Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors

Abhishek Bhattacharjee and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{abhattac, mrm}@princeton.edu

Abstract

Translation Lookaside Buffers (TLBs) are a staple in modern computer systems and have a significant impact on overall system performance. Numerous prior studies have addressed TLB designs to lower access times and miss rates; these, however, have been targeted towards uniprocessor architectures. As the computer industry embraces chip multiprocessor (CMP) architectures, it is important to study the TLB behavior of emerging parallel workloads.

This work presents the first full-system characterization of the TLB behavior of emerging parallel applications on real-system CMPs. Using the PARSEC benchmarks, representative of emerging RMS workloads, we show that TLB misses can hinder system performance significantly. We also evaluate TLB miss stream patterns and show that multiple threads of a parallel execution experience a large number of redundant and predictable misses. For our evaluated benchmarks, 30% to 95% of the total misses fall under this category. Our results point to the need for novel TLB designs encouraging inter-core cooperation, either through hierarchically shared TLBs or through inter-core TLB prediction mechanisms.

1. Introduction

Microprocessors supporting paged virtual memory employ a Memory Management Unit (MMU) for virtual to physical address translation and memory reference validation. To avoid high-latency accesses to operating system page tables, MMUs store translations in instruction and data Translation Lookaside Buffers (TLBs). While there are a number of options for TLB placement and lookup [13], most systems place them in parallel with the first-level cache, effectively inserting them in the critical path of processor pipelines. As a result, TLBs play a crucial role in processor performance [4, 11, 12, 14].

Several solutions have been proposed to improve TLB performance, both in software and in hardware. In particular, solutions addressing TLB characteristics such as size, associativity, and multilevel hierarchies have a significant impact on miss latencies and access times [3, 16]. Other strategies involve the use of superpages [17] and prefetching techniques to hide the cost of TLB misses [11, 15]. While effective, these strategies all target uniprocessor designs. The advent of chip multiprocessors (CMPs) necessitates a shift from the traditional uniprocessor focus to understanding how parallelism affects the virtual memory system, as well as TLB latencies and miss rates.

In this paper, we present the first full-system characterization of TLB behavior for emerging parallel workloads on CMPs. Our evaluations use the PARSEC benchmarks [2], representative of emerging parallel applications. These benchmarks come from a variety of application domains, ranging from financial analysis to media processing, and use both data-parallel and pipeline-parallel schemes. Not only is TLB behavior crucial to the performance of these workloads, but we also find substantial correlation in the TLB misses experienced across multiple cores of a CMP. Our work therefore makes the case for TLB designs to exploit inter-core correlation either through shared and hierarchical architectures or through inter-core TLB prediction schemes. Our specific contributions are as follows:

- We perform the first full-system characterization of TLB misses for emerging parallel workloads. These workloads suffer significantly from TLB misses, with the benchmark `Canneal` spending as much as 0.7 cycles per instruction (CPI) per core on D-TLB misses on a 4-core AMD Opteron.

- Across a range of core counts and TLB sizes, we show that multiple threads often TLB miss on the same virtual to physical address translation. Because multiple threads usually operate on similar sets of data and instructions, up to 95% of all TLB misses in our benchmarks occur on translations already missed upon by another thread.

- We also investigate the presence of stride patterns in TLB accesses across threads. We see that TLB misses that are unique to a single thread are often a predictable stride away from the TLB miss of another thread. For example, for the benchmark `Blackscholes`, 90% of all unique TLB misses are of the form where thread N's D-TLB miss is on a virtual address 4 pages away from a previous TLB miss by thread N-1.

Overall, this work is an early characterization that lays the foundation for future CMP TLB hardware designs, hardware and software management policies, and prediction schemes targeted at hiding TLB miss latencies in CMPs. In

Benchmark	Domain	Parallelization		Data Usage		Data Working Set Size	
		Model	Granularity	Sharing	Exchange	Native	Simlarge
Blackscholes	Financial Analysis	Data-parallel	Coarse	Low	Low	2MB	2MB
Canneal	Engineering	Unstructured	Fine	High	High	2GB	256MB
Facesim	Animation	Data-parallel	Coarse	Low	Medium	256MB	256MB
Ferret	Similarity Search	Pipeline	Medium	High	High	128MB	64MB
Fluidanimate	Animation	Data-parallel	Fine	Low	Medium	128MB	64MB
Streamcluster	Data Mining	Data-parallel	Medium	Low	Medium	256MB	16MB
Swaptions	Financial Analysis	Data-parallel	Coarse	Low	Low	512KB	512KB
VIPS	Media Processing	Data-parallel	Coarse	Low	Medium	16MB	16MB
x264	Media Processing	Pipeline	Coarse	High	High	16MB	16MB

Table 1. Summary of PARSEC benchmarks used in our TLB studies. Note the wide range of application domains, varying parallel models, granularities and data sharing characteristics. Data working set sizes for input types *Native* and *Simlarge* are also provided.

particular, our results indicate that exploiting TLB miss information among cores can lead to significant performance improvements by eliminating redundant TLB misses as well as those predictable by stride patterns.

Our paper is structured as follows. Section 2 discusses background and related work. In Section 3, we detail our choice of benchmarks and experimental infrastructure. Section 4 then presents real-system CMP TLB performance studies, highlighting cases of severe I-TLB and D-TLB behavior. Then, in Section 5, we evaluate TLB miss redundancy followed by a study of stride patterns in Section 6. Finally, we conclude in Section 7.

2. Background and Related Work

Because TLBs are performance-critical and are accessed on every instruction and data reference, CMPs typically provide private per-core TLBs. Each TLB is therefore largely oblivious (except for shootdowns) to the behavior of the others. These TLBs are either *hardware-managed* or *software-managed*. On a miss, a hardware-managed TLB uses a hardware state machine to walk the page table, locate the mapping, and insert it into the TLB. This design is efficient as it perturbs the pipeline only slightly. When the state machine handles a TLB miss, there is no need to take an expensive interrupt. Moreover, miss handling does not pollute the instruction cache. In the worst case, a few lines of the data cache may be polluted when scanning through the page table. Typical hardware-managed TLB miss latencies range from 10 to 50 cycles [9] and are commonly adopted by x86 architectures [8, 19].

The primary disadvantage of hardware-managed TLBs is that they require the page table organization to be fixed; the operating system (OS) has no flexibility in choosing or modifying designs. In contrast, RISC architectures such as MIPS or SPARC often use software-managed TLBs [7, 12]. In these schemes, the operating system receives an interrupt on a TLB miss and vectors into a specific miss handler, which walks the page table and refills the TLB. Since the OS has full control of page table handling, the data structure is flexible. However, there can be an associated performance cost. First, the use of precise interrupts means that the pipeline must be flushed, removing a possibly large number of instructions from the reorder buffer. Second, the miss handler itself is usually 10 to 100 instructions long [9] and may miss in the instruction cache, adding to the miss latency. Finally, the data cache may also be polluted through the course of handling the miss.

Numerous prior works have studied the behavior of benchmarks and operating systems on these TLB designs for uniprocessor architectures. Typical TLB studies in the 1980s and 1990s placed TLB handling at 5-10% of system runtime [4, 12, 14]. However, Huck and Hays showed that in extreme cases, overheads can be as high as 40% of the total runtime [5]. Furthermore, Anderson showed that software-managed TLB miss handlers are among the most commonly executed primitives [1] while Rosenblum et al. [14] demonstrated that these handlers can account for 80% of the kernel's computation time. More recently, Kandiraju and Sivasubramaniam showed that D-TLB handling can amount to 10% of the runtime of SPEC CPU2000 workloads [10]. Although most of these studies address data from the 1990s, their insights on the importance of TLB miss handling still apply to contemporary systems.

Unlike these previous studies, our work focuses on the TLB miss behavior of emerging parallel workloads on novel CMP architectures. We wish to study not only the impact of these workloads on CMP system performance, but also opportunities for improving TLB performance by exploiting cooperation among multiple cores on chip.

3. Methodology

Our goal is to study the impact of parallel workloads on real-system TLBs. We also wish to analyze potential patterns in TLB misses across cores. To meet these goals, we need to accomplish three objectives. First, we must choose a set of benchmarks representative of emerging parallel applications on CMPs along with appropriate input data sets. Second, to quantify real-system TLB performance issues, we need to choose an appropriate system to run our workloads. Third, since a real system does not provide easy access to the actual virtual/physical address pairs causing TLB misses, we also need to choose a software simulator to study inter-core patterns in TLB misses. The following sections present our methodology choices and setups.

System	1.8GHz 4-core AMD Opteron (K8)
Pipeline	3-way superscalar, 72-entry ROB
L1 Caches	64KB I and D Cache (dual-ported) (virtually indexed, physically tagged)
MMU	HW-managed, per-core, 2-level TLB
L1 I-TLBs	40-entry, fully assocative
L1 D-TLBs	40-entry, fully associative 2 D-TLBs, one per L1 D-Cache port
L2 TLBs	512-entry, 4-way
TLB Latencies	Avg. L1 Miss, L2 Hit: 5 cycles Avg. L1 Miss, L2 Miss: 25 cycles

Table 2. Architecture of AMD Opteron sytem used to study the severity of TLB misses on parallel workloads.

3.1 Benchmarks and Input Sets

Our studies use benchmarks from PARSEC, a novel benchmark suite focused on emerging multithreaded workloads representative of next-generation shared-memory programs for CMPs [2]. Table 1 lists the nine PARSEC workloads used in this study[1]. To ensure that our observations are indeed general across a range of parallelization schemes and workloads, we choose benchmarks from a variety of application domains using multiple parallelization schemes (unstructured, pipeline, and data-parallel), parallelization granularities, and inter-core communication characteristics.

Table 1 shows working set sizes for the PARSEC *Native* and *Simlarge* input data sets. The *Native* inputs are intended to study application performance on real machines; we therefore use these for our real-system characterization to realistically stress the TLBs. Unfortunately, these input sets exceed computational demands considered feasible for simulation by several orders of magnitude. Therefore, our simulator-based studies of inter-core TLB miss patterns use the *Simlarge* data sets. These input sets use the largest possible working sets and amounts of parallelism manageable by software simulations.

3.2 Real-System CMP TLB Performance

To assess the real-system impact of TLB misses, we run our workloads on a CMP with high-performance, hardware-managed TLBs. As detailed in Table 2, our target machine uses a 2-level TLB hierarchy. Since the L1 caches are virtually-indexed and physically-tagged, TLB translations are required for every L1 reference. In addition, the L1 D-Cache is dual ported, with one L1 D-TLB per port. Table 2 also gives miss latencies for each TLB level [6, 19].

The 4-core AMD Opteron chip includes hardware performance monitoring counters (PMCs), which can be configured to monitor system events without disrupting execution flow. We configure the PMCs to track L1 and L2 TLB miss events as well as the total number of instructions retired in the parallel section of our workloads. The PMC system monitors up to four event classes at a given time.

[1]These are the PARSEC workloads that run on both our real-system and simulation infrastructures . We plan to study the other benchmarks in the future.

System	Ultrasparc III Cu CMPs (4, 8, 16 core)
OS	Sun Solaris 10
MMU	SW-managed, per-core TLBs
	Simulated MMU Architectures
SF 280R	64-entry, 2-way I-TLB and D-TLB
SF 3800	16-entry, fully assoc. I-TLB (locked/unlocked pages) 128-entry, 2-way I-TLB (unlocked pages) 16-entry, fully assoc. D-TLB (locked/unlocked pages) 2 x 512-entry, 2-way D-TLBs (unlocked pages)

Table 3. Simulated Sun Fire server MMUs in Simics.

We therefore collect our results over two benchmark runs, one for I-TLB and one for D-TLB events.

3.3 Simulation Infrastructure

We use Virtutech Simics [18] to study in more detail the particular virtual/physical address requests resulting in TLB misses. Table 3 shows how our Simics CMP models Sun's Ultrasparc III Cu processors with a variety of core counts. We focus on two primary MMU architectures, the Sun Fire 280R (representative of Sun's entry-level servers with typical TLB sizes), and the Sun Fire 3800 (containing one of the largest TLB organizations to date). The SF 3800 has a complex MMU architecture with separate 16-entry fully-associative L1 I and D-TLBs used primarily by the OS for locking pages. Moreover, the SF 3800 uses two L1 512-entry D-TLBs for unlocked translations. These are accessed in parallel for each data reference and can be configured by the OS to hold translations for different page sizes. In our simulations, the OS configures both TLBs to the same page size, making the two D-TLBs equivalent to a single 1024-entry D-TLB.

Since both MMUs are software-managed, the OS receives an interrupt on a TLB miss. We instrument the Simics source code to track these interrupts, thereby detecting requested virtual/physical address pairs prompting TLB misses. We then study these pairs for common miss addresses across cores.

3.4 Details of our Approach

To gauge opportunities for improving TLB performance by exploiting inter-core cooperation, we do the following:

1. First, we characterize TLB performance for our benchmarks on the real-system AMD Opteron from Table 2 using the *Native* inputs. Since we will subsequently be using software simulation with the smaller *Simlarge* input sets, it is imperative to relate performance with these inputs to real-system performance using the *Native* inputs. Therefore, we also study real-system TLB performance using *Simlarge* on the AMD Opteron.

2. Second, we run the workloads with *Simlarge* inputs on the simulator to study how often multiple cores TLB miss on the same virtual/physical address translation. We study how this is affected by parallelization characteristics and assess the OS contribution to this redundancy. These studies indicate the potential for inter-core TLB cooperation.

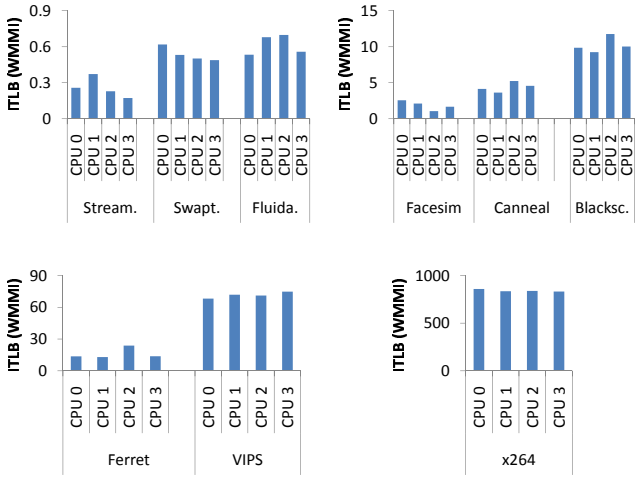

Figure 1. I-TLB weighted misses per million instructions (WMMI) on 4-core Opteron with *Native* inputs.

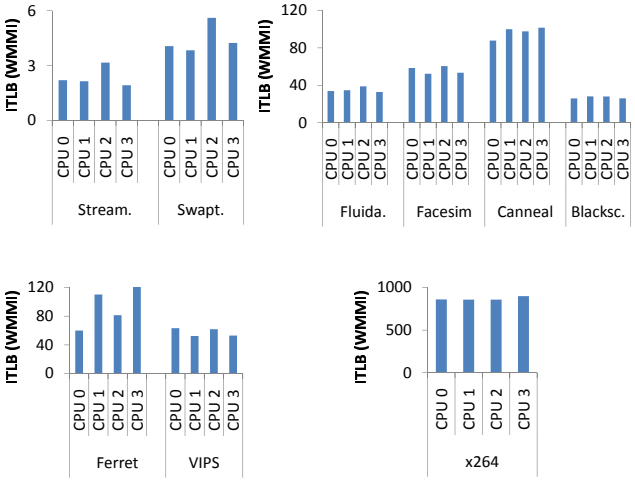

Figure 2. I-TLB weighted misses per million instructions (WMMI) on 4-core AMD Opteron with *Simlarge* inputs.

3. Third, we run the workloads with *Simlarge* inputs to study inter-core strides in TLB misses; for example, if a core misses on virtual page N+1 if another core misses on virtual page N. These patterns may also be exploited by prediction schemes for increased TLB performance.

4. Real-System TLB Miss Characterizations

4.1 Weighted TLB Misses

To characterize real-system CMP TLB performance on the AMD Opteron chip from Table 2, we must first devise a uniform metric of comparison. For this purpose, we choose *Weighted Misses per Million Instructions* (WMMI). In one level of TLB, this would simply be TLB misses per million instructions. However, to aggregate the impact of two levels of TLB misses, we weight by respective miss penalties. For our architecture, this is:

$$WMMI = MMI(L1_{miss}L2_{hit}) + \frac{(L1L2_{penalty}) \times MMI(L1L2_{miss})}{L1_{penalty}}$$

Here MMI($L1_{miss}L2_{hit}$) represents the number of TLB L1 misses per million instructions that result in L2 hits and MMI($L1L2_{miss}$) is the number of TLB L1 misses per million instructions that also result in L2 misses. With the latencies from Table 2, this equation becomes:

$$WMMI = MMI(L1_{miss}L2_{hit}) + 5 \times MMI(L1L2_{miss})$$

4.2 Instruction TLB Performance

We begin our real-system characterization of TLBs by showing WMMI for emerging parallel workloads on I-TLBs. We first present our studies for the *Native* inputs and then analyze how they compare with the *Simlarge* results.

4.2.1 Native Data Inputs

Figure 1 demonstrates the I-TLB WMMI contributions for the PARSEC workloads using *Native* inputs across the 4 cores of the AMD Opteron. We arrange the benchmarks into 4 sub-graphs (note y-axes) in ascending order of the I-TLB WMMI. For each application, we plot each core's TLB misses separately to show the variability that is (or is not) present.

Overall, Figure 1 shows that most benchmarks miss in the I-TLB infrequently. The most severe I-TLB behavior is for `x264` with a WMMI of roughly 960, but even this amounts to a CPI contribution under 0.005.

Second, Figure 1 indicates that substantial variation exists in I-TLB WMMI across benchmarks. In particular, `x264's` WMMI is orders of magnitude higher than the other benchmarks. This is because `x264` uses a pipeline parallel model with one stage per input video frame. Therefore, the benchmark is executed with a number of threads greater than cores. For the *Native* input set on the 4-core AMD Opteron, 512 threads are produced. The benchmark does ensure that only one thread runs on a CPU at a time; however, the greater thread count implies that when a new thread context switches in, there is a burst of misses, increasing WMMI by orders of magnitude.

Third, Figure 1 demonstrates that I-TLB WMMI is similar across cores for each benchmark. This is particularly true for data-parallel applications because multiple threads collaborate by performing similar instructions on different data in the form of a single thread body function.

The bottom line is that there is significant similarity in I-TLB miss contribution across cores because of the collaborative nature of threads. This in turn hints at opportunities for inter-core TLB cooperation to boost performance.

4.2.2 Simlarge Data Inputs

We now study the TLB performance of the benchmarks using *Simlarge* inputs. These are the input sets typically used

in simulation studies. Although one might expect only D-TLB impact from data-set scaling, we find that I-TLBs are affected as well. Figure 2 shows that *Simlarge* inputs can yield I-TLB performance numbers substantially different from *Native* input results. Specifically we note:

First, the WMMI counts *increase* for *Simlarge* inputs. The increase can be by orders of magnitude (eg. `Streamcluster`, `Swaptions`) or by 1.2-2× (eg. `VIPS`, `x264`). In fact, `Fluidanimate's` WMMI increases two orders of magnitude so that it is grouped into a higher sub-graph. This increase can be attributed to the fact that the number of instructions is lowered for *Simlarge*, but I-TLB misses do not scale down commensurately. This is because *Simlarge* inputs are scaled from *Native* in a way that is guaranteed to preserve the code path and typically just reduces the amount of data that the program operates on. Therefore, the number of I-TLB misses are amortized over significantly fewer instructions.

Second, some benchmarks like `x264` see a smaller increase in WMMI with *Simlarge*. This is because I-TLB misses are heavily influenced by thread count. For `x264`, the input set spawns a number of threads much larger than the number of available cores. Therefore, a huge instruction count reduction from 84 billion (*Native*) to 2.1 billion (*Simlarge*) is matched by a thread count reduction from 512 threads to 128 threads. This commensurately decreases I-TLB misses and keeps WMMI roughly similar.

From these observations, we conclude that when using results from microarchitectural studies with *Simlarge*, I-TLB misses are typically more frequent than they would be on a real system with *Native* workloads.

Finally, as with *Native*, *Simlarge* also shows similar inter-core WMMI for a given benchmark, raising the possibility of using inter-core cooperation to boost performance.

4.3 Data TLB Performance

We now assess the D-TLB behavior of the PARSEC workloads with the *Native* and *Simlarge* inputs.

4.3.1 Native Input Sets

Figure 3 indicates that D-TLB misses are particularly severe for the workloads with *Native* inputs. Once again, WMMI contributions from the D-TLB are graphed in ascending order with sub-graphs grouping benchmarks with similar D-TLB miss counts (the application groups are different than in the I-TLB data of Figures 1 and 2). We note the following from Figure 3:

First, D-TLB WMMIs are orders of magnitude higher than their I-TLB counterparts and can be particularly detrimental to system performance. For example, `Canneal` suffers from a WMMI of 123K per core, corresponding to a CPI of 0.7 spent on D-TLB misses.

Second, the relative D-TLB WMMI suffered by the workloads tracks the working set sizes provided in Table 1 for data-parallel workloads. Therefore `Swaptions`, which has the lowest working set of 256KB also has the lowest WMMI. In contrast, `Canneal` (2GB working set) sees the highest WMMI per core.

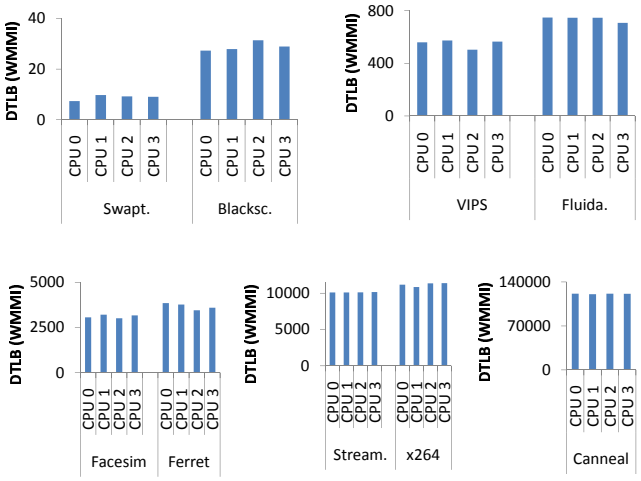

Figure 3. D-TLB weighted misses per million instructions (WMMI) on 4-core Opteron with *Native* inputs.

Third, pipeline-parallel workloads (`Ferret` and `x264`) have WMMI numbers higher than their working sets indicate. For example, while `Ferret` and `Fluidanimate` both have 128MB working sets, the former's WMMI is 5× the latter's WMMI. This is because `Ferret` uses dedicated thread pools per pipe stage. Each pool has enough threads to occupy the entire CMP and therefore, a 4 core system actually runs 16 threads. Furthermore, `Ferret's` working set is made up of an image database that is linearly scanned in its entirety by most of the threads. This high thread count, coupled with memory-intensive linear scans, results in many more D-TLB misses.

Fourth, the exact WMMI numbers and working set sizes are highly dependent on benchmark characteristics. For example, `Canneal's` D-TLB performance is particularly poor because it uses pseudo-random accesses to a huge amount of data that does not fit into the cache or D-TLB. Therefore, the accesses exhibit low spatial and temporal locality [2], increasing D-TLB misses. Moreover, `Canneal` is classified by Bienia et al. as *unbounded*. A workload is "unbounded" if it becomes more useful to users with increased amounts of data. This means that their working sets are expected to grow aggressively in the future, further exacerbating D-TLB performance.

Figure 3 shows that while `Facesim`, `Fluidanimate` and `Streamcluster` have relatively high WMMIs, they still outperform `Canneal` significantly. This is partly due to smaller working sets. In addition, unlike `Canneal`, they are also streaming benchmarks meaning that they exhibit spatial locality in data references. This also contributes to their superior D-TLB WMMI.

Fifth, the D-TLBs also show marked similarity in WMMI across cores for individual benchmarks. Intuitively, this is reasonable considering that multiple threads cooperate on the same data-set and so are likely to have similar TLB misses. Therefore, we again see scope for inter-core TLB cooperation for improved performance.

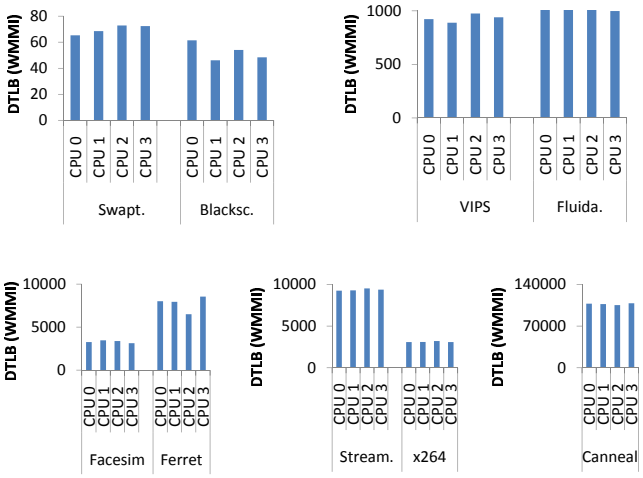

Figure 4. D-TLB weighted misses per million instructions (WMMI) on 4-core Opteron with *Simlarge* inputs.

4.3.2 Simlarge Data Inputs

Figure 4 shows D-TLB WMMI values for the benchmarks using *Simlarge* inputs. The graphs show that:

First, most benchmarks see a 1-2× rise in WMMI when using *Simlarge* over *Native* inputs. For example, the WMMI for `VIPS` rises from 570 to 950. One might expect that the significant downscaling in data-set size (see Table 1) would reduce WMMI. On the contrary, the trend varies with the application. Overall, however, *Simlarge* typically sees a greater reduction in instruction counts than in TLB misses.

The pipeline-parallel `x264` does have a much lower WMMI with the *Simlarge* input. This is because the working set remains the same as for the *Native* input while the number of spawned threads decreases from 512 to 128. This decreases the number of D-TLB misses.

Third, `Canneal's` WMMI for *Simlarge* also drops to roughly 85% of the value from *Native*. Here, we do expect a drop in WMMI due to the large drop in the working set size to 256MB. However, `Facesim`, which has also has a 256MB working set, has a much smaller WMMI. There are two reasons for this. First, `Canneal` accesses the same working set size in half as many instructions as `Facesim`. Second, `Facesim` is a streaming application employing an iterative Newton-Raphson algorithm over a sparse matrix stored in two arrays. It therefore has much better spatial and temporal locality than `Canneal`.

Finally, we again observe the similarity in D-TLB WMMI across cores per benchmark. As with the I-TLBs this is due to the collaborative nature of both data-parallel and pipeline-parallel threads.

Based on these observations, *Simlarge* sees D-TLB behavior similar to real-system *Native* workloads. While D-TLB WMMI values are typically higher than for the *Native* results, this difference is much less pronounced than for I-TLBs. Therefore, one should remember that results with *Simlarge* will typically show a slightly greater impact of TLB behavior than on a real system with *Native* inputs.

4.4 Summary of Observations

Based on this real-system characterization, we draw a few conclusions. First, emerging parallel workloads can stress current TLB designs, even for MMUs with relatively high-performance TLBs (eg. AMD Opteron). This is particularly true for D-TLBs, which are susceptible to stressmarks like `Canneal`. Therefore, it is imperative to research designs to handle this TLB pressure.

Second, inputs for typical microarchitecural simulators such as *Simlarge* usually show poorer TLB performance than the *Native* inputs, particularly for I-TLBs. While this implies that proposed mechanisms to improve TLB performance on *Simlarge* inputs should also be applicable to real-system applications using *Native* inputs, *Simlarge* may overpredict improvements likely on real systems.

Finally, we have noted the similarity of inter-core I-TLB and D-TLB misses. To further investigate the potential for using this behavior for inter-core cooperation, we devote the following sections to studying the virtual/physical address pairs causing TLB misses.

5. Studying Inter-Core Shared TLB Misses

5.1 Definitions, Nomenclature, and Approach

We begin our study of TLB miss redundancy by defining nomenclature used in this section. A TLB miss results in a new TLB entry consisting of the requested virtual page number (VP), the corresponding physical page (PP), the process context ID (CID) for which this translation is valid, protection information (Prot), replacement policy and status bits (Status), and for TLBs supporting multiple page sizes, the particular page size for this translation (PS). Based on this information, we define a *TLB miss tuple* as the 5-tuple, $\langle CID, VP, PP, Prot, PS \rangle$.

To assess correlation and redundancy in TLB misses, we classify some TLB misses as *Inter-Core Shared* (ICS). In an N-core CMP, a TLB miss on core N is ICS if it corresponds to a TLB miss tuple matching the TLB miss tuple of a previous miss on any of the other N-1 cores within a fixed analysis window of M instructions. Furthermore, the number of sharers corresponds to the number of distinct cores whose TLB miss tuples match in this M-instruction window.

The choice of analysis window, M, will affect the degree to which ICS TLB misses are close enough temporally to exploit sharing or prediction schemes. Since TLB misses occur at a relatively coarse temporal granularity, M is set to 1 million instructions for our experiments. For a 10 MMI benchmark, this allows us to compare a typical TLB miss to at least 10 prior misses.

We use the Simics parameters described in Table 3 to quantify ICS across the PARSEC workloads with *Simlarge* inputs. Our results classify all TLB misses by their degree of inter-core sharing (eg. whether they are shared by 2, 3, or all 4 cores on a 4-core CMP).

We note at this point that Simics is a functional simulator. While timing models would provide more insight,

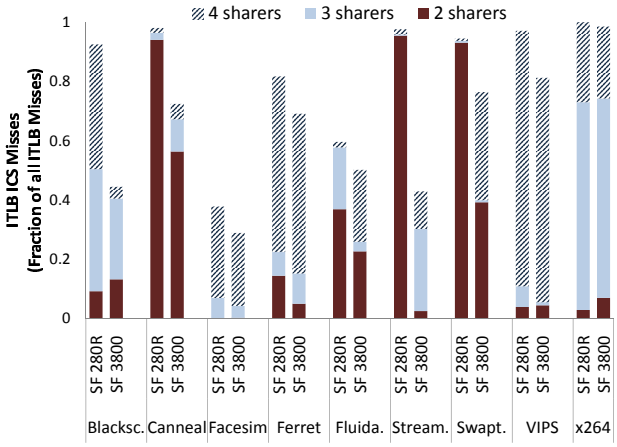

Figure 5. Inter-Core Shared (ICS) I-TLB Misses are a large fraction of total I-TLB misses for both low-end and high-end MMUs on a 4-core CMP.

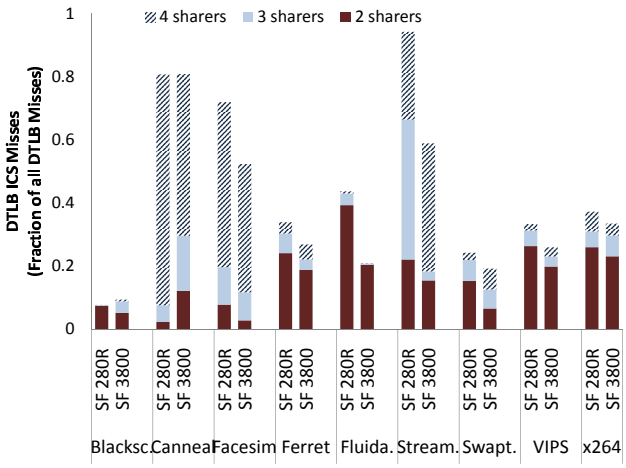

Figure 6. Inter-Core Shared (ICS) D-TLB Misses can be as high as 94% for `Streamcluster` on a 4-core CMP.

we aim to capture TLB behavior on large, realistic datasets, which would be too slow on timing simulators. Moreover, our chosen metric to analyze TLB behavior is that of misses per million instructions, which would remain consistent through timing models. Therefore, our work lays the foundation for future TLB hardware proposals that are carefully evaluated using timing simulators.

5.2 Inter-Core Shared I-TLB Misses

Figure 5 plots ICS I-TLB misses on a 4-core CMP running the PARSEC workloads with *Simlarge* inputs. The two bars for each application represent the SF280R (low-end) and SF3800 (high-end) MMUs. Each bar has components corresponding to TLB misses with 2 sharers, 3 sharers and 4 sharers, normalized to total system I-TLB misses. From these graphs, we note:

First, data-parallel benchmarks benchmarks experience a large number of ICS I-TLB misses across both MMUs. This is because data-parallel applications typically employ a single thread body function for multiple threads. For example, the master thread in `Blackscholes` initializes portfolio data before it spawns worker threads that carry out similar operations on separate parts of the data. This results in over 90% of all I-TLB misses shared by at least 2 cores.

Second, Figure 5 shows that the pipeline-parallel benchmarks, `Ferret` and `x264` have ICS I-TLB misses above 70% and 93% of all I-TLB misses for both MMUs. This is because unlike the data-parallel workloads, which spawn 1 thread per core, the pipeline-parallel workloads are designed to spawn many more threads than cores. However, like the data-parallel workloads, multiple threads in these workloads can execute similar instructions on multiple cores. Therefore, with more threads present and executing similar instructions, ICS sharing is high.

The data shows opportunities to eliminate I-TLB misses through inter-core cooperation. As just one example, novel TLBs that either share entries among cores or use inter-core TLB access prediction would ideally eliminate 50% of the 2-core misses, 66% of the 3-core misses, and 75% of the 4-core misses, improving system performance.

5.3 Inter-Core Shared D-TLB Misses

Figure 6 shows ICS D-TLB miss contributions on a 4-core CMP with the SF280R and SF3800 MMUs. Again, we run the PARSEC workloads with *Simlarge* inputs and classify the ICS misses according to their sharing degree, this time normalized to the total D-TLB miss count. From these results, we observe the following:

First, `Canneal` and data-parallel benchmarks like `Facesim` and `Streamcluster` exhibit high ICS miss contributions, above 55%. Given that these benchmarks particularly stress the D-TLB (Figure 3), this presents a valuable opportunity to improve system performance.

Second, sharing is strongly determined by program characteristics. For example, all of `Canneal's` threads actively share the working set [2]; consequently, 70% of all D-TLB misses are shared among 4 cores. In contrast, `VIPS` mostly shares a modest amount of data between two threads; this causes the high contribution of 2-core shared misses to the ICS D-TLB misses.

Third, Figure 6 shows that 30-40% of the D-TLB misses for the pipeline-parallel benchmarks `Ferret` and `x264` are shared by at least 2 cores. However, larger analysis windows significantly increase sharing. This is because in pipeline-parallel workloads, different algorithmic stages or threads operate on data in a pipeline–common data structures are passed through all the threads during the entire benchmark run. Therefore, when using an analysis window equivalent to the full benchmark runtime, the data sharing increases substantially across cores. We have run experiments to quantify this change and see that over 90% of the TLB misses are ICS for both `Ferret` and `x264` when the analysis window is set to the entire benchmark run (in con-

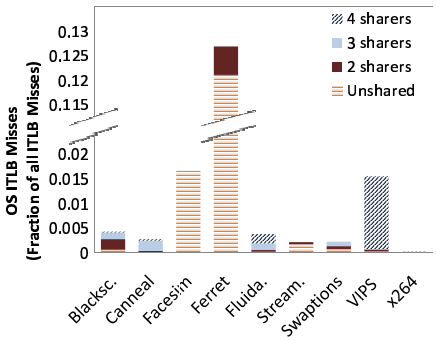

Figure 7. OS contributions to I-TLB misses on the SF280R MMU for a 4-core CMP. The OS can prompt a high number of ICS misses – for example, over 95% of all OS I-TLB misses on `VIPS` are seen on all 4 cores.

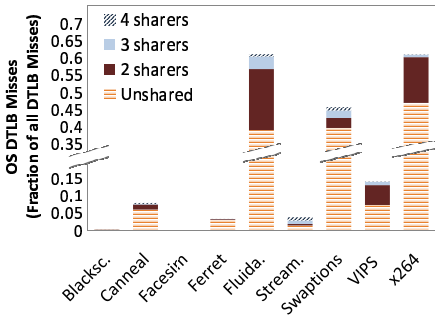

Figure 8. OS contributions to D-TLB misses on the SF280R MMU for a 4-core CMP. The OS can prompt many ICS misses – for example, over 40% of all OS D-TLB misses on `Fluidanimate` are seen on all 4 cores.

trast, data-parallel benchmarks are much less dramatically affected). TLB prediction schemes based on chains of past D-TLB misses [15] may be able to exploit this behavior to reduce D-TLB misses on pipeline-parallel applications.

Hence, there is considerable scope to develop shared TLB architectures and prediction hardware to exploit correlated D-TLB misses. Moreover, since D-TLB misses can severely affect system performance (eg. `Canneal`), we anticipate great gains in performance from these efforts. Note that certain workloads, particularly `Blackscholes`, show few ICS D-TLB misses. Section 6 offers alternatives to cope with these cases.

5.4 Inter-Core Shared OS TLB Misses

We now study the impact of the OS on ICS misses. A number of prior works have established that OS TLB behavior can critically impact system performance [1, 14]. Therefore, we identify TLB misses from Solaris 10 in our simulations by tracking context ID. While this does capture kernel scheduler and system daemon activities, it also implicitly includes operations requested from user-space, eg., system calls to common library routines.

5.4.1 Inter-Core Shared OS I-TLB Misses

Figure 7 presents the Solaris kernel's contribution to I-TLB misses on the 4-core SF280R MMU organization (while we have also studied OS misses on the SF3800, these are similar to the SF280R and are therefore not presented here). The kernel's I-TLB misses are plotted as a fraction of the benchmark's total I-TLB miss count. As before, the OS TLB misses are split into those that are unshared and those that are ICS (with the degree of sharing specified).

Figure 7 indicates that I-TLB contributions from the OS are minimal, with most benchmarks seeing under 2% of their misses from this source. `Ferret` is an exception with above 10% of its I-TLB misses from the OS. Figure 7 also shows that notable ICS I-TLB contributions may exist, as with `VIPS`. Therefore, novel TLB prediction schemes and organizations that exploit application ICS can also exploit ICS in OS I-TLB activity.

5.4.2 Inter-Core Shared OS D-TLB Misses

Figure 8 illustrates D-TLB ICS contributions from the OS. These results are again based on the 4-core SF280R simulations (again, the SF3800 results are similar to this) and show that the OS D-TLB behavior can have a much higher impact on system performance than I-TLBs. In particular, `Fluidanimate`, `Swaptions`, and `x264` experience more than 45% of their D-TLB misses from the OS. For `Swaptions`, this is because its small working set implies a low number of D-TLB misses, increasing the OS contribution to the total. In contrast, the high OS D-TLB count for `x264` arises from its high thread spawn count, causing heavy access to the threading library and kernel process control structures. Figure 8 also shows that many of the OS D-TLB misses are ICS. For example, more than 20% of the OS D-TLB misses of `Fluidanimate`, `VIPS` and `x264` are shared by at least 2 cores.

5.5 Thread Count Versus ICS TLB Misses

While substantial inter-core redundancy exists in TLB misses for a 4-node CMP, it is essential to study how this behavior scales to future CMPs with larger core counts. Therefore, we vary the core counts on our modeled CMP in Simics and study levels of ICS in TLB misses.

Figure 9 shows our observed results for I-TLB misses. For each benchmark, we plot the *average number of sharers* for I-TLB misses through the entire benchmark execution. Our results indicate that the degree of ICS in I-TLB misses increases dramatically in a number of cases, most notably for `Blackscholes`, `VIPS`, and `x264`.

Figure 10 indicates that greater core counts also increase ICS sharing for D-TLBs. This is particularly true for two unbounded benchmarks with severe D-TLB behavior: `Canneal` and `Facesim`. Therefore, not only will the scope for novel TLB architectures and prediction schemes be more pronounced for future CMPs, their potential performance improvements would be substantial.

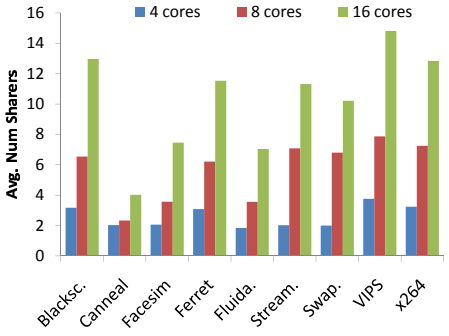

Figure 9. The average number of sharers per I-TLB miss increases with higher core counts, particularly for `Blackscholes`, `VIPS`, and `x264`.

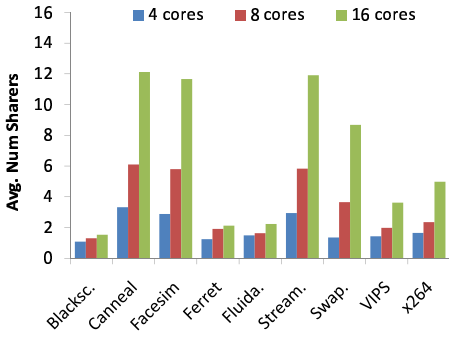

Figure 10. The average number of sharers per D-TLB miss increases with higher core counts, particularly for `Canneal`, `Facesim`, and `Streamcluster`.

6 Studying Inter-Core Stride TLB Misses

The previous sections of the paper have detailed the presence of significant redundancy in inter-core I-TLB and D-TLB miss patterns. However, there remains a set of benchmarks which see only trivial levels of inter-core sharing. The most notable workload is `Blackscholes` in which multiple cores share just under 10% of all D-TLB misses. For these cases, TLB optimizations exploiting inter-core shared misses will provide only modest performance gains. Therefore, we devote this section to exploring alternate patterns in TLB misses. As we will show, benchmarks with low ICS levels can still exhibit predictable stride accesses. For example, if core 0 accesses page N, core 1 accesses page N + 1.

While our studies in this section are pertinent to both I-TLBs and D-TLBs, we focus on D-TLB misses because they are particularly detrimental to performance.

The particular steps in this study are as follows:

1. We begin by examining the runtime TLB behavior of benchmarks with low inter-core sharing, such as `Blackscholes`. This gives us insight into application characteristics influencing ICS.

2. We define *Inter-Core Predictable Stride TLB Misses*, our metric to evaluate predictable stride TLB accesses.

3. We study predictable stride accesses across all workloads and show that although the ICS may be low in certain benchmarks, there are many inter-core stride TLB misses.

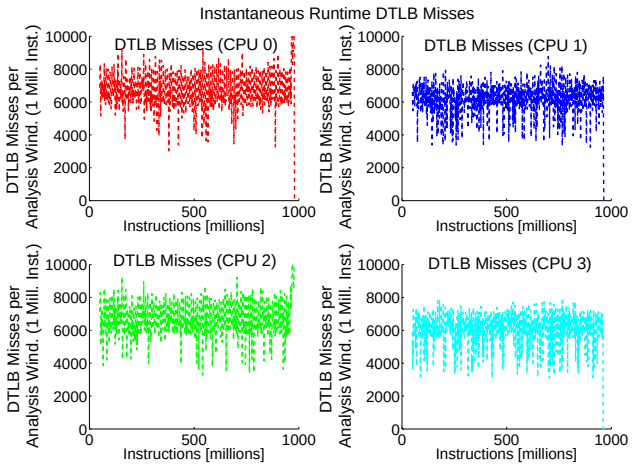

Figure 11. Runtime D-TLB misses per million instructions for `Blackscholes` on a 4-core CMP with SF280R MMUs. Note that all cores experience similar miss counts.

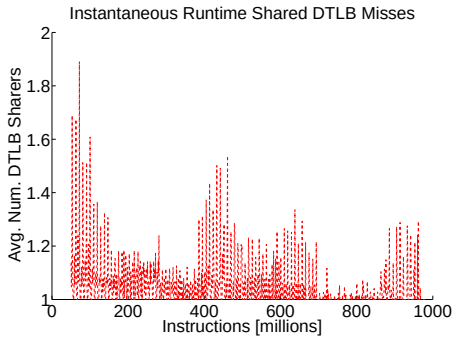

Figure 12. Average number of sharers per D-TLB Miss for `Blackscholes`. Note the low inter-core sharing.

6.1 TLB Misses in Low-ICS Workloads

Figure 11 shows the runtime D-TLB behavior of `Blackscholes` on a 4-core CMP with SF280R MMUs. Each plot represents the progress of a single core. Figure 12 shows the corresponding plot for the average number of cores sharing each D-TLB miss through execution. Based on these graphs, we can see that:

First, Figure 12 shows that sharing is low through the entire benchmark run. On average TLB misses are shared by roughly 1.1 cores. There are instances where this average can rise beyond 1.5, but generally, sharing is very modest. Second, despite the low sharing, Figure 11 shows that all the cores see equivalent D-TLB miss plots through runtime. This indicates that although cores operate on distinct data, they operate similarly on this data and stress the D-TLB equally. As Bienia et al. note [2], the main thread of `Blackscholes` spawns off worker threads that process parts of the portfolio of options independently, operating

	Servicing CPU			
	CPU 0	**CPU 1**	**CPU 2**	**CPU 3**
Requesting CPU 0	0.03%	0.67%	0.21%	**20.56%**
Requesting CPU 1	**20.73%**	0.01%	1.12%	1.04%
Requesting CPU 2	2.21%	**21.04%**	0.01%	0.99%
Requesting CPU 3	1.23%	0.23%	**21.52%**	0.02%

Table 4. Percentage of `Blackscholes`' unshared D-TLB misses covered by strides of +4 pages in analysis window of 1 million instructions on a 4-core CMP with SF280R MMUs. Roughly 84% of all misses fall in this stride.

similarly but without communication.

In fact, this behavior is true of a number of benchmarks, particularly data-parallel ones which assign different threads to operate similarly on different parts of the data. Therefore, one might expect that some benchmarks employ stride accesses–for example if thread 0 operates on page N of a data structure, thread 1 operates on page N+1. If sufficiently predictable, these strides could be exploited by novel TLB prediction schemes. We define this concept more precisely in the next section.

6.2 Defining Inter-Core Predictable Stride TLB Misses

Similar to the TLB miss tuple from Section 5.1, we need an information tuple to compare TLB miss addresses for stride patterns. For this purpose, we define a *Stride TLB Miss Tuple* as the 3-tuple $\langle CID, VP, PS \rangle$.

In this context, we define an *Inter-Core Predictable Stride TLB Miss* (ICPS) in the following way. Suppose that at instruction I_i, core 0 has a TLB miss with the stride TLB miss tuple $\langle CID_i, VP_i, PS_i \rangle$. Now suppose that at a later instruction I_j, core 1 has a TLB miss with stride TLB miss tuple $\langle CID_j, VP_j, PS_j \rangle$. These misses are considered ICPS with a stride of S if the following hold:

1. $I_j - I_i < M$ instructions (analysis window)
2. $\langle CID_j, VP_j, PS_j \rangle = \langle CID_i, VP_i + S, PS_i \rangle$

In this terminology, we call core 1 the *Requesting CPU* (since it sees the TLB miss a stride S away) and core 0 the *Servicing CPU* (since it is the CPU relative to which the stride is made).

Based on this metric, we sweep through a number of different potential stride values for all the workloads. Again, we use 1 million instructions for the value M. Our results are presented in the next section.

6.3 ICPS TLB Miss Results

We again use the example of `Blackscholes` to show ICPS results. By analyzing miss patterns for a number of stride values, we find that `Blackscholes` heavily employs inter-core strides of -4 and +4 pages.

Table 4 shows the percentage of `Blackscholes'` unshared D-TLB misses now covered by a stride of +4 pages. Each row index represents the requesting CPU while the column index represents servicing CPU. Each table entry provides the percentage of unshared D-TLB misses predictable in strides of +4 pages for the requesting and servicing CPU pair.

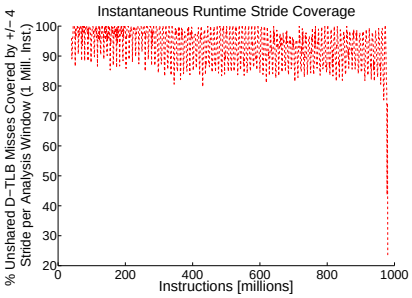

Figure 13. Runtime percentage of unshared D-TLB misses with +4 or -4 page strides in `Blackscholes` on a 4-core CMP with SF280R MMUs. Note that 85% to 98% of all D-TLB misses are consistently covered by these strides.

Overall, Table 4 shows that roughly 84% of unshared D-TLB misses in `Blackscholes` are covered by +4 page strides. For example, 20.73% of all unshared D-TLB misses fall in this +4 page stride when CPU 1 is requesting and CPU 0 is servicing. Similarly, another 21% of all unshared D-TLB misses are covered by +4 page strides when CPU 2 is requesting and CPU 1 is servicing. The two remaining major contributions from +4 page strides occur when CPU 3 requests and CPU 2 services, and when CPU 0 requests and CPU 3 services.

Figure 13 shows a runtime plot of the percentage of unshared D-TLB misses covered by strides of +4 or -4 pages for `Blackscholes`. As shown, the inclusion of -4 page strides in addition to +4 pages raises the stride coverage to values consistently higher than 90%. Therefore, while `Blackscholes` may have little ICS to exploit, the marked presence of strides in access hints at TLB inter-core stride-based prediction schemes for performance improvements. Moreover, we expect the benefits of these approaches to increase at higher core counts.

While we have focused on `Blackscholes` in this example, a number of benchmarks show stride patterns in TLB misses. Table 5 shows the prominent D-TLB strides experienced by all the tested workloads on a 4-core CMP with SF280R MMUs. The second column of the table shows the percentage of total D-TLB misses that are inter-core shared by at least two cores for each workload. The third and fourth columns represent the dominant D-TLB stride patterns and the percentage of *unshared* D-TLB misses that can be captured by these strides. Finally, the fourth column combines the D-TLB misses that are shared and captured by strides to provide the percentage of total misses that would be predictable by novel hardware exploiting inter-core TLB cooperation. Furthermore, the benchmarks are arranged in descending order in terms of these total predicable TLB misses.

Overall, Table 5 shows that most benchmarks have significant stride patterns that raise the predictable D-TLB miss numbers over 50% in most cases. Moreover, while stride patterns can help with benchmarks with low shared D-TLB misses (eg. `VIPS`), they can also improve applications like `Facesim`, which already has high sharing.

Benchmark	% of Total D-TLB Misses Inter-Core Shared (by at least 2 cores)	Prominent Stride Values	% of Total Unshared D-TLB Misses Inter-Core Predictable Stride	% of Total D-TLB Misses Predictable by Sharing or Strides
Streamcluster	94.2%	No prominent strides	None	94.2%
Blackscholes	8.7%	+4, -4 pages	93.2%	93.8%
Facesim	73.1%	+2, -2, +3, -3 pages	76.1%	93.5%
Canneal	83.2%	No prominent strides	None	83.2%
VIPS	33.7%	+1, -1, +2, -2 pages	55.1%	70.2%
Fluidanimate	44.1%	+1, -1, +2, -2 pages	36.0%	64.2%
Swaptions	24.2%	+1, -1, +2, -2 pages	42.2%	56.1%
x264	37.3%	+1, -1, +2, -2 pages	16.2%	44.3%
Ferret	33.8%	No prominent strides	None	33.8%

Table 5. Stride coverage for PARSEC workloads in order of the fraction of total D-TLB misses predicted by either inter-core shared misses or inter-core predictable stride misses. All these results are for a 4-core CMP with SF280R MMUs. Note that above 50% of D-TLB misses for most benchmarks are predictable with a combination of the two approaches.

The particular stride patterns vary across benchmarks. While we have shown that Blackscholes sees strides in a regular pattern between core N and N+1, other benchmarks can use strides more irregularly. For example, in VIPS, a significant number of stride D-TLB misses are requested by cores 0, 1, and 3 and serviced by core 2. Intelligent TLB stride prediction hardware will need to be adaptive to these benchmark nuances.

Therefore, we have shown that ample opportunity exists to take advantage of inter-core predictable stride TLB misses in the absence (or even presence) of inter-core shared TLB misses. Inter-core TLB cooperation schemes and hardware designed to exploit this behavior can be expected to raise TLB performance considerably.

7. Conclusion

Our full-system exploration of the TLB behavior of emerging parallel workloads on real-system CMPs has shown the growing importance of TLBs in CMP design. Specifically we have shown that D-TLB performance is particularly poor for certain benchmarks such as Canneal. As workloads become more complex with larger data sets (possibly unbounded), it will be imperative to overcome these TLB performance problems. Moreover, we have shown how the rates of I-TLB and D-TLB misses on CMPs are strongly determined by application characteristics and input data sets.

The results also indicate that many I-TLB and D-TLB misses are inter-core shared. This is particularly crucial for the D-TLB behavior of Canneal, Streamcluster, and Facesim, all of which are in dire need of D-TLB performance improvements. In addition, we have shown that for benchmarks like Blackscholes which have low D-TLB inter-core sharing, prominent inter-core predictable stride patterns exist.

Future MMU organizations will need to exploit this behavior to counter the performance limitations of contemporary TLBs in CMPs. Our results clearly advocate inter-core cooperation. For example, TLB hardware that predicts future accesses by analyzing other cores may substantially improve performance. Another approach might be to explore shared, hierarchical TLB organizations. While L1 TLBs are typically too performance-critical to be shared among cores, it may be practical to investigate shared or hierarchically shared L2 and L3 TLBs. We expect that these schemes would be feasible for both HW and SW-managed TLBs.

Overall, our work presents the first detailed characterization study of the TLB behavior of CMPs. To parallel programmers using PARSEC, this characterization provides guidance on the expected performance of TLBs and considers how the application structure influences this behavior. For OS designers, our work provides a foundation for studying newer virtual memory organizations to mitigate poor TLB behavior of parallel workloads. Finally, for hardware research, we offer insights that may help computer architecturs select appropriate workloads for stressing TLB behavior in their parallel studies.

8. Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Joel Emer and Li-Shiuan Peh for their suggestions on improving the quality of our submission. Finally, Chris Bienia's help with the PARSEC workloads and insights on their behavior were instrumental to our research. We would also like to thank Virtutech for providing the Simics source code for the SunFire MMUs.

This work was supported in part by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. In addition, this work was supported 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] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[3] J. B. Chen, A. Borg, and N. Jouppi. A Simulation Based Study of TLB Performance. *Intl. Symp. on Comp. Arch.*, 1992.
[4] D. Clark and J. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Transactions on Computer Systems*, 1985.
[5] H. Huck and H. Hays. Architectural Support for Translation Table Management in Large Address Space Machines. *Intl. Symp. on Computer Architecture*, 1993.
[6] ixbtlabs.com. Platform Benchmarking with RightMark Memory Analyzer: AMD K7/K8 Platforms.
[7] B. Jacob and T. Mudge. Software-Managed Address Translation. *Intl. Symp. on High Performance Computer Architecture*, 1997.
[8] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 1998.
[9] 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.
[10] 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.
[11] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *Intl. Symp. on Computer Architecture*, 2002.
[12] D. Nagle et al. Design Tradeoffs for Software Managed TLBs. *Intl. Symp. on Computer Architecture*, 1993.
[13] X. Qui and M. Dubois. Options for Dynamic Address Translations in COMAs. *Intl. Symp. on Comp. Arch.*, 1998.
[14] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. *ACM Transactions on Modeling and Computer Simulation*, 1995.
[15] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-Based TLB Preloading. *Intl. Symp. on Comp. Arch.*, 2000.
[16] M. Talluri. Use of Superpages and Subblocking in the Address Translation Hierarchy. *PhD Thesis, Dept. of CS, Univ. of Wisc.*, 1995.
[17] 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.
[18] Virtutech. Simics for Multicore Software. 2007.
[19] www.sandpile.org. AMD K8 details.