at 16 cores.

Figure 19 shows that `Blackscholes` and `Ferret`, however, do have lower performance improvements in the 16-core case. These workloads differ from the rest in that their D-TLB MMIs actually drop at higher core counts, lowering the potential benefits of any scheme targeting D-TLB behavior. Nevertheless, even for these workloads, ICC prefetching gives substantial improvements of 57% and 58% for `Blackscholes` and `Ferret` respectively.

7.2 Handling Multiprogramming

Adapting ICC prefetchers for multiprogrammed workloads is readily accomplished. First, in Leader-Follower prefetching, confidence counters detect and minimize bad prefetches. Because misses in the D-TLB and PB might re-initiate prefetching however, a second counter per-core is needed to track how many total prefetches from the core were bad. When this rises above a predefined threshold, Leader-Follower prefetching may be prevented from re-initiating until a context switch on the core, when we would want to re-evaluate the use of Leader-Follower prefetching.

Second, Distance-based Cross-Core prefetching may be used for multiprogramming with no additional hardware. Since each DT entry has a context ID, prefetching is based on patterns seen in the same application (though it might be based on multiple threads). Therefore, in a scenario where multiple single-threaded workloads run, prefetching would only be initiated based on patterns seen within cores, as in uniprocessor Distance-based prefetching.

7.3 Accommodating Superpaging

Much recent research in academia and industry has recognized the fact that future workloads will demand greater TLB reach (the maximum size of memory mapped by a TLB) and has proposed *superpaging* as a potential solution [22, 23]. Superpages use the same linear address space as conventional paging but have sizes that are power-of-two multiples of the baseline page size and are aligned in both virtual and physical memory. The obvious benefit of superpages is that they permit greater TLB reach without an increase in TLB size. As a result, commercial processors typically support multiple superpage sizes.

Because ICC TLB prefetching operates orthogonally to superpaging, we anticipate that both techniques may be accommodated in contemporary CMPs. However, analyzing the performance implications of combining both approaches is a complex issue. First, since superpaging implies larger pages, the probability of multiple threads accessing the same page increases. This implies that Leader-Follower prefetching will become more effective. Second, the potential span of strides would also decrease, making strides for Distance-Based Cross-Core prefetching easier to deduce. At the same time, superpaging may also reduce intrinsic TLB misses. As a result, TLB prefetching may fundamentally matter less, although TLB-intensive workloads will always exist. Given this set of complex tradeoffs, a thorough quantitative treatment is out of the scope of this paper but presents an important future direction.

8. Conclusion

Our primary goal in this work has been to show that TLB miss correlations between multiple CMP cores can be exploited to eliminate TLB misses, thereby boosting performance significantly. To accomplish this, we have proposed and evaluated two inter-core cooperative TLB prefetchers: Leader-Follower and Distance-based Cross-Core prefetchers. Not only do these schemes eliminate a considerable number of TLB misses individually, they can be combined with modest hardware to eliminate 13% to 89% of workload TLB misses.

We have also explored hardware and software implementations ranging from a fully-hardware case with average performance improvements of 8% to 46% to a hardware/software hybrid with improvements of 4.5% to 32% for multiple TLB sizes. Moreover, we have shown that benefits are even greater with higher core counts.

Ultimately this work may be used by designers to augment contemporary hardware and software-managed MMUs with simple mechanisms to tackle the increasing TLB-pressure of emerging parallel workloads. Moreover, our results point to a range of possibilities, in both hardware and software, to eliminate TLB misses through prefetching. We believe that this flexibility offers valuable opportunities for more intelligent TLB designs in the future.

9. Acknowledgements

We thank the anonymous reviewers for their feedback. We also thank Li-Shiuan Peh for her suggestions on improving the quality of our submission and Chris Bienia for his help with understanding the PARSEC workloads. Finally, we also thank Mahmut Kandemir for his suggestions to improve our final version of the paper.

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. In addition, this work was supported in part by the National Science Foundation under grant CNS-0627650.

References

[1] T. Anderson et al. The Interaction of Architecture and Operating System Design. *Intl. Symp. on Architecture Support for Programming Languages and Operating Systems*, 1991.

[2] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[3] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[4] J. B. Chen, A. Borg, and N. Jouppi. A Simulation Based Study of TLB Performance. *Intl. Symp. on Computer Architecture*, 1992.

[5] T. Chen and J. Baer. Effective Hardware-based Data Prefetching for High-Performance Processors. *IEEE Trans. on Computers*, 1995.

[6] D. Clark and J. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1), 1985.

[7] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. *Intl. Conf. on Parallel Processing*, 1993.

[8] H. Huck and H. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. *Intl. Symp. on Computer Architecture*, 1993.

[9] B. Jacob and T. Mudge. Software-Managed Address Translation. *Intl. Symp. on High Performance Computer Architecture*, 1997.

[10] B. Jacob and T. Mudge. A Look at Several Memory Management Units: TLB-Refill, and Page Table Organizations. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[11] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 1998.

[12] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. *Intl. Symp. on Computer Architecture*, 1997.

[13] G. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks. *ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, 2002.

[14] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *Intl. Symp. on Computer Architecture*, 2002.

[15] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Comp. Arch. News*, 2005.

[16] D. Nagle et al. Design Tradeoffs for Software Managed TLBs. *Intl. Symp. on Computer Architecture*, 1993.

[17] X. Qui and M. Dubois. Options for Dynamic Address Translations in COMAs. *Intl. Symp. on Comp. Arch.*, 1998.

[18] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. *ACM Transactions on Modeling and Computer Simulation*, 1995.

[19] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-Based TLB Preloading. *Intl. Symp. on Comp. Arch.*, 2000.

[20] V. Srinivasan, E. Davidson, and G. Tyson. A Prefetch Taxonomy. *IEEE Transaction on Computers*, 53(2), 2004.

[21] Sun. UltraSPARC III Cu User's Manual. 2004.

[22] M. Talluri. Use of Superpages and Subblocking in the Address Translation Hierarchy. *PhD Thesis, Dept. of CS, Univ. of Wisc.*, 1995.

[23] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1994.

[24] Virtutech. Simics for Multicore Software. 2007.