

Improving Cache Locality for Thread-Level Speculation Systems

by

Stanley Lap Chiu Fung

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Stanley Lap Chiu Fung, 2005

Stanley Lap Chiu Fung
Master of Applied Science, 2005
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

With the advent of chip-multiprocessors (CMPs), *Thread-Level Speculation* (TLS) remains a promising technique for exploiting this highly multithreaded hardware to improve the performance of an individual program. However, with such speculatively-parallel execution the cache locality once enjoyed by the original uniprocessor execution is significantly disrupted: for TLS execution on a four-processor CMP, we find that the data-cache miss rates are nearly four-times those of the uniprocessor case, even though TLS execution utilizes four private data caches.

We break down the TLS cache locality problem into instruction and data cache, execution stages, and parallel access patterns, and propose methods to improve cache locality in each of these areas. We find that for parallel regions across 13 SPECint applications our simple and low-cost techniques reduce data-cache misses by 38.2%, improve performance by 12.8%, and significantly improve scalability—further enhancing the feasibility of TLS as a way to capitalize on future CMPs.

Acknowledgements

First and foremost, I would like to extend my thanks to my supervisor, Greg Steffan, for providing the guidance and support I so much needed in pursuit of this degree. His thoughtful and insightful feedback has improved the quality of my research and my writing. These two years in graduate school have been a wonderful learning experience. I would also like to thank Cristiana Amza, Andreas Moshovos and Lacra Pavel for being on my thesis committee.

My family has been a continual support throughout the years and deserve most of the credit. My parents and my sister Joyce have always been encouraging over the phone and internet. I thank Stella for her constant understanding and love.

Contents

Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Thread Level Speculation Basics	2
1.2 The TLS Cache Locality Problem	4
1.3 Research Goals	6
1.4 Overview	6
2 Thread Level Speculation	8
2.1 Software Only Approaches	8
2.1.1 The LRPD Test	9
2.2 Hardware Only Approaches	10
2.2.1 The Speculative Multithreaded Processor	10
2.2.2 The Dynamic Multithreading Processor	11
2.3 Hybrid Approaches	12
2.3.1 The Multiscalar Architecture	12
2.3.2 The Hydra Chip-Multiprocessor	13
2.3.3 The Stampede Approach	14

3	Experimental Framework	18
3.1	The STAMPede TLS Compiler	18
3.2	Simulation Environment	19
3.3	Benchmark Applications	21
3.4	Summary	23
4	The TLS Cache Locality Problem:	
	A Closer Look	24
4.1	Related Work	24
4.2	Evidence of Disrupted Locality	29
4.3	Breakdown of the Cache Locality Problem	31
4.4	Private Cache and Shared-Cache Architectures	32
4.5	Execution Stages	33
4.6	Data and Instruction Cache	36
4.7	Data Cache Miss Patterns in Parallel Regions	37
4.8	Summary	44
5	Techniques for Improving TLS Cache Locality	45
5.1	Scheduling the Sequential Region	45
5.2	Exploiting Read-Based Sharing Patterns	48
	5.2.1 Broadcasting for all Load Misses	49
	5.2.2 Throttling Broadcast	51
5.3	Exploiting Write-Based Sharing Patterns	53
5.4	Exploiting Strided Miss Patterns	60
5.5	Summary	62
6	Combining the Techniques	64

6.1	Performance Impact of Combining all Techniques	64
6.2	Impact of Re-selecting Parallel Regions	70
6.3	Impact on Scalability	71
7	Conclusions	74
7.1	Contributions	74
7.2	Future Work	76
	Bibliography	78

List of Figures

1.1	TLS program execution, when the original sequential program (S) is divided into two epochs (E1 and E2).	2
1.2	Example of broken locality in the speculative execution of the iterations of a simple for-loop.	5
2.1	Using cache coherence to detect a RAW dependence violation.	15
4.1	The TLS cache locality problem: (a) comparing data cache miss rates for the original sequential execution and the speculatively-parallel TLS execution; (b) the fraction of data cache misses during parallel regions where the missing cache line is currently resident in another processor's data cache (we call these <i>locality misses</i>).	30
4.2	Our investigation of the cache locality problem.	31
4.3	Private and shared cache CMP architectures	32
4.4	Execution stages exhibited by a TLS program.	34
4.5	Impact of duplicating the entire sequential data cache to all private caches at the beginning of parallel regions.	35
4.6	The impact of ideal instruction and data caches where misses have no latency on TLS execution. I is the ideal instruction cache model; D is the ideal data cache model; I/D is both the ideal instruction and data cache model.	37

4.7	Read-based sharing misses: <i>full-broadcast</i> involves reads to the same cache line from all four processors, while <i>partial-broadcast</i> only requires two or three processors to read the same data.	39
4.8	Write-based sharing misses: <i>producer-consumer</i> involves a producer processor that writes a data, followed by one or more consumer processors read the same data. Multiple processors writing to the same cache line are considered to be <i>migratory pattern</i> . These two patterns are the most common ones in write-based sharing, while there are other cases.	39
4.9	<i>Strided pattern</i> that occurs over different processors.	40
4.10	Other access patterns: when there is no other access from any processor to the same cache line within the window, we consider those misses as <i>random accesses</i> ; <i>same dcache accesses</i> are consecutive misses to the same cache line from the same processor, these are mainly conflict misses.	40
4.11	Miss pattern breakdown averaged across all benchmarks with varies window size (1000 - 16000 cycles).	41
4.12	Data cache miss patterns within parallel regions.	42
5.1	TLS execution with a floating and fixed sequential processor. In (a), the processor which executed the last speculative thread of the parallel region goes on to execute the sequential region. In (b), one processor (P0) is elected to execute all sequential regions.	46
5.2	Performance impact of a fixed sequential processor relative to floating.	47
5.3	Impact of broadcasting all load misses on parallel regions.	50
5.4	Impact of throttling broadcasting with profiling on parallel regions, relative to the full broadcast scheme.	52

5.5	A mechanism for detecting cache lines involved in write-based sharing, and aggressively invalidating and forwarding them to the next cache. Each processor maintains a <i>recent store table</i> , <i>push required buffer</i> , and an <i>invalidation PC list</i>	54
5.6	Impact of our technique for exploiting write-based sharing patterns on parallel regions.	56
5.7	Performance impact of varying the size of the <i>recent store table</i> , <i>invalidation PC list</i> and <i>push required buffer</i> from 4 to 32 entries with a bus interconnect.	58
5.8	Impact of stride-based prefetching on parallel regions.	61
6.1	Impact on the number of data cache misses within parallel regions of combining the three techniques, relative to the baseline model: exploiting read-based sharing (<i>RB</i>), write-based sharing (<i>WB</i>), and strided prefetching (<i>ST</i>).	65
6.2	Impact on the performance of parallel regions of combining the three techniques, relative to the baseline model: exploiting read-based sharing (<i>RB</i>), write-based sharing (<i>WB</i>), and strided prefetching (<i>ST</i>).	66
6.3	Summary of all our techniques, showing average program speedup of 13 SpecInt benchmarks relative to sequential execution. <i>Float</i> shows the performance of a “floating” sequential processor, <i>Baseline</i> includes a fixed sequential processor, <i>RB</i> exploits read-based sharing, <i>WB</i> exploits write-based sharing, and <i>ST</i> performs strided prefetching.	67
6.4	Impact on data cache misses of all our techniques within parallel regions, showing number of misses relative to the baseline. <i>Baseline</i> includes a fixed sequential processor, <i>RB</i> exploits read-based sharing, <i>WB</i> exploits write-based sharing, and <i>ST</i> performs strided prefetching.	68

6.5	Impact of re-selecting parallel regions after applying our techniques with a bus interconnect, showing program speedup relative to the sequential execution: B is the baseline, B/RW implements both broadcast all load misses and aggressive writeback techniques, and R/RW reselects parallel regions after applying those techniques.	71
6.6	Impact of our techniques for improved locality on the scalability of parallel regions, as we vary the number of processors from 2 to 8. The improvement is most pronounced for BZIP2_COMP and VPR_PLACE, but is also significant on average across all benchmarks.	72

List of Tables

3.1	Simulation parameters.	20
3.2	Benchmark descriptions and inputs used.	22
3.3	Benchmark statistics (based on a 4-processor CMP with baseline TLS support).	23

Chapter 1

Introduction

The chip multiprocessor revolution has begun: all major processor vendors have announced chip multiprocessor (CMP) designs, including Intel’s “Smithfield” (dual-core Pentium IV’s), AMD’s Opteron (dual-core), IBM’s Power 4/5 (dual-core), and Sun Microsystems’ Niagara (8 cores). While it is relatively straightforward to improve the throughput of a workload using these CMPs, to improve the performance of an individual program it must somehow be parallelized into threads. Parallelizing a program can be done either by hand or by using a parallelizing compiler. Doing it by hand is tedious and is prone to errors. It is also difficult for a parallelizing compiler to parallelize a general purpose program due to ambiguous memory pointers and possible data dependences. One promising possibility for automatically-parallelizing general-purpose programs is *Thread-Level Speculation* (TLS) [2, 3, 6, 12, 17, 18, 25, 29, 33, 35–38, 41] which allows the compiler to create parallel threads even in the presence of ambiguous memory references, relying on the underlying hardware support to detect dependence violations and recover from failed speculation.

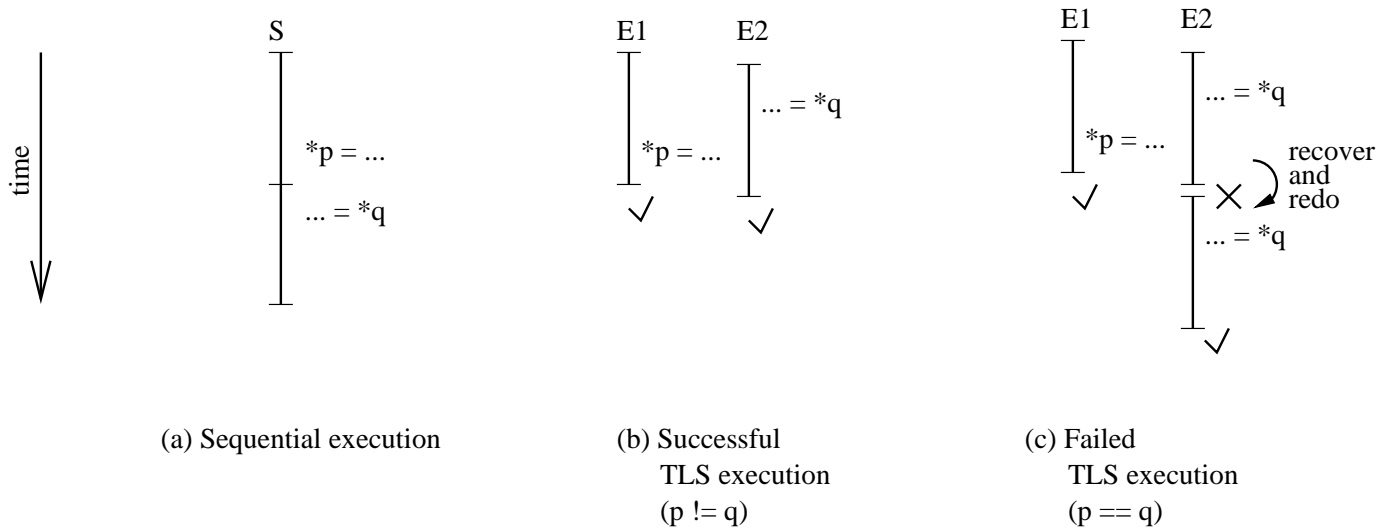


Figure 1.1: TLS program execution, when the original sequential program (S) is divided into two epochs (E1 and E2).

1.1 Thread Level Speculation Basics

Thread-Level Speculation (TLS) allows the compiler to automatically parallelize general-purpose programs by supporting parallel execution of threads, even in the presence of statically ambiguous data dependences. The underlying hardware ensures that speculative threads do not violate any dynamic data dependence and buffers the speculative data until it is safe to be committed. When a dependence violation occurs, all the speculative data will be invalidated and the violated threads will be re-executed with correct data. Figure 1.1 demonstrates a simple TLS example. The sequential program, S in Figure 1.1(a), is carved into two chunks of work, labeled as E1 and E2 in Figure 1.1(b) and (c). They are then executed speculatively in parallel, even though the addresses of the pointers p and q are not known until runtime. Hence, TLS allows us to extract any potentially available thread-level parallelism.

A read-after-write (true) data dependence happens when p and q both point to the same memory location. Since the store produces data that will be read by the dependent load, these store and load instructions need to be executed in the original sequential program order. In Figure 1.1(b), p and q do not point to the same location. Speculation is successful and both speculative threads can commit their results at the end of execution. However, as shown in Figure 1.1(c), when both pointers point to the same location, a true data dependence between the two speculative threads is detected. Speculation fails because it leads to an out-of-order execution of the dependent load-store, which violates the sequential program order. The offending thread is then violated and re-executed with proper data.

There are various proposed thread-level speculation systems that aim at exploiting thread-level parallelism in sequential programs by using parallel speculative threads. Among all different types of thread-level speculation systems that have been proposed, there are three common key components: (i) breaking sequential program into speculative threads—this task has to be done efficiently to maximize thread-level parallelism and minimize the overhead; (ii) tracking data dependences—since the threads are executed speculatively in parallel, the system has to be able to determine whether the speculation is successful; (iii) recovering from failed speculation—in the case when speculation has failed, the system has to repair the incorrect architectural states and data, and discard the speculative work. Different TLS systems have different implementations of these three components. Chapter 2 describes several previously-proposed TLS systems in detail. In this thesis, we focus on one particular flavour of TLS: the STAMPede [37,38] approach.

1.2 The TLS Cache Locality Problem

Under TLS, a sequential application is divided into speculative threads, which are in turn executed on the processors of the underlying CMP. In a typical CMP, the processors will share a unified second-level cache, but will each have private first-level data and instruction caches. While the original sequential program would have executed on a single processor using only one data and instruction cache, with TLS that program is divided across several processors and will therefore use several data and instruction caches. Although the main motivation of TLS is to allow a single program to exploit distributed resources, spreading the memory accesses of a program across multiple caches can dramatically disrupt the cache locality enjoyed by the original sequential execution.

Figure 1.2 shows a simple example of broken locality during parallel execution. Inside the `for` loop, as shown in Figure 1.2(a), variable `x` loads in a new value from an array in every iteration. Since the cache line size is 32 bytes and each array element is 8 bytes, four array elements share the same cache line. Figure 1.2(b) shows the sequential execution of the loop. There is only one miss every four accesses. When a miss occurs, the entire 32-byte cache line is loaded into the cache and each line contains four array elements: the first one is used by the current iteration and the rest are used by the next three iterations. Therefore, the next three accesses to the array are cache hits since the array elements being accessed are already in the cache. On the other hand, when the loop is executed in parallel on four processors, all the cache accesses will result in misses, as shown in Figure 1.2(c). Since each processor only loads the data into its own private data cache, the system no longer benefits from locality.

To quantify the scope of the cache locality problem for TLS, we compare the data cache miss rate for each sequential SPECint application with that of the speculatively-parallel version on a

```

unsigned long long array[N];
...
for (index = 0; index < N; index++){
    x = array[index];
    ...
}

```

(a) pseudo code

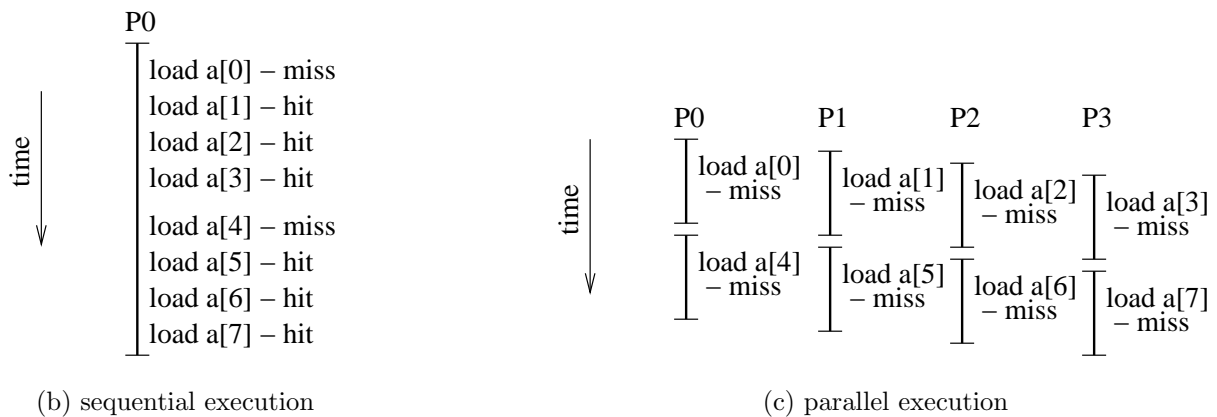


Figure 1.2: Example of broken locality in the speculative execution of the iterations of a simple for-loop.

4-processor CMP with 32KB first-level data caches (the details of our experimental framework are available in Chapter 3). Overall, the average data cache miss rate for the TLS versions is nearly four times that of the original sequential versions. One metric that demonstrates the problem is *locality misses*: a locality miss is a cache miss for a line that is currently resident in another cache. *Locality misses* account for 61.1% of the cache misses during TLS execution, indicating that locality has indeed been disrupted.

1.3 Research Goals

The goal of this research is to provide hardware techniques to improve cache locality and hence performance of TLS systems. We first perform a thorough classification of the cache locality problem for TLS. Based on this classification, we propose hardware optimizations to address each class of locality problem. Our techniques must be both:

- 1) **Cost-effective:**—we want to minimize the amount of extra hardware and modification to the existing hardware. We focus on techniques with low cost and avoid any optimization that requires complex hardware support.
- 2) **Scalable:**—we want our techniques to scale well to CMPs with up to eight processors. We focus on techniques that are distributed and do not require any centralized mechanisms.

We evaluate our techniques with detailed simulation on a wide variety of benchmark applications, and demonstrate the corresponding performance improvements of on an existing TLS system.

1.4 Overview

The remainder of this dissertation is organized as follows. In Chapter 3 we describe the underlying support for TLS on which we base our evaluation, including benchmark applications and simulation infrastructure. In Chapter 4 we classify the TLS execution stages and the resulting cache locality problems, and identify the most important categories of cache misses to address. Guided by this classification, in Chapter 5.1 we investigate options for scheduling the stages of execution, and in Chapters 5.2, 5.3, and 5.4 we evaluate techniques for exploiting read-only and write-based sharing patterns, and strided miss patterns respectively. In Chapter 6 we quantify

the impact of the combination of all techniques, demonstrate the importance of reselecting parallel regions, and show the impact on scalability. Finally we conclude in Chapter 7.

Chapter 2

Thread Level Speculation

This chapter presents the background material and related work in the area of thread-level speculation that is necessary for the rest of the dissertation. Speculative thread-level parallelism has recently been proposed to boost performance in addition to instruction-level parallelism. Thread-level speculation has significant potential in speeding up applications where independent threads are difficult to find. The main idea is to split the sequential program into threads, which are then executed speculatively in parallel. Data dependences among speculative threads are checked at runtime, and when speculation fails the system recovers and re-executes the violated speculative work. There are many different TLS systems that have been proposed. Based on their implementations, we group them into three categories: software-only, hardware-only and hybrids that use both hardware and software.

2.1 Software Only Approaches

Most of the early work on software-only speculation [23, 34] involves a combination of compile-time and runtime parallelization on loops that iterate over arrays. The main advantage is

that these approaches do not require extra hardware beyond generic multiprocessor support. However, most of the software approaches are limited to arrays and have very large overhead. Since we are concerned with improving TLS for hardware-based approaches, here we only briefly describe one of the earliest works on software speculation.

2.1.1 The LRPD Test

Rauchwerger *et al* [34] proposed a complete software approach for speculatively parallelizing loops in the presence of ambiguous data dependences for array accesses. The target loop is parallelized as if it were completely parallelizable. During speculative execution, shadow arrays are used to keep track of all read and write accesses of the shared variables. These arrays are examined at the end of the loop execution to ensure that the memory accesses of different iterations did not overlap. When speculation fails, the entire loop is re-executed sequentially. There are a few shortcomings to this pure software speculation scheme. First, it requires the creation of shadow arrays for all shared data. These shadow arrays take up large amount of storage. Second, this approach is only applicable to loops with arrays, but not applicable to modern pointer-based applications. Third, data dependences are only checked after the entire loop finishes execution. There is no mechanism to detect failed speculation while the loop is running. Fourth, if the loop is not entirely parallelizable, LRPD test is not able to extract partial parallelism and the loop is re-executed sequentially. However, since this approach targets array-based loops, traditional compiler optimizations to improve cache locality may be applicable, although such optimizations are beyond the scope of this thesis.

2.2 Hardware Only Approaches

Several microarchitectures [2,29,33] have been proposed for supporting thread-level speculation. The hardware provides the mechanisms to decide where and when to create threads and ensure program correctness. The major advantages of these approaches are: i) they have the ability to handle pointer-based applications and; ii) programs do not need to be re-compiled to take advantage of the hardware speculation support. However, without high-level knowledge of the program structure, it is difficult for the hardware to find to right spot to create threads for speculation. Although these hardware approaches have no software overheads, they are generally more complex compared to approaches that exploit software, since all the optimizations and transformations must be implemented in hardware, including thread selection, runtime dependence checks and failed speculation recovery. In this section we describe two major hardware-only TLS projects.

2.2.1 The Speculative Multithreaded Processor

Speculative Multithreaded Processor [29, 33] uses hardware to identify loops at runtime and creates speculative threads of successive iterations of the same loop. The identified loop is not necessarily an innermost loop and the loop iterations do not need to be independent. The sequential program is partitioned into threads dynamically by the hardware and does not require any compiler intervention. The processor consists of several thread-units that execute multiple iterations of the loop simultaneously. The thread-units are arranged in a ring topology and threads are assigned to the thread-units following the original sequential execution order. Each thread-unit has its own physical register file and register map table. When a speculative thread is created, its local register file and register map table are copied from its predecessor. The instruction and data caches are shared among all thread-units. Data dependences through

memory are tracked with a *multi-value cache*. This centralized cache structure stores for each address as many different data words as the number of thread-units. Misspeculation is detected by broadcasting each store's effective address to the succeeding threads and each thread checks in its load/store queue for any matching load. If there is a load that matches the broadcasted store address, the offending thread and all its dependent threads are re-executed. The cache hierarchy of the Speculative Multithreaded Processor is the same as a sequential processor, where all the speculative threads share the same set of caches. Therefore, cache locality is naturally preserved among different threads. However, the shared-cache approach requires support for storing multiple versions of the same cache line in the shared-cache, which involves complex hardware and increases the latency of cache accesses.

2.2.2 The Dynamic Multithreading Processor

The Dynamic Multithreading Processor (DMT) [2] has hardware to create threads at procedure and loop boundaries and executes the threads speculatively on a simultaneous multithreaded pipeline. Each thread has its own PC, rename tables, trace buffer and load/store queues. The control logic keeps a list of the thread order in the program and the start PC of each thread. A thread stops when its PC reaches the start of the next thread in the order list. A new thread uses the register context from its parent thread as input for speculative execution. When it retires, the speculative register inputs are compared to the final values at the end of the prior thread. Speculation fails if these two sets of register values are different. The load/store queues are used to track memory data dependences; any out-of-order execution of a dependent load/store pair results in failed speculation. Failed speculation recovery is organized using a large trace buffer that stores all speculative instructions and data. DMT supports partial recovery: only those threads that violated the original data dependences are re-executed, and instructions and data

in the trace buffer that are not affected by the mispredicted data are re-used. Similar to the Speculative Multithreaded Processor, the DMT processor uses only one set of caches and hence locality is preserved with speculative execution.

2.3 Hybrid Approaches

Hybrid TLS systems are more popular than pure software or hardware approaches, since they can take advantage of both the compiler and hardware. Thread selection can be done easier and better by the compiler with the knowledge of program structure. Transformation and optimization can also be done by the compiler to improve parallelism, while the hardware takes care of performance critical operations such as runtime data dependence checks and recovery from failed speculation. This section describes three hybrid TLS system: (1) The Multiscalar—the earliest hybrid TLS system; (2) The Hydra; (3) STAMPede—the TLS system on which this dissertation is based.

2.3.1 The Multiscalar Architecture

In the Multiscalar architecture [14, 16, 41], the compiler is responsible for breaking up a single-threaded program into small tasks, which might possibly have data or control dependences between them. These small tasks are then distributed to a collection of parallel processing units under the control of a centralized hardware sequencer. Each of these processing units fetches and executes the assigned tasks simultaneously, exploiting thread-level parallelism. The processing units are arranged in a ring, of which the head processing unit is executing the earliest thread while the tail processing unit is executing the latest thread. This ring arrangement makes tracking data dependences easier as the tasks are ordered by the processing units in the ring. The hardware sequencer keeps track of the control flow of the program and when control

speculation fails, the violated thread and all the later speculative threads are squashed. To support data speculation, the Multiscalar architecture includes the Address Resolution Buffer (ARB) [13] which is used to store speculative data and to track data dependences. All memory accesses issued by the processing units are filtered by the ARB before reaching the cache system. The ARB buffers speculative modifications and commits them to the cache only when these modifications become non-speculative. The ARB increases the latency for all memory accesses and can easily become the bottleneck of the system. The Speculative Versioning Cache (SVC) [17] is a follow-up work on the ARB. The SVC operates on distributed caches with a snooping bus-based coherence scheme, eliminating the latency and bandwidth problems of the ARB. However, it results in a lower cache hit-rate because data is spread across multiple private caches. Our hardware locality techniques for improving cache locality would be applicable to the SVC implementation of the Multiscalar as well.

2.3.2 The Hydra Chip-Multiprocessor

The speculation support on the Hydra chip multiprocessor [18, 19] is the most similar to the Stampede approach to TLS, which we will describe in the next section. Both schemes use a combined hardware/software approach. In the Hydra framework, the compiler marks the potential parallel loops and the hardware distributes the iterations among the processor cores. The hardware also spawns threads when there is a subroutine call. The original processor executes the subroutine call while another processor executes the code following the subroutine call speculatively. To track data dependences and recover from violations, additional bits are added to each cache line tag to record speculation states and there is a secondary cache buffer which buffers all writes to the shared second-level cache during speculation. In the Hydra architecture each processor has its own private cache, hence data is spread among all private caches, reducing locality. Therefore, the Hydra would also benefit from our cache locality optimizations.

2.3.3 The Stampede Approach

We describe Stampede TLS [3, 4, 37–39] in more detail since it is the foundation of this work. Stampede TLS uses both software and hardware to perform speculation. It relies on the compiler to decide which part of the program to speculatively parallelize, and uses hardware support to track data dependences and recover from failed speculation.

The STAMPede compiler divides the program into speculatively parallel units of work called *epochs* and inserts special TLS instructions to tell the hardware when to execute speculatively. During speculative execution, each epoch *spawns* the next epoch through a lightweight fork mechanism. The spawn mechanism forwards initial parameters and the program counter to the appropriate processor. To track data dependences, each epoch is timestamped with an epoch number, which indicates the original sequential ordering of the program. Epochs can only commit their result when it becomes non-speculative. The commit order is tracked by passing the *homefree token*, which indicates that all previous epochs have already committed their results to memory. After an epoch commits all its speculative modifications, it passes the *homefree token* to the next epoch. Upon receiving the homefree token, an epoch becomes non-speculative since all the earlier epochs have already completed their speculative work and committed the results back to memory. At this point, the speculative cache lines of this non-speculative epoch can transition to the corresponding non-speculative states and speculation is considered to be successful for this epoch. On the other hand, when speculation fails all the speculatively modified lines are invalidated and the speculatively loaded lines transition to non-speculative states. After the recovery is done, the violated epoch is re-executed with proper data.

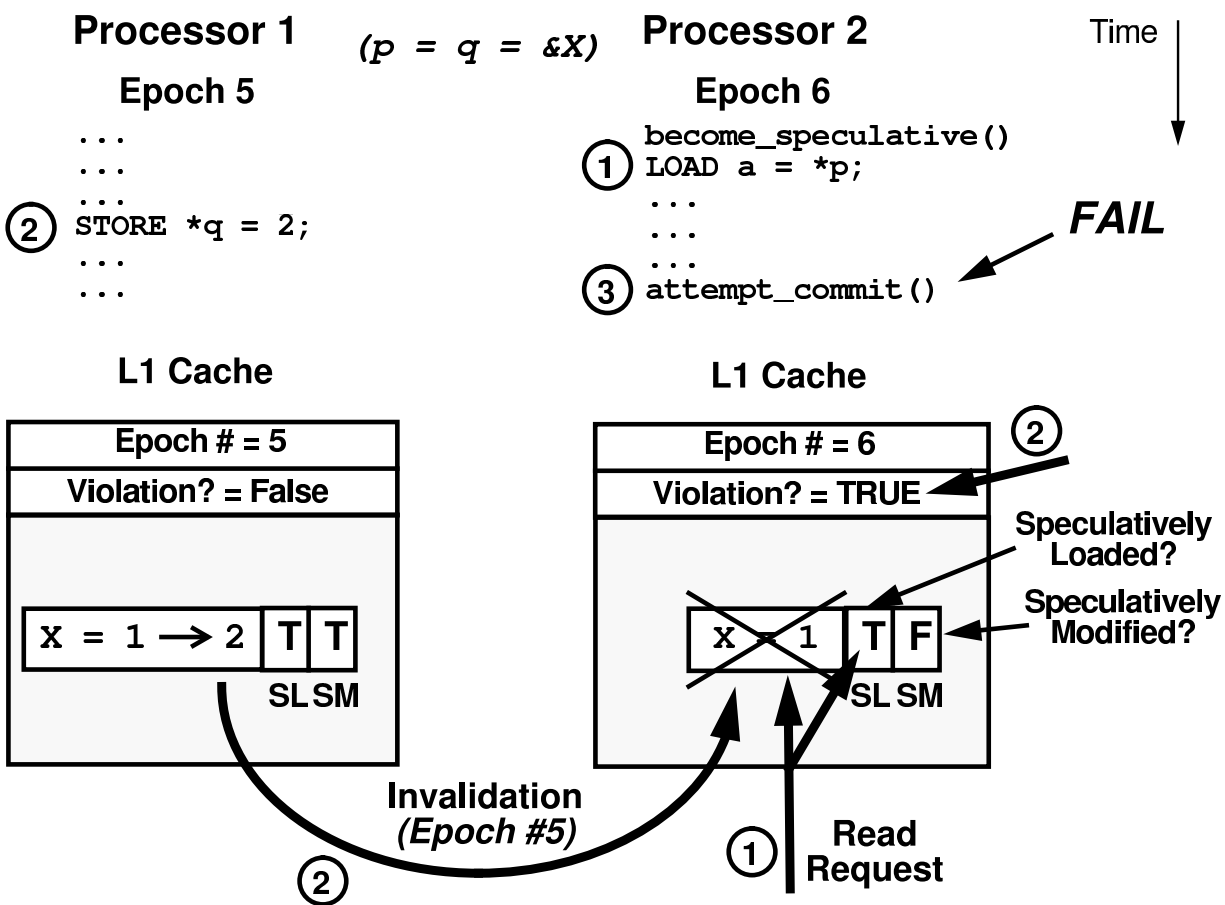


Figure 2.1: Using cache coherence to detect a RAW dependence violation.

Stampede TLS does not use any special buffer for storing speculative data. Instead, both speculative and permanently committed data are stored in the cache system and an extended invalidation based cache coherence scheme is used to track the speculative state of the cache lines [37]. To illustrate the basic idea behind the STAMPede hardware coherence scheme, we show an example of how it detects a read-after-write (RAW) dependence violation in Figure 2.1. Recall that a speculative load violates a RAW dependence if its memory location is subsequently modified by another epoch such that the store should have preceded the load in the original sequential program. To track speculative data, the state of each cache line is augmented to indicate whether the cache line has been speculatively loaded (SL) and/or speculatively modified (SM). For each cache, the *epoch number* indicates the sequential ordering of that epoch with respect to all other epochs, and a flag indicates whether a data dependence violation has occurred. In the example, *epoch 6* performs a speculative load, so the corresponding cache line is marked as speculatively loaded. *Epoch 5* then stores to that same cache line, generating an invalidation containing its epoch number. When the invalidation is received, three things must be true for this to be a RAW dependence violation. First, the target cache line of the invalidation must be present in the cache. Second, it must be marked as having been speculatively loaded. Third, the epoch number associated with the invalidation must be from a *logically-earlier* epoch. Since all three conditions are true in the example, a RAW dependence has been violated; *epoch 6* is notified by setting the *violation flag*.

To avoid failed speculation caused false sharing, the TLS hardware tracks the speculative state for writes at a per-word level. It allows disjoint portions of a line to be read and written speculatively by different epochs without causing speculation to fail. This scheme reduces the number of data dependence violations due to false sharing significantly and has great impact on performance.

More details of the simulation framework of STAMPede TLS, and the benchmarks that we use for evaluating our locality techniques are discussed in the next Chapter.

Chapter 3

Experimental Framework

Before we investigate the cache locality problem, in this chapter we describe the infrastructure for evaluating TLS that this work is based on, including both hardware simulation and compiler support. While this study is within the context of a particular TLS implementation, the techniques that we suggest for improving cache locality are applicable to other TLS systems as well [17, 18, 38], such as discussed in Chapter 2.

3.1 The STAMPede TLS Compiler

STAMPede TLS relies on the compiler to define where to speculate. Performing this task without the use of detailed profile information is an open research problem. For the evaluations in this dissertation, the compiler uses profile information to decide which loops in a program to speculatively parallelize.

The compiler infrastructure developed at CMU [3, 4] is based on the Stanford SUIF 1.3 compiler system. The benchmarks are first converted to SUIF format by using the SUIF C

compiler. At this stage, the TLS compiler uses profile information to decide which loops in a program to speculatively parallelize. First, every loop in every benchmark application are measured by instrumenting the start and end of each potential speculative region (loop) and epoch (iteration). Second, those loops that are unlikely to contribute to improved performance are removed from consideration. Third, each loop is unrolled by 1, 2, 4 and 8 to generate several versions of each benchmark to measure. Then the performance of each loop and unrolling are measured when running speculatively in parallel with the baseline hardware support for TLS. The set of loops and unrolling that maximize overall performance of the benchmark are selected.

Once speculative regions are selected, the TLS compiler inserts TLS instructions to tell the hardware where to speculate. Optimization for data forwarding and synchronization is also performed at this stage [39]. The compiler identifies the computation chain of the data to be synchronized, and uses dataflow analysis to schedule the synchronization in the earliest safe location. Synchronization allows us to speculatively parallelize the program even though there are known data dependences. At the end, the compiler outputs C code that has embedded TLS instructions, and the resulting C code is then compiled with `gcc` to produce a MIPS executable.

3.2 Simulation Environment

We evaluate our support for TLS through detailed simulation. Our simulator models 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [44], but modernized to have a 128-entry reorder buffer. We simulate systems with multiple processing cores, where each has its own physically private data and instruction caches, connected to a unified second level cache by either a bus or a crossbar interconnect which are modeled as follows: The bus is shared among all processors and only one processor can use the bus at anytime. When

Table 3.1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Interconnect	8B per cycle
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

the bus is occupied, requests are queued in a buffer and are served on a first-come-first-serve basis. With the crossbar, each processor may connect directly to one bank of the unified-cache at a time, and contending requests for a bank or processor are queued. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 3.1.

3.3 Benchmark Applications

We report results for all of the SPECint95 and SPECint2000 benchmarks [9] except for the following: 252.EON, which is written in C++ and therefore not handled by our compiler; 126.GCC, which is similar to 176.GCC; 147.VORTEX, which is identical to 255.VORTEX; and 134.PERL, which is similar to 253.PERLBMK; for 129.COMPRESS, 254.GAP, 164.GZIP, and 300.TWOLF, the region selection algorithm has opted to select no regions at all and we exclude them from our main study (although we revisit them later in Chapter 6.2). We do not claim that there is no speculative parallelism available in these applications, but that they at least require more advanced compiler and/or support for TLS than we investigate in this dissertation. A brief description of each benchmark and the input data set used is given in Table 3.3—we use the *ref* input set for every benchmark. To maintain reasonable simulation time, we truncate the execution of all appropriate benchmarks by fast-forwarding the initialization portion of execution and simulating up to the first billion instructions, beginning simulation with a “warmed-up” memory system loaded from a pre-saved snapshot. Since the sequential and TLS versions of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code so that the executions are comparable.

Table 3.2: Benchmark descriptions and inputs used.

Benchmark	Description	Input	
SPEC2000	BZIP2_COMP	compression	input.source from ref, comp phase
	CRAFTY	chess board solver	ref
	GCC	compiler	expr.i from ref
	MCF	combinatorial optimization	ref
	PARSER	natural language parsing	ref
	PERLBMK	perl interpreter	diffmail.pl from ref
	VORTEX	OO database	bendian1.raw from ref
	VPR_PLACE	place and route for FPGAs	place portion of ref input
	VPR_ROUTE	place and route for FPGAs	route portion of ref input
SPEC95	GO	game playing, AI, plays against itself	9stone21.in from ref
	IJPEG	image processing	vigo.ppm from ref
	LI	lisp interpreter	ref
	M88KSIM	microprocessor simulator	ref

Table 3.3 shows some detail on the benchmarks studied. Note that we have separated the compression phase of BZIP2, and the place and route phases of VPR. The fraction of execution which has been parallelized ranges from 4% (VORTEX) to 97% (MCF), and averages 35%, through a number of different parallel regions per application. Many of these applications spend most of the execution time in regions other than loops, for example in recursion. The speedups of baseline TLS are reported with respect to the original executable, without any TLS instructions or overhead. Except for PERLBMK, all the benchmarks have improved performance in parallel regions with the TLS baseline hardware, over sequential execution. The speedup of speculatively-parallelized regions, ranging from 6% to 205%. Looking at program performance, IJPEG and M88KSIM are 77% and 56% faster respectively, and four other applications improve by at least 8%. Four other applications show more modest improvement, while VORTEX and PERLBMK perform slightly worse than the original sequential version. This is mainly because the performance of non-parallelized regions is reduced for the TLS versions, due to inserted TLS instructions (for code transformation and compiler optimizations) and decreased cache locality

Table 3.3: Benchmark statistics (based on a 4-processor CMP with baseline TLS support).

Benchmark	Portion of Dynamic Execution Parallelized (Coverage)	Baseline Parallel Region Speedup	Baseline Program Speedup
BZIP2_COMP	28%	1.22	1.08
CRAFTY	9%	1.25	1.00
GCC	11%	1.48	1.01
GO	17%	1.30	1.03
IJPEG	84%	2.12	1.77
LI	8%	1.06	1.00
M88KSIM	57%	3.05	1.56
MCF	97%	1.16	1.15
PARSER	7%	1.31	1.02
PERLBMK	10%	0.97	0.99
VORTEX	4%	1.18	0.99
VPR_PLACE	76%	1.19	1.10
VPR_ROUTE	48%	1.06	1.02
average	35.1%	1.41	1.13

(data is spread to multiple caches). On average across all applications we achieve a program speedup of 13%.

3.4 Summary

In this chapter, we provided the details of our compiler, simulator and benchmarks. Our simulation infrastructure for the STAMPede TLS system is detailed and realistic. The TLS baseline hardware achieves a significant parallel region speedup, but a more modest program speedup due to coverage. In the next chapter, we begin our investigation into improving cache locality for TLS.

Chapter 4

The TLS Cache Locality Problem: A Closer Look

The previous chapters introduced the STAMPede approach to TLS and our simulation environment. In this chapter, we investigate the cache locality problem for TLS to understand more about this problem before proposing any solution. First, we describe some related cache locality work, although most of the previous related work focused on generic multiprocessor systems without the support for thread-level speculation. Next, we show evidence of broken cache locality during TLS execution by comparing the cache behaviour of sequential programs and their speculatively-parallelized counterparts. Then, we systematically break down the problem and address each category individually.

4.1 Related Work

To the best of our knowledge, there has been no significant work on improving cache locality for TLS execution of general-purpose programs. In contrast, there has been a great deal of

work on improving locality for array-based, scientific programs [8, 43], in particular employing software transformations that adjust access patterns and data layout to improve locality. Since TLS involves speculative execution beyond just array accesses, traditional compiler transformations are not applicable to TLS. There is previous research on improving cache locality for multiprocessor systems [1, 10, 22], however there are two major differences between TLS systems and traditional multiprocessor systems: First, most TLS systems are proposed for small scale chip-multiprocessors which have less than eight processing units and hence communication is within a single chip. In contrast, traditional multiprocessor systems usually focus on building larger systems and involve processors on multiple chips connected by off-chip interconnects. Second, TLS aims at exploiting the thread-level parallelism on general-purpose programs, which are difficult to be statically parallelized by parallelizing compilers due to their extensive use of pointers. However, large-scale multiprocessor systems aim at improving execution time of those highly-parallelizable scientific applications. Therefore cache locality techniques that were proposed for traditional multiprocessor systems are not applicable to TLS directly. Furthermore, in this thesis we consider only hardware techniques for exploiting parallel access patterns to reduce cache misses and improve locality, although interesting future work would further consider potential compiler techniques—but that is beyond the scope of this work.

Several hardware techniques for improving locality in multiprocessor systems have been proposed, including works on detecting data sharing patterns [1, 10, 22] and self-invalidating cache lines [26, 27]. In [22], Kaxiras *et al.* showed that widely shared data in parallel programs occurs frequently and there is considerable performance benefit in providing hardware support to reduce access time for those widely shared data. Three schemes for detecting which data is widely shared were evaluated. The first scheme observes all coherence requests in the interconnect; the second scheme uses the directory to discover shared data; the third scheme is based on the load instruction program-counter (PC) value. If a load instruction accessed widely-shared data in the

past, then it is likely to access widely shared data in the future. Whether a load accessed widely shared data is judged by its miss latency: large miss latency is interpreted as an access to widely shared data. This performs the best of all three proposed shared data detection schemes. However, the major drawback is that all the PCs of the load instructions have to be known outside the processor core, thus requiring each miss request to carry the PC of the miss instruction. In [1], Abdel-Shafi *et al.* evaluated the performance of two remote-write operations: *WriteThrough* and *WriteSend*. With *WriteThrough*, the producer processor updates the memory at the same time as it writes the data to its cache, while *WriteSend* updates the cache of a specified processor. The remote-writes operations are inserted by hand based on the algorithm proposed by Mowry [31]. This work demonstrated that remote writes provide significant performance benefits in cache-coherence shared memory multiprocessors on scientific benchmarks. In our work, we also found that it is beneficial to use the PC for detecting sharing patterns in TLS and write-based sharing misses results in long coherence latency. We designed a hardware prediction mechanism for detecting write-based sharing patterns that does not require carrying PC values in cache miss requests. Since our scheme is implemented completely in hardware, it does not require any modification to the program nor any compile-time support. Our prediction scheme and its performance impact on general-purpose program under TLS are presented in Section 5.3.

A last-touch predictor [26] predicts the last access to a memory block by one processor before the block is accessed and subsequently invalidated by another processor. It aims at reducing the coherence overhead for invalidating remote cache lines in a distributed shared memory system. Similar to last-touch prediction, dead-block prediction [27] is a technique that predicts when a cache block is *dead*, meaning that there will not be any future access to this cache line before it is evicted from the cache. When a cache line is predicted to be dead, the prefetcher is triggered to replace the dead cache line with a potentially useful prefetched line. It enhances the timeliness of prefetching since the prefetching mechanism is started as early as the replaced line is predicted

to be dead, usually soon after the last access to that line. These self-invalidation techniques are proven to perform well for a uniprocessor or even generic multiprocessor system, however TLS cache coherence schemes require speculative cache lines to remain in the private caches until commit time, and hence self-invalidation can only be performed after each speculative thread has completed. To improve the locality of write-based sharing accesses in Section 5.3, we propose a technique that involves self-invalidating cache blocks for TLS.

Memory latency can be tolerated through prefetching [5, 11, 15, 21, 30], however most of the previous prefetching work was developed for sequential execution on a single processor. While prefetching can be supported in both hardware or software, we will briefly discuss a few pure hardware prefetching schemes as well as schemes that make use of the compiler. Most hardware prefetching mechanisms work by recording the history of program counter values and memory addresses. A stride prefetcher [5, 11, 15] has proven to work well for loads whose addresses follow an arithmetic progression. However, most general purpose programs use dynamic memory allocation and linked data structures, which lead to irregular memory access patterns. An alternative mechanism is to try to find a correlation between miss addresses in an attempt to predict future miss addresses [20, 21]. The Markov prefetcher [21] is an example of such correlation prefetching. Other prefetchers rely on the compiler to provide prefetch hints [42, 45]. These software based techniques are good at predicting misses due to pointer de-references. Most of the proposed prefetching schemes prefetch from off-chip memory to on-chip cache, while our locality schemes focus on improving locality within a chip for a CMP. These schemes for prefetching from off chip should be complementary with our on-chip locality techniques for TLS systems.

In most CMPs, either the level-two or level-three cache is shared among all processors. During parallel execution when one thread accesses a cache line, this line is brought into the shared cache where all other threads have access to it. This cache sharing property allows TLS itself

to have prefetching effects. Many schemes have been proposed to speculatively execute *helper threads* which prefetch or speculatively precompute for a main thread [24, 28, 46]. The main idea of speculative precomputation is to use the idle hardware contexts on a simultaneous multithreaded architecture (SMT) to execute helper threads. These threads run ahead of the main thread and trigger future cache misses in advance. Previous research has shown that speculative precomputation can speed up memory-intensive sequential programs significantly on SMT architectures. However, since each processor core on a CMP has its own private level-one data cache while sharing a level-two cache, speculative threads can only prefetch data up to the shared cache but not the level-one private caches. Therefore speculative precomputation prefetching schemes are not as effective on CMPs as for an SMT processor. In this work, we focus on improving cache locality for CMPs with private caches, and our techniques could potentially help speculative precomputation on CMPs as well.

Most closely-related to this work, Brown *et al.* [7] proposed two schemes to improve the performance of speculative precomputation on a CMP. The first scheme is broadcasting load misses: when any of the processors has a load miss, the cache line will be loaded to the private cache of all other processor cores, as well as the shared caches. This scheme artificially models a shared-cache architecture, like the SMT architecture. We find that similar support works well to improve locality and performance for TLS. The second proposed scheme requires the processors to snoop on the data bus: when other peer cores miss a load and issue a request to the shared cache, each processor has to check if the data is in its private cache. If any processor has the data ready to share, it sends the data to the requesting processor. The shortcoming of this peer-sharing scheme is that each processor core essentially has to do a look-up in its private cache whenever a load miss occurs in any of the cores, therefore the number of look-ups increases as there are more processor cores in the system. This increases the pressure on the private caches and can potentially delay other critical cache requests. We use a similar technique to eliminate

read misses in TLS parallel execution, however our technique does not require a look-up in the private caches for each access, and hence it is more scalable.

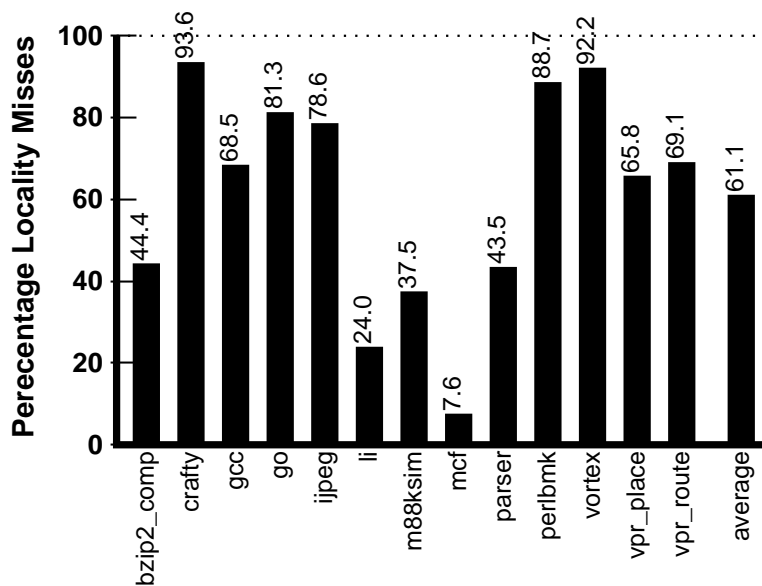
4.2 Evidence of Disrupted Locality

To demonstrate the TLS cache locality problem, in Figure 4.1(a) we compare the data cache miss rate for each sequential SPECint application with that of the speculatively-parallel version on a 4-processor CMP with 32KB first-level data caches and a bus interconnect (the details of our experimental framework are available in Chapter 3). We observe that the miss rate increases significantly in every case. M88KSIM, PERLBMK, and VPR_ROUTE suffer the most, with more than 800% increase in miss rate. There are three benchmarks that have less than 20% increase in miss-rate: LI, PARSER and VORTEX. One reason for the low percentage of locality misses is that the threads tend to share less data compared to other benchmarks; the other reason is that they have relatively low parallel region coverages (see Table 3.3). As shown in Figure 4.1(a), overall the average data cache miss rate for the TLS versions is nearly four times that of the original sequential versions.

As further evidence that the culprit is reduced cache locality, in Figure 4.1(b) we show the fraction of data cache misses during parallel regions for which the missing cache line is currently resident in another processor's data cache. On average 61.1% of all misses are such *locality misses*, indicating that locality has indeed been significantly disrupted. Only MCF has less than 10% locality misses. Although MCF suffers significantly from its poor cache performance, most of the cache misses are also misses to the second-level cache, which are not considered to be locality misses. In five benchmarks, CRAFTY, GO, IJPEG, PERLBMK and VORTEX, more than 75% of cache misses are locality misses. This high percentage of locality misses results in huge

Application	D-Cache Miss Rate		
	Seq.	TLS	Increase
BZIP2_COMP	0.025	0.048	93.4%
CRAFTY	0.015	0.034	129.8%
GCC	0.016	0.028	72.4%
GO	0.014	0.043	209.8%
IJPEG	0.007	0.030	306.9%
LI	0.009	0.010	9.3%
M88KSIM	0.004	0.039	808.9%
MCF	0.277	0.383	37.9%
PARSER	0.031	0.037	19.1%
PERLBMK	0.012	0.124	934.2%
VORTEX	0.012	0.014	22.7%
VPR_PLACE	0.049	0.104	110.3%
VPR_ROUTE	0.016	0.145	788.2%
Average			272.5%

(a) Comparing program miss rates.



(b) Percentage locality misses in parallel regions.

Figure 4.1: The TLS cache locality problem: (a) comparing data cache miss rates for the original sequential execution and the speculatively-parallel TLS execution; (b) the fraction of data cache misses during parallel regions where the missing cache line is currently resident in another processor’s data cache (we call these *locality misses*).

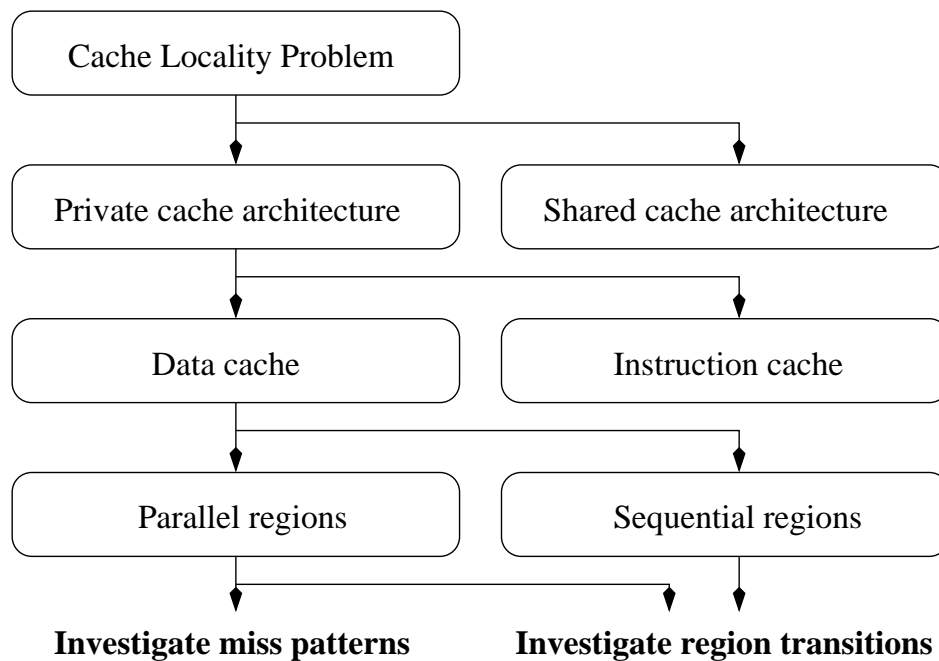


Figure 4.2: Our investigation of the cache locality problem.

increases in overall cache miss-rate, except for VORTEX: although VORTEX has 92.2% locality misses in parallel regions, it only has 22.7% increase in cache miss rate with TLS execution. As we have seen in Table 3.3, the parallel regions in VORTEX only covers 4% of the entire program, hence limits the impact of broken cache locality in parallel regions on the overall cache miss rate. The results clearly illustrates the cache locality problem for TLS.

4.3 Breakdown of the Cache Locality Problem

Since the cache locality problem for TLS systems is quite broad, in this section we systematically break the problem down so we can focus on the most important opportunities for improvement, as illustrated in Figure 4.2. We consider shared cache architectures, such as when an SMT [40]

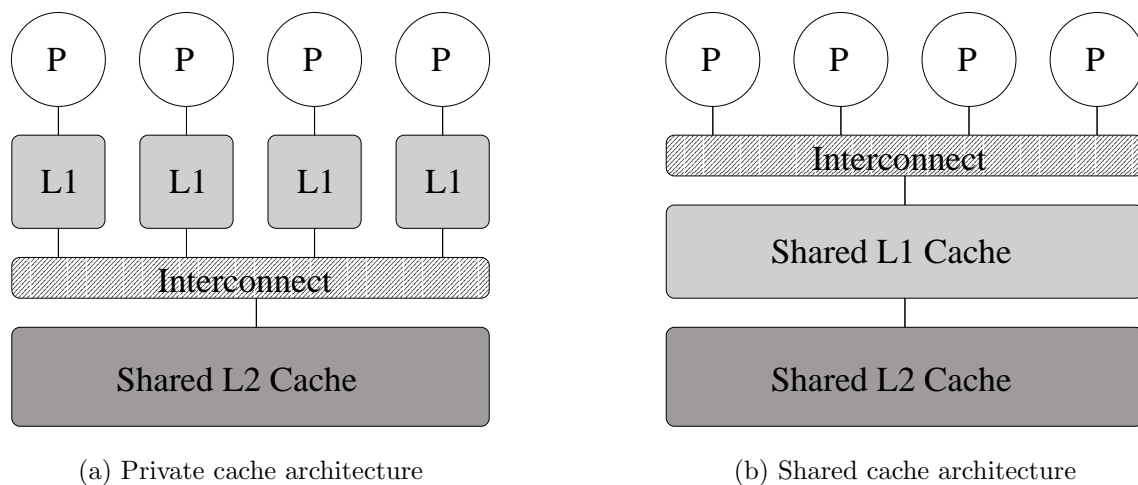


Figure 4.3: Private and shared cache CMP architectures

processor supports TLS [2, 33], although the more substantial locality problems are the result of private cache TLS support. Similarly, we consider instruction cache locality for TLS, but find that instruction cache misses are overshadowed by the impact of data cache misses. We divide data cache miss behaviour temporally, into regions where (i) the program executes sequentially and (ii) those where it executes speculatively in parallel. We further classify the data cache misses observed during the execution of parallel regions into several categories based on the observed patterns of misses. The results of this classification suggest several techniques for improving TLS cache locality which we evaluate later in this dissertation.

4.4 Private Cache and Shared-Cache Architectures

TLS support has been proposed for both shared [2, 37] and private [17, 19, 38] cache architectures, and both have interesting cache behaviour. For a shared cache architecture, as shown in

Figure 4.3(a), the hardware-supported threads of execution (such as independent processors or the contexts of a *simultaneously-multithreaded* processor (SMT) [40]) all share the same cache hierarchy. In this case the cache locality of the original sequential program is relatively preserved for the TLS execution, since only the one cache hierarchy is used. Although the TLS execution does suffer from *speculative versioning conflicts* [37] (the need to store multiple speculative versions of the same cache line in a single cache), this problem is beyond the scope of this work and has been addressed previously [37].

For current CMPs, architectures where each processor has its own private first-level data and instruction caches, as shown in Figure 4.3(b), are more common than shared-cache architectures. In the private-cache case, the cache locality enjoyed by the original sequential program has definitely been disrupted (as demonstrated in Figure 4.1), hence we focus our efforts on this area.

4.5 Execution Stages

A TLS program, like any parallel program, is divided into regions of code which are executed either sequentially or in parallel. The cache behaviour of a TLS program will therefore change significantly as the program transitions from sequential to parallel execution and back again, hence we further divide the problem into the different stages of execution that occur: *startup*, *steady-state*, and *wind-down*, as shown in Figure 4.4. These stages are repeated throughout the execution of the program, occurring for each dynamic parallel region instance, and exhibiting the following behaviour.

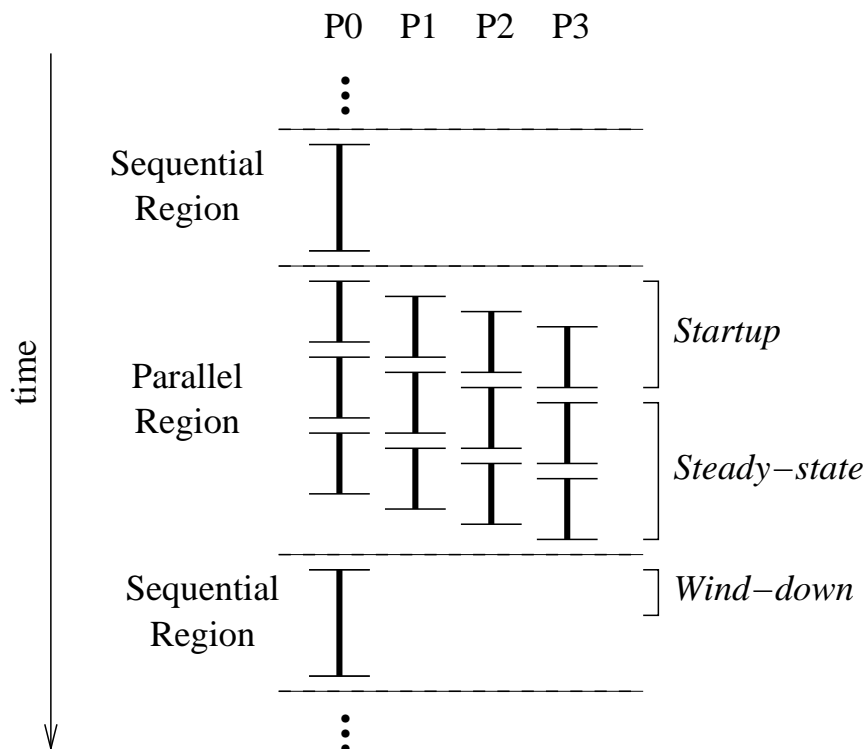


Figure 4.4: Execution stages exhibited by a TLS program.

Startup: In the initial transition from sequential to parallel execution, it is expected that certain data which will be used by every speculative thread will result in locality misses for every processor, with the possible exception of the processor which executed the preceding sequential region. We define *startup misses* as those that occur for the first speculative thread per processor at the beginning of a parallel region. In order to investigate the potential of eliminating these startup misses, we conducted an experiment where all processors copy the entire private data cache from the processor that runs the previous sequential region (at no cost); therefore, every processor starts executing the parallel region with a warm cache. The copying process is modeled to be instantaneous and does not generate any traffic. Figure 4.5 shows the performance of this model relative to the baseline. The results are mixed: five benchmarks have more than 1% speedup; five benchmarks have within 1% performance impact; three benchmarks have more

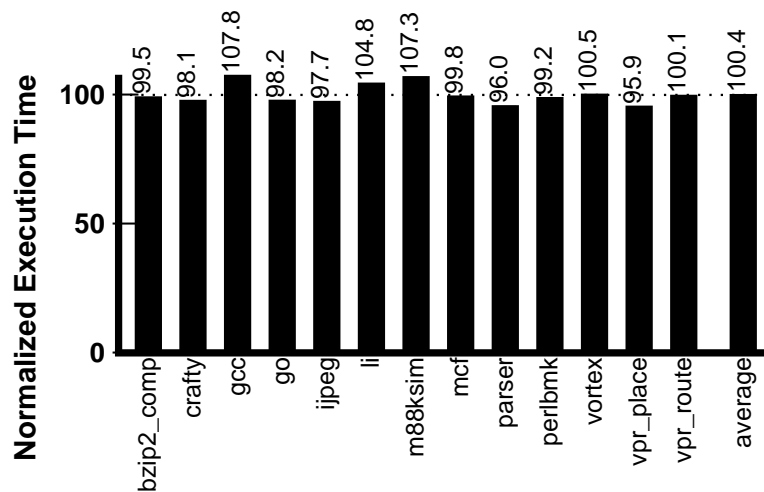


Figure 4.5: Impact of duplicating the entire sequential data cache to all private caches at the beginning of parallel regions.

than 4% slowdown. On average, warming up the private caches by duplicating the sequential cache improves performance by less than 0.5%, which is negligible. Although *startup* misses exist in parallel regions, addressing them directly is not worth it.

Steady-state: In a parallel region, after suffering any startup misses, execution enters a *steady-state* where we expect the majority of locality misses to occur. This stage of execution is our main focus, and we further classify its data cache misses in Section 4.7.

Wind-down: At the end of a parallel region we transition back to sequential execution, and expect that this sequential region will initially suffer a significant number of locality misses for data resident in the caches of temporarily inactive processors. In Chapter 5.1 we demonstrate how scheduling the sequential region can impact the number of locality misses observed during this *wind-down* stage.

4.6 Data and Instruction Cache

Access patterns for instructions and data are quite different: instruction references are normally read-only and exhibit high locality, while data references are read/write and can show a broad range of behaviour. Hence the memory hierarchy typically has separate first level caches for instructions and data, and we are obliged to investigate them separately.

To clarify the potential benefits of improving cache locality for TLS execution, we model ideal instruction and data caches where every reference is considered to be a hit. As shown in Figure 4.6, we found that the ideal data cache (D) improved the performance of parallel regions by 19% on average, which indicates there is a good potential in eliminating data cache misses. Most benchmarks have within 30% performance gain with an ideal data cache, while MCF has a huge 65% reduction in execution time. As we mentioned before, MCF’s performance is limited by the large amount of misses at the shared level-two cache. Since our work mainly focuses on improving cache locality in the private caches, we would not expect MCF to have a significant amount of speedup with our proposed techniques. However, existing work that involves prefetching from off-chip memory for uniprocessors should be applicable to MCF, and is expected to have a complementary effect with our techniques.

On average, The ideal instruction cache (I) provides 0.1% improvement. This result is intuitive, since so far only loops have been speculatively parallelized in our benchmark applications; loops will have very good instruction cache locality, with the exception of possible cold misses during the *startup* stage of each parallel region. We also evaluate the case with both ideal instruction and data cache (I/D), however it only yields a tiny 0.3% performance gain compared to having an ideal data cache only. Since instruction references involve only reads, simple methods such as broadcasting all references, or having a shared instruction cache would eliminate this

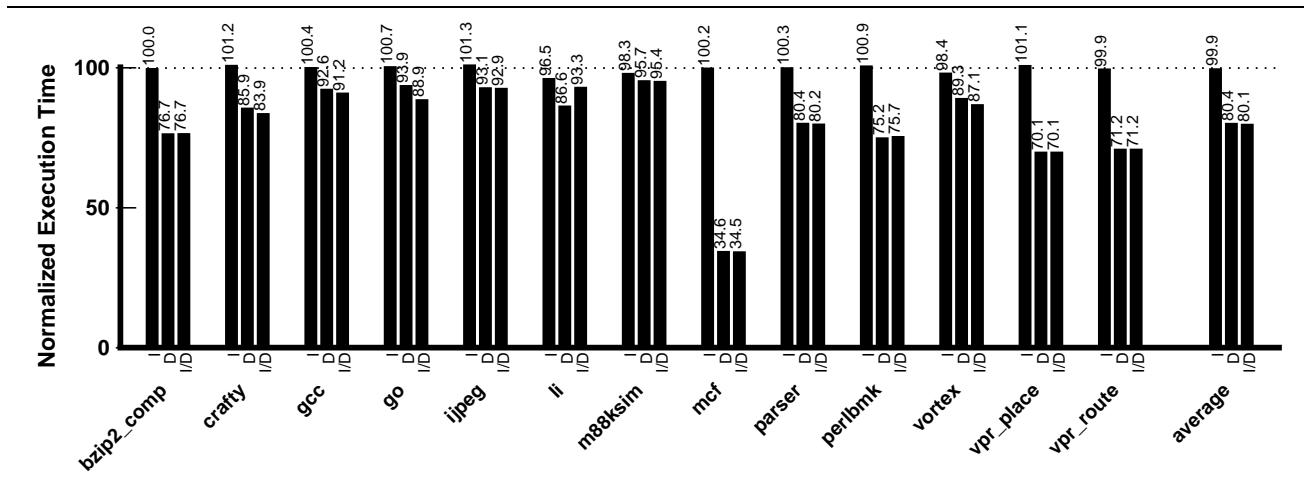


Figure 4.6: The impact of ideal instruction and data caches where misses have no latency on TLS execution. I is the ideal instruction cache model; D is the ideal data cache model; I/D is both the ideal instruction and data cache model.

small overhead. However, we do not evaluate these solutions further because there is limited performance gain, and instead focus on improving cache locality for data.

4.7 Data Cache Miss Patterns in Parallel Regions

To guide our efforts to improve cache locality for parallel regions, we analyzed traces of data cache misses. We collect a trace of all the data cache misses that occur within parallel regions. For each of these misses, we record: (i) the cycle-time when it happens; (ii) the program-counter value of the load/store instruction; (iii) the target memory address of the miss; (iv) the ID of the processor that generates the request; (v) the type of the cache request (speculative/non-speculatively, read/write); and (vi) whether it is a L2 miss.

After we obtain the trace of cache misses from our simulator, we feed the trace into our classifier which observes the following five patterns of miss:

1. Second-level cache (L2) misses
2. Read-based sharing misses, where a cache line is read by multiple processors. Figure 4.7 shows two types of read-only sharing misses: *full-broadcast* and *partial broadcast*. *Full-broadcast* involves reads to the same cache line from all four processors, while *partial-broadcast* only requires two or three processors to read the same data.
3. Write-based sharing misses, where a cache line is written (and possibly read) by multiple processors. While there are many cases of write-based sharing, two most common cases include *producer-consumer* and *migratory pattern*, and are demonstrated in Figure 4.8. *Producer-consumer* involves a producer processor that writes a data, followed by one or more consumer processors read the same data. Multiple processors writing to the same cache line results in *migratory pattern*.
4. Strided miss patterns, where the addresses of missing cache lines progress by a fixed stride, as shown in Figure 4.9.
5. The remaining misses (*other*), which apparently have no observable pattern and are likely conflict and capacity misses, as shown in Figure 4.10.

Since we are tracking misses at the cache line level, instead of the actual addresses of the load/store, false sharing is included in our sharing miss pattern and is one of the major causes of locality misses.

Some misses fit under multiple categories—for example, a strided miss may also be a L2 miss; a read-based miss can also be a write-based miss when the same cache line is written by one processor and read by multiple processors. To ensure that categories are disjoint, we

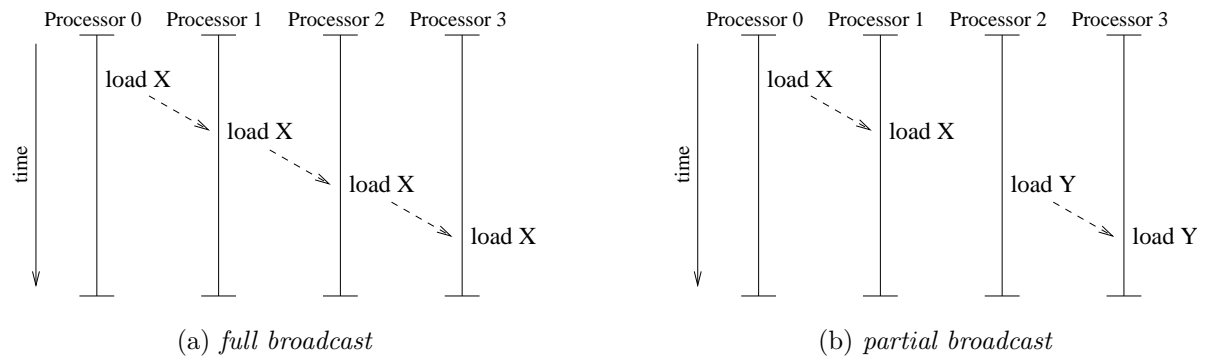


Figure 4.7: Read-based sharing misses: *full-broadcast* involves reads to the same cache line from all four processors, while *partial-broadcast* only requires two or three processors to read the same data.

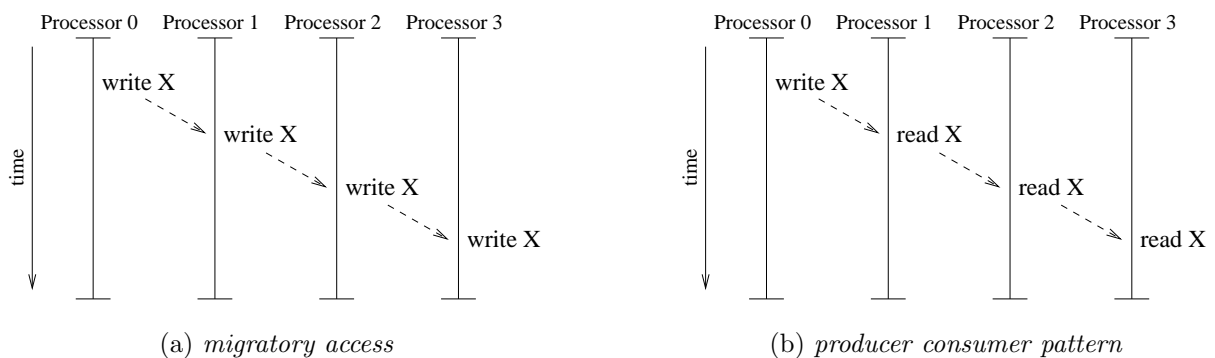


Figure 4.8: Write-based sharing misses: *producer-consumer* involves a producer processor that writes a data, followed by one or more consumer processors read the same data. Multiple processors writing to the same cache line are considered to be *migratory pattern*. These two patterns are the most common ones in write-based sharing, while there are other cases.

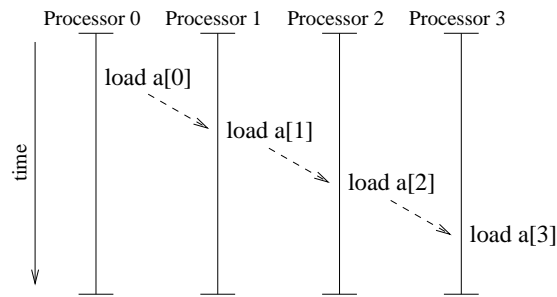
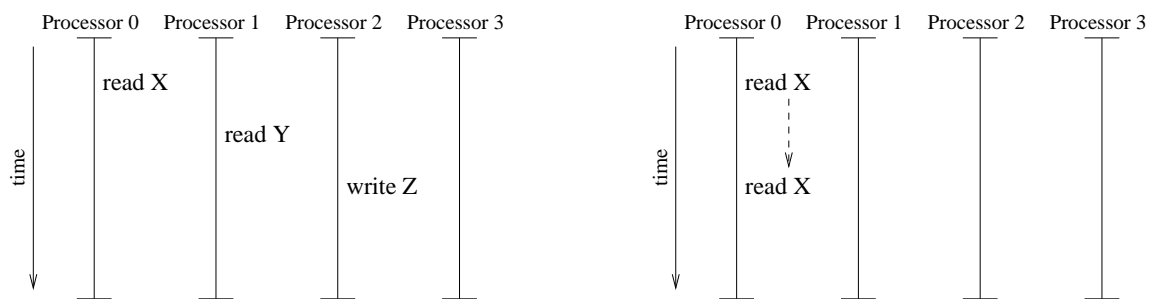
(a) *stride access*Figure 4.9: *Strided pattern* that occurs over different processors.(a) *random access*(b) *same dcache access*

Figure 4.10: Other access patterns: when there is no other access from any processor to the same cache line within the window, we consider those misses as *random accesses*; *same dcache accesses* are consecutive misses to the same cache line from the same processor, these are mainly conflict misses.

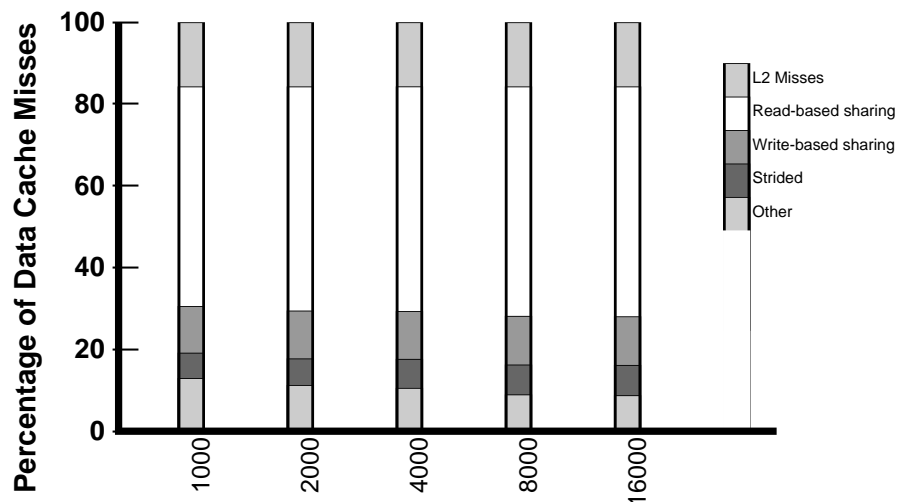


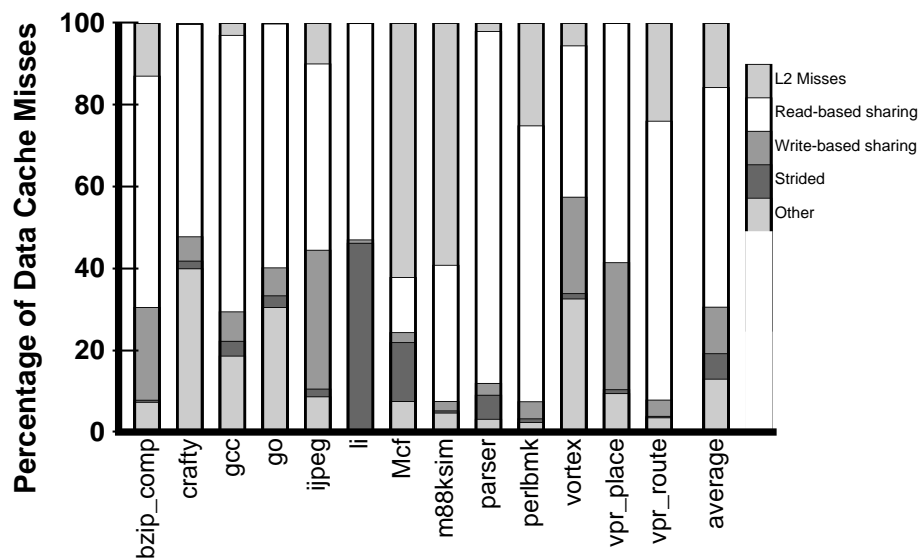
Figure 4.11: Miss pattern breakdown averaged across all benchmarks with varies window size (1000 - 16000 cycles).

attempt to fit each miss within the described categories in priority order. Figure 4.12(a) lists the percentage of misses for each category (which are in this priority order), averaged across all benchmark applications. We also show the breakdown for each individual application in Figure 4.12(b).

Classifying miss patterns based on a trace must be done carefully. To decide whether a set of misses belong in a given category, we analyze all of the misses within a fixed-size sliding window of 1000 cycles of the trace. Figure 4.11 shows the miss breakdown averaged cross all benchmarks with different window sizes. As the window size changes from 1000 cycles to 16000 cycles, the breakdown does not vary significantly. The portion of L2 misses stays unchanged, while there are tiny increases in read-based, write-based and stride categories. These results indicate that 1000-cycle window size is large enough to capture most of the recognizable miss patterns.

Data Cache Miss Pattern	Percentage
L2 misses	15.7%
Read-based sharing	53.7%
Write-based sharing	11.4%
Strided	6.2%
Other	13.0%

(a) Miss pattern breakdown averaged across all benchmarks.



(b) Detailed miss pattern breakdown

Figure 4.12: Data cache miss patterns within parallel regions.

From the figure, we observe that data cache misses within parallel regions exhibit interesting behaviour. L2 Misses, which comprise an average of 15.7% of all misses, are given the highest priority: an L2 miss cannot also be a locality miss (since our system enforces the *inclusion property*) hence we do not want to target these misses. L2 misses are an equivalent problem for both the sequential version of a program and its TLS counterpart, and can be potentially addressed with known techniques for prefetching [5, 11, 15, 32]. More than about 60% of the data cache misses in MCF and M88KSIM are L2 misses, which limits the potential performance improvement of our locality techniques.

Cache lines that exhibit read-based sharing are the most common, representing 53.7% of all misses. This is promising, since under TLS read-based cache lines are much easier to deal with than those which are modified. In particular, PARSER has 86% of read-based sharing misses, indicating that most of the misses are shared among the processors. However, as shown in Figure 4.1(a), it only has 43% locality misses: most of the misses in PARSER are *partial-broadcast* which involves two processors sharing the same data. We address this category of misses in Section 5.2. In contrast, cache lines involved in *write-based sharing* represent 11.4% of misses, but are still worth addressing—hence we do so in Section 5.3. Strided accesses comprise another 6.2% of misses, and so we address those in Section 5.4. Among all 13 benchmarks, LI has the most stride misses, which account for 46% of the total cache misses—hence we expect that LI would perform well with our stride prefetching technique. Finally, cache lines in the *other* category represent the remaining 13.0% of misses. Although our locality techniques do not directly address these misses (since eight out of thirteen benchmarks have only less than 10% of misses in this category), our techniques are sufficient to target the majority, 71.3%, of the data cache misses in parallel regions.

4.8 Summary

In this chapter we described previous work on improving cache locality and provided a thorough investigation into the TLS cache locality problem. We systematically decomposed the problem and decided to focus mainly on data cache locality within parallel regions. In the remainder of this dissertation we propose techniques to address the transitions between sequential and parallel regions, as well as the sharing and strided miss categories which combined constitute more than 70% of all cache misses in parallel regions.

Chapter 5

Techniques for Improving TLS Cache Locality

Having motivated the TLS locality problem, this chapter describes techniques that target different areas of the locality problem. We present four locality techniques which cover both sequential regions and parallel regions, and also different categories of data cache access patterns in parallel regions. We evaluate our proposed techniques with detailed simulation across 13 benchmarks and compare the results with the baseline STAMPede TLS hardware.

5.1 Scheduling the Sequential Region

In the *wind-down* stage (as illustrated in Figure 4.4), we expect that the sequential region will suffer locality misses for cache lines that are resident in the caches of now inactive processors. In this chapter we investigate the impact of the different possibilities for scheduling which processor executes the sequential region. In particular we consider two options, as illustrated in Figure 5.1. First, we consider a “floating” sequential processor, where the processor which executed the last

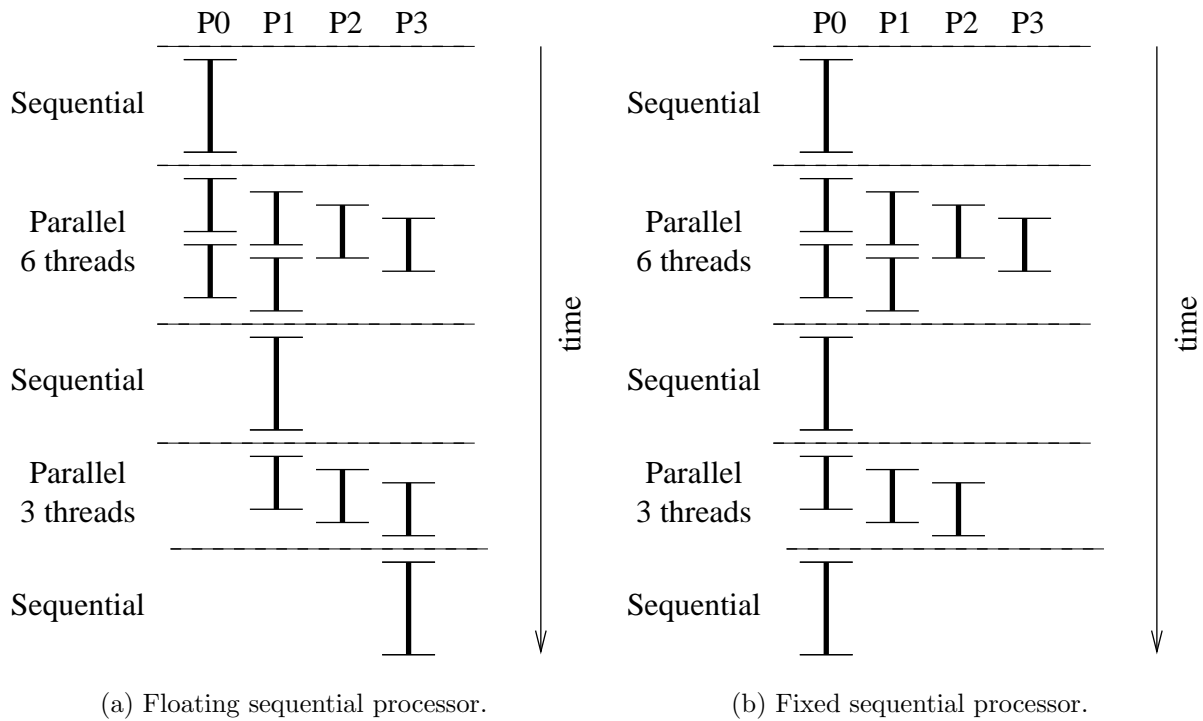
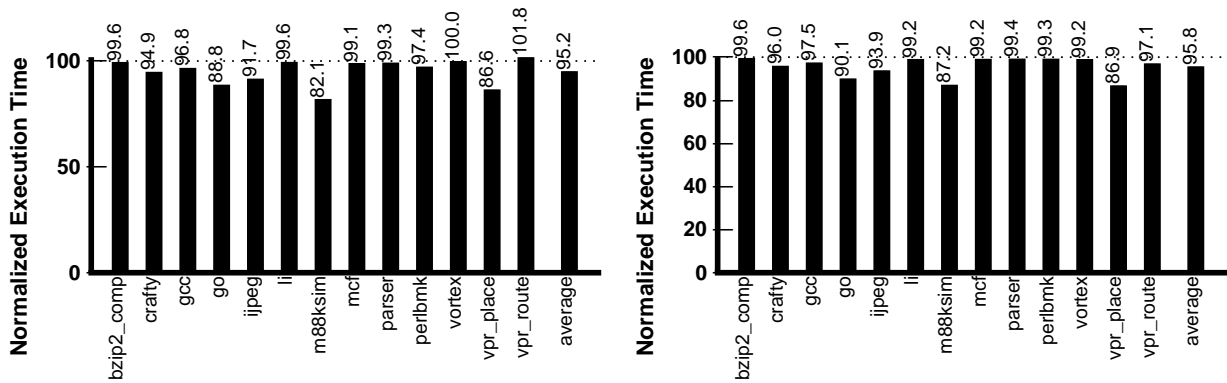


Figure 5.1: TLS execution with a floating and fixed sequential processor. In (a), the processor which executed the last speculative thread of the parallel region goes on to execute the sequential region. In (b), one processor (P0) is elected to execute all sequential regions.

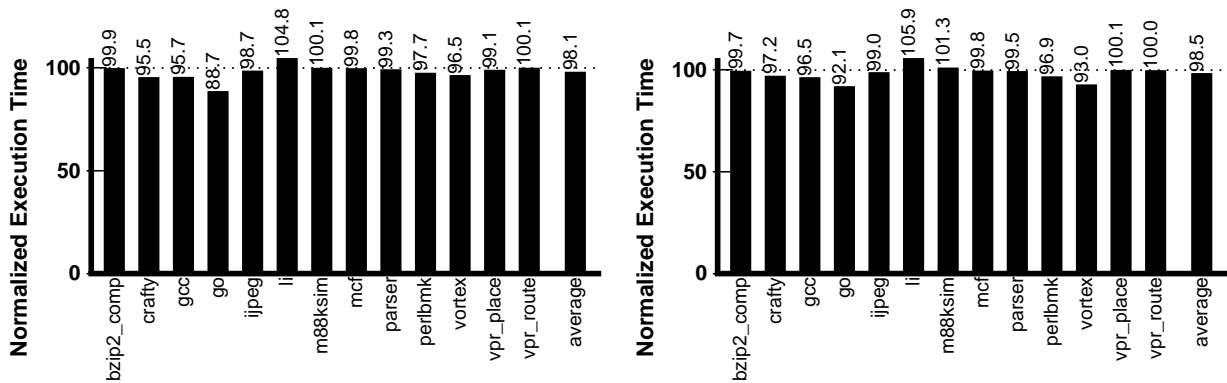
speculative thread of the parallel region goes on to execute the sequential region. Intuitively, this scheme assumes that there is potential cache locality between the last speculative thread and the sequential region. Second, we consider a “fixed” sequential processor, where one processor is elected to execute all sequential regions. This scheme assumes the potential benefit of cache locality between sequential regions, for cache lines which are not evicted during the parallel region.

In Figure 5.2(a) and 5.2(b) we show the performance of sequential regions only for the fixed sequential processor scheme relative to floating. Evidently the cache locality between sequential regions is much more prevalent than the locality between the last speculative thread of a parallel



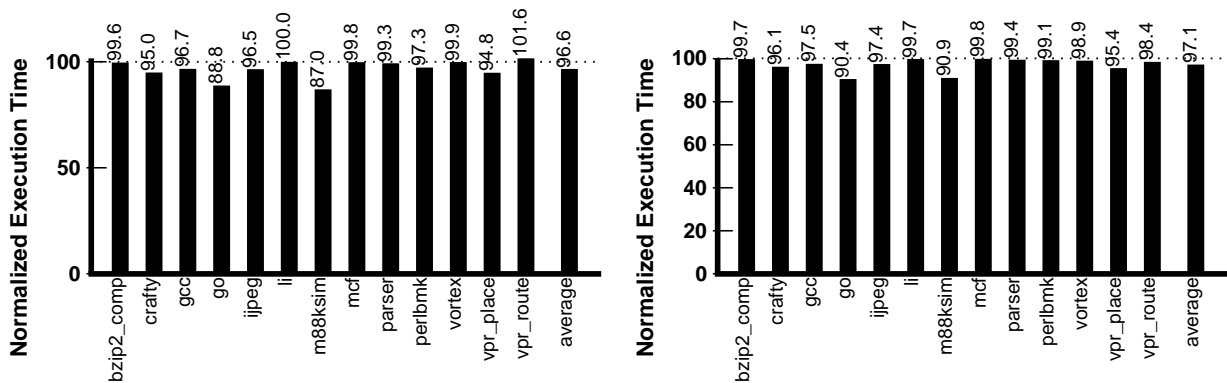
(a) Sequential Regions (bus).

(b) Sequential Regions (crossbar).



(c) Parallel Regions (bus).

(d) Parallel Regions (crossbar).



(e) Overall Program (bus).

(f) Overall Program (crossbar).

Figure 5.2: Performance impact of a fixed sequential processor relative to floating.

region and the subsequent sequential region: on average the fixed sequential processor scheme performs nearly 5% better with a bus interconnect, and nearly 4% better with a crossbar. For both architectures, three applications (GO, M88KSIM, and VPR_PLACE) perform more than 10% better in sequential regions. Figures 5.2(c) and 5.2(d) show the performance within parallel regions; on average the fixed sequential processor scheme outperforms by 1.9% and 1.5% with a bus and a crossbar respectively. Parallel regions also benefit since all the threads are guaranteed to execute on the same processor since the fixed processor always runs the first thread of every parallel region and speculative threads are assigned in a round-robin fashion. This decision also has a significant impact on program performance, as shown in Figure 5.2(e) and 5.2(f) where three applications perform more than 5% better, and all benchmarks perform on average 3.4% and 2.9% better for bus and crossbar respectively. In the bus architecture, since all the processors share the same bus, they benefit directly from the reduced contention in the bus; while there are multiple connections in a crossbar interconnect, reducing the contention in one link will not directly benefit all processors.

While the improvement with a fixed sequential processor is not tremendous, proper scheduling of the sequential region is essentially free and therefore worth doing. Hence we use the fixed sequential processor scheme for the remainder of our investigation and consider it to be part of our “baseline” measurement when comparing with other schemes for improving cache locality.

5.2 Exploiting Read-Based Sharing Patterns

Our classification of miss patterns within parallel regions shows that read-only sharing miss patterns dominate all other access patterns, comprising more than half of all cache misses. In other words, a large number of misses are for data which multiple speculative threads read.

Therefore, for any given load miss it is highly likely that other processors will soon suffer a load miss for the same cache line. This observation motivates a technique which, for a given load miss, “pushes” the resulting cache line to caches other than the one which originally suffered the load miss.

5.2.1 Broadcasting for all Load Misses

The simplest scheme for addressing read-based sharing patterns is for every load miss to result in a broadcast of that cache line to all data caches. For a CMP with a bus interconnect between the first-level data caches and unified second-level cache, such as we model in this work, such broadcasting is fairly trivial to implement and does not generate additional traffic. All data caches simply snoop on the bus for any read requests serviced by the unified cache and fill their caches with the resulting cache line, so long as doing so does not replace a cache line that is currently in a speculative or modified state (if this occurs then the broadcast cache line is simply dropped). Implementing such broadcasting with a crossbar interconnect is less efficient, since pushing data to each private cache requires a separate connection.

We do not expect this scheme to eliminate all of the cache misses involved in read-only sharing patterns (50% of all misses) since we require at least one miss to trigger the broadcast mechanism for every such cache line. Figures 5.3(a) and 5.3(c) show the performance of this broadcasting scheme relative to our baseline model. On average across all benchmarks, this simple broadcasting scheme eliminates 27.7% and 23.9% of the data cache misses in speculative regions, for bus and crossbar architectures respectively. This technique also improves execution time for every application with a bus interconnect, by 7.3% on average, as shown in Figure 5.3(b). Three applications, PARSER, VPR_PLACE, and VPR_ROUTE have more than 10% reduction in

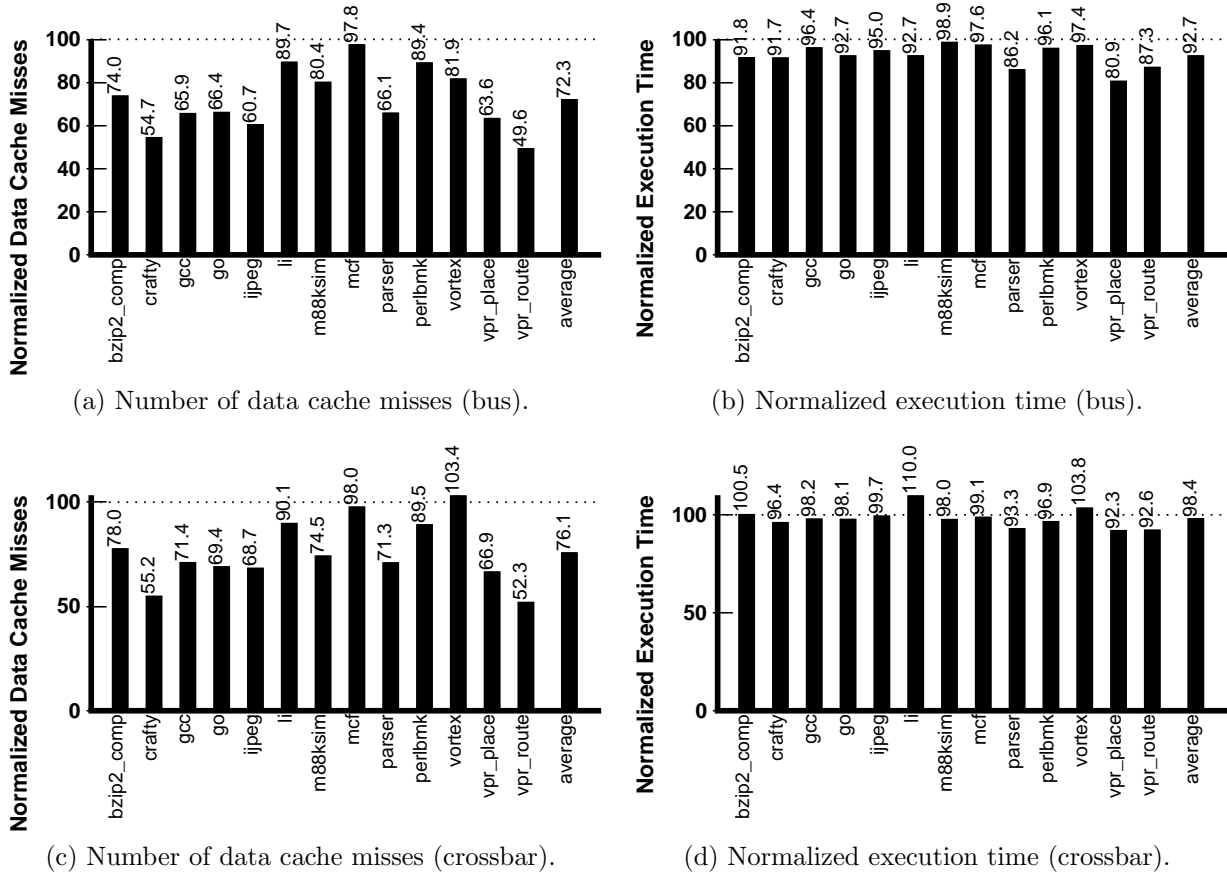
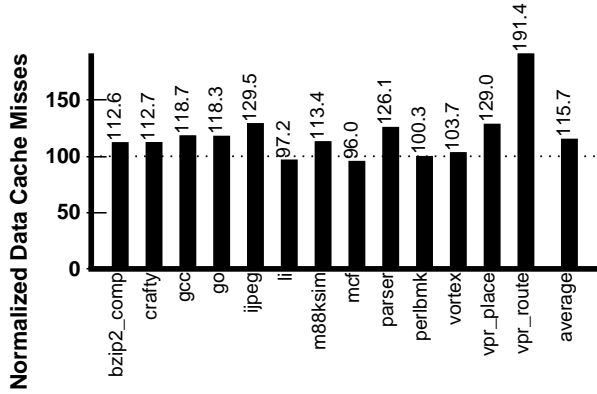


Figure 5.3: Impact of broadcasting all load misses on parallel regions.

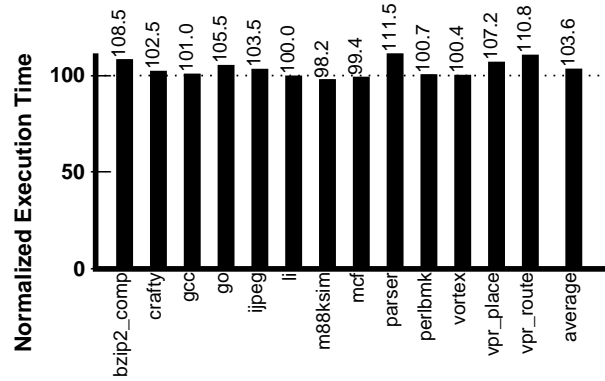
parallel region execution time since most of the data cache misses are read-based sharing, which benefit most from this broadcasting scheme. In the case of a crossbar interconnect, although this scheme reduces the number of misses, performance does not improve as much as it does with a bus interconnect. On average, broadcasting gives 1.6% speedup across all benchmarks, as demonstrated in Figure 5.3(d), with three benchmarks having a slowdown ranging from 0.5% to 10%. The main reason is due to the inefficiency of broadcasting with a crossbar which requires an additional interconnect transaction to send data to each other processor, hence increasing the contention of the crossbar. One example is LI which has 7.3% performance improvement with a bus interconnect, but 10% slowdown with a crossbar. The additional broadcasting traffic is clearly the cause of the slowdown for crossbar in this case, since in both cases (bus and crossbar) broadcast reduces data cache misses by roughly 10%.

5.2.2 Throttling Broadcast

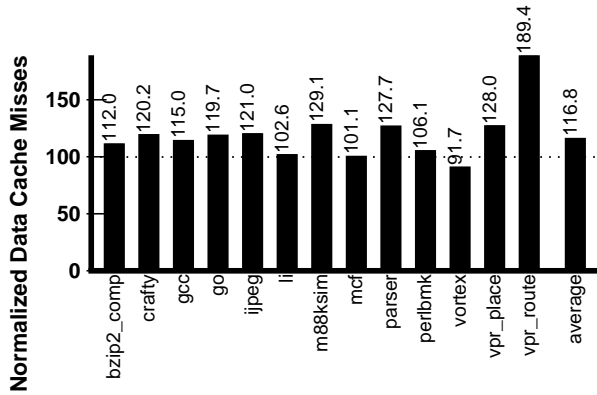
Although the scheme proposed above for broadcasting the cache line for each load miss to every cache seems to have worked well, a potential overhead of this scheme is the pollution created when a cache line is “pushed” to a cache which does not use the cache line, possibly evicting a useful cache line in the process. To investigate whether such cache pollution is a significant overhead, we attempted to throttle the amount of broadcasting, narrowing the broadcast cache lines to those which are truly needed by multiple caches. Before designing an efficient implementation of throttled broadcast, we began by modeling an unrealistic but aggressive scheme by feeding our profile of read-only sharing misses (from our trace of execution described in Section 4.7) into a second simulation run which would then only broadcast cache lines that were pre-identified as showing this pattern. We found that aggressively throttling broadcast in this manner does not further improve performance with a bus interconnect, as shown in Fig-



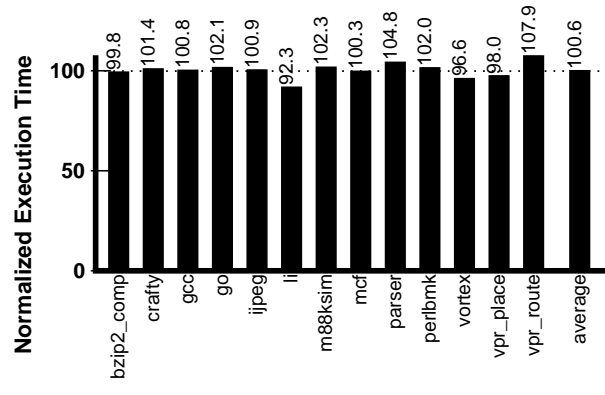
(a) Number of data cache misses (bus).



(b) Normalized execution time (bus).



(c) Number of data cache misses (crossbar).



(d) Normalized execution time (crossbar).

Figure 5.4: Impact of throttling broadcasting with profiling on parallel regions, relative to the full broadcast scheme.

ures 5.4(a) and 5.4(b), indicating that pollution is not a problem for broadcast load misses in that case. Besides pollution, the increased traffic of broadcast could be a problem for the crossbar interconnect; for such a system a more throttled broadcast may be beneficial. We model the profile-base broadcast with a crossbar and the results are shown in Figures 5.4(c) and 5.4(d). On average the throttled broadcast scheme does not perform better than the full broadcast scheme. In the case where broadcasting is detrimental, such as in LI and VORTEX, throttling successfully eliminates bad broadcasts to preserve interconnect bandwidth for other useful traffic, and hence outperforms the full broadcast scheme. However in the case when broadcasting is beneficial due to large amount of data sharing, throttling reduces good broadcasts and limits the potential performance gain by broadcasting. We do not recommend throttling broadcasting since it does not give any performance gain over the simple full broadcast scheme in most benchmarks and performing throttling at runtime requires expensive hardware to distinguish good and bad broadcasts. Compiler techniques for recognizing broadcasting patterns are an interesting possibility, however this is out of the scope of this work since we focus solely on hardware techniques.

5.3 Exploiting Write-Based Sharing Patterns

The underlying coherence scheme for supporting TLS execution that we use is necessarily a write-back scheme, since only the first-level data caches may hold speculative modifications; in other words, any speculatively-modified cache line must remain in the first-level data cache until the corresponding speculative thread is committed, at which point that cache line simply transitions to a normal modified state. When such a modified (and non-speculative) cache line is read or written by another processor, or if it is replaced, then that cache line must be written back to the second-level cache and propagated to the requesting processor's data cache.

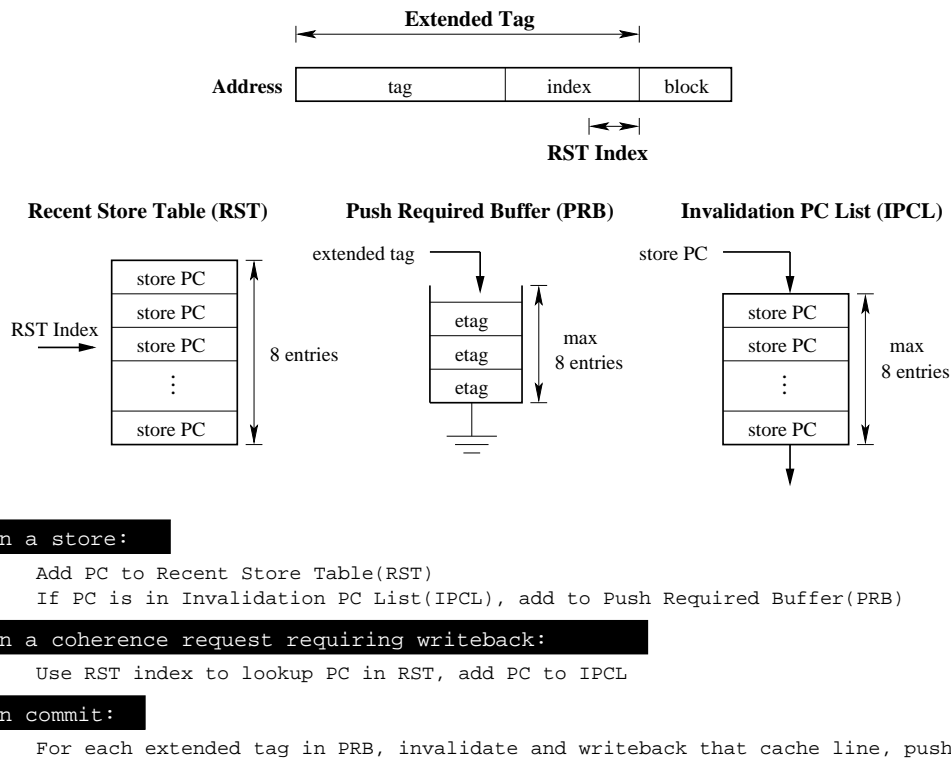


Figure 5.5: A mechanism for detecting cache lines involved in write-based sharing, and aggressively invalidating and forwarding them to the next cache. Each processor maintains a *recent store table*, *push required buffer*, and an *invalidation PC list*.

In Chapter 4, we showed that write-based sharing misses are significant, hence we endeavor to reduce them. Ideally, any cache line which is written by one processor and then accessed by another could be aggressively propagated ahead of time. One scheme would be to broadcast all modified cache lines at commit-time—but this scheme would generate too much traffic, and would increase the amount of time to acquire exclusive ownership when writing a cache line (since the broadcast would have created so many copies). Instead we prefer a more selective scheme which predicts when a cache line is involved in write-based sharing, and at commit time writes-back, self-invalidates, and pushes that cache line to the next processor—eliminating the cache miss, and expediting the acquisition of exclusive access.

We propose a mechanism for predicting cache lines involved in write based sharing, which consists of maintaining the following three components per processor—as illustrated in Figure 5.5. First, a *recent store table* (RST) which is direct-mapped, indexed by the last 3 bits of the set-index of a store address, and tracks the PCs of recent stores. Second, a *push required buffer* (PRB) which saves a list of extended cache tags (tag plus set index) which are to be invalidated, written back, and pushed to another cache when the speculative thread commits. This list can simply overflow when it is full, since correctness is not an issue for this technique. Finally, we require an *invalidation PC list* (IPCL)—a FIFO queue of store PCs.

The operation of our technique is as follows. When a store executes, the store PC is saved in the RST (using three bits of the store address as an index). If that PC is currently in the IPCL, then it has been identified as being involved in write-based sharing in the past, and hence the extended tag for that cache line is added to the PRB. For every external coherence request that generates a writeback, such as a read, read-exclusive, or invalidation request, we lookup the corresponding store PC in the RST, and add that PC to the IPCL. Finally, when the speculative thread commits, for each entry in the PRB we *self-invalidate* and write back the corresponding cache line, and “push” it to the next processor’s data cache. For this dissertation, we assume that speculative threads are assigned to processors in round-robin order, and hence the “next processor” is easily predictable. If this were not the case, one could easily add a processor ID to each entry of the IPCL to track which other processor is involved in the write-based sharing and should be the target of the push.

The effect of our technique is similar to that of dynamic self-invalidation [27], although our technique does not need to implement versioning numbers to decide which blocks to self-invalidate and we also attempt to push the cache line to the next cache.. Furthermore, last-touch prediction [26] cannot be used in our approach since modified cache lines may not be propagated

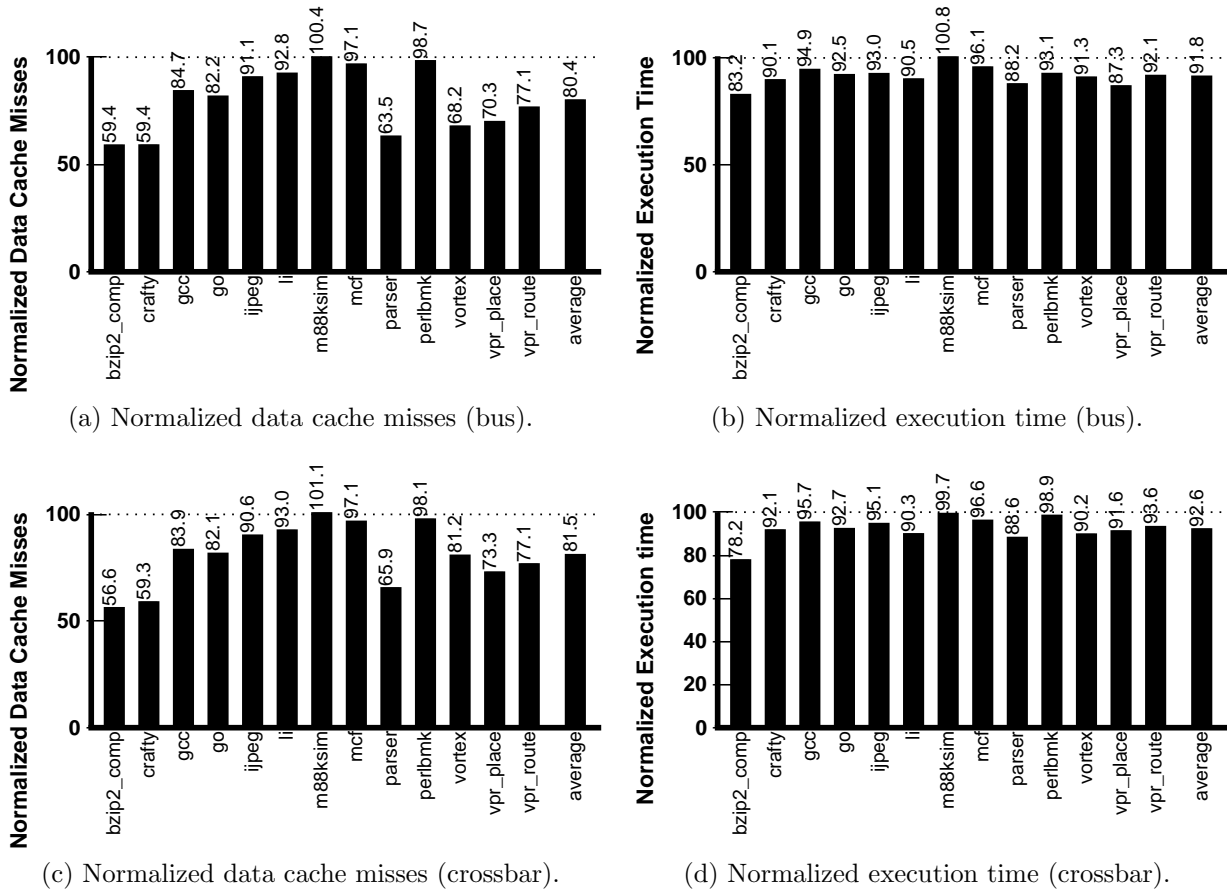


Figure 5.6: Impact of our technique for exploiting write-based sharing patterns on parallel regions.

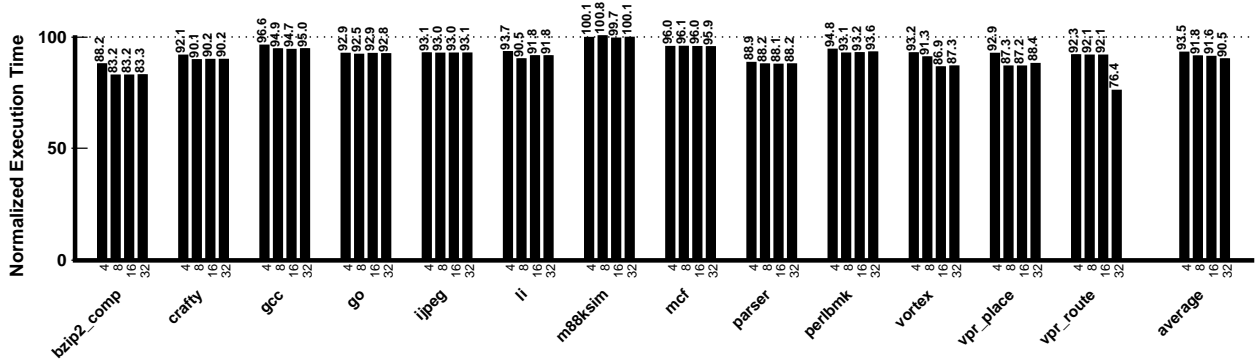
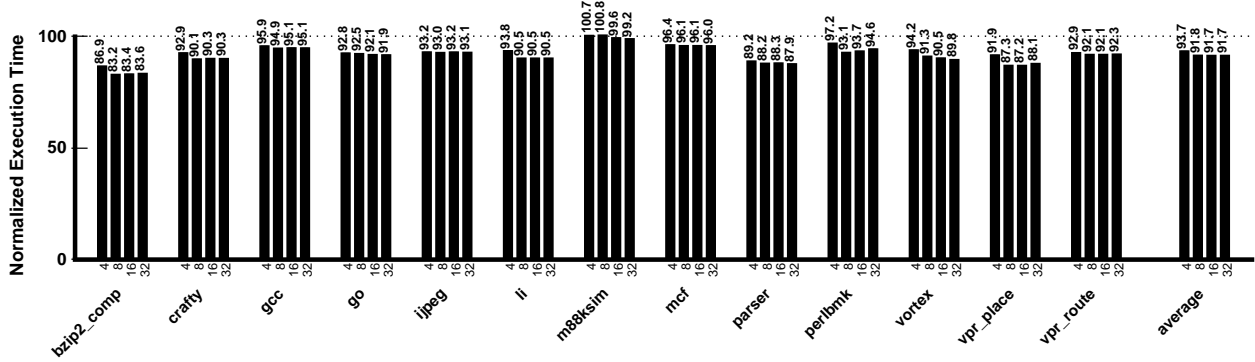
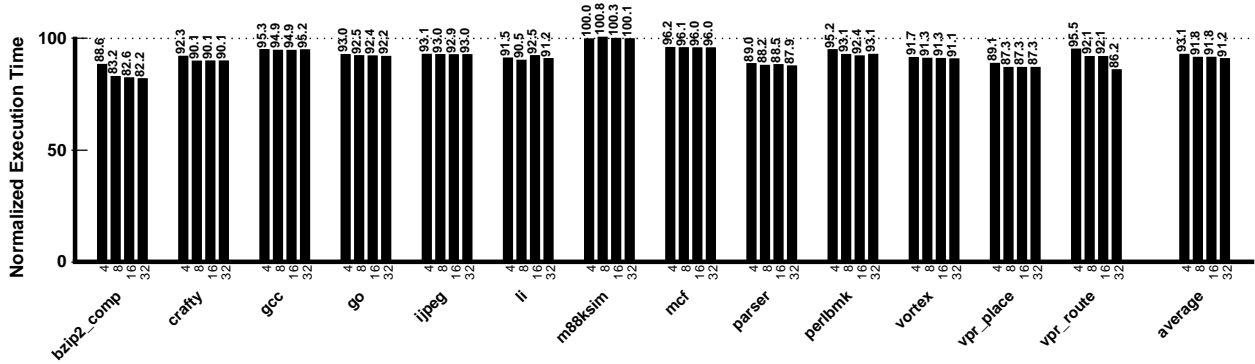
until the speculative thread commits.

Figure 5.6 shows the performance of our write-based sharing technique relative to our baseline model. With a bus interconnect, this scheme eliminates 19.6% of the data cache misses in speculative regions and improves execution time for most applications, by 7.8% on average, as shown in Figure 5.6(a) and 5.6(b). This technique also performs well on a crossbar interconnect: the number of data cache misses is decreased by 18.5% while execution time is improved

by 7.4%, as shown in Figure 5.6(c) and 5.6(d). This scheme has similar performance on both bus and crossbar for all benchmarks. It reduces data cache misses by more than 40% in two applications, BZIP2_COMP and CRAFTY, while four other applications have more than 20% reduction in misses.

Our scheme significantly decreases the number of data cache misses by an average of nearly 20%, however there is only 11.4% of write-based sharing misses according to our classification results presented in Table 4.12(a). One reason is that we assigned higher priority to read-based sharing over write-based sharing misses to avoid overlapping between categories, cache misses that are both read-based sharing and write-based sharing will be only counted as read-based sharing misses. One example is that when a write-miss is followed by two read-misses to the same address: the read-misses are read-based sharing misses since they are consecutive reads, and at the same time they are also write-based sharing misses since the cache line is produced by a recent write. Another reason for our write-based technique to have more miss reduction than the number of write-based sharing misses recorded by the trace is that we only record cache misses, but not cache hits. In the case when a speculative thread has a read miss followed by a write hit to the same cache line, the write hit is not recorded. If this pattern repeats over threads, we would record a series of read misses to the same address and hence consider them as read-based sharing misses. Indeed, this access pattern benefits most from our proposed techniques for write-based sharing misses, since writes are followed by other writes from other processors.

We investigated the sensitivity of our write-based scheme to the size of the three proposed hardware structures, *recent store table*, *invalidation PC list* and *push required buffer*. Since our design space consists of three variables, in all our experiments when we evaluate the sensitivity of the scheme to the size of one structure, we keep the size of the other two structures to be

(a) Varying the size of the *recent store table*.(b) Varying the size of the *invalidation PC list*.(c) Varying the size of the *push required buffer*.Figure 5.7: Performance impact of varying the size of the *recent store table*, *invalidation PC list* and *push required buffer* from 4 to 32 entries with a bus interconnect.

eight entries.

As we see in Figure 5.7(a), increasing the size of the RST from 4 to 8 entries improves parallel regions performance by 1.7% on average, with four applications having more than 2% speedup. Further increasing the RST to 16 and 32 entries does not have much performance impact, with the exception that `VPR_ROUTE` has a dramatic 16% reduction in execution time when switching from a 16-entry RST to a 32-entry RST. As we will see, `VPR_ROUTE` also benefits from having a huge PRB, while all other benchmarks are insensitive to size of PRB. We conclude that an 8-entry RST is sufficient to capture most of the mappings of recent store PCs and addresses.

In Figure 5.7(b) we show the parallel region performance as we vary the size of the IPCL from 4 to 32 entries. Five applications, `BZIP2_COMP`, `LI`, `PERLBMK`, `VORTEX` and `VPR_PLACE`, have more than 3% performance gain when the size of the IPCL is increased from 4 to 8 entries, with an average of 1.9% gain across all benchmarks. However, increasing the size of the IPCL beyond 8 entries does not have any significant performance improvement. This result indicates that an 8-entry IPCL is big enough to hold all the recent PCs of the store instructions that would potentially produce write-based sharing misses, and a IPCL bigger than 8 entries tends to keep more old information that is no longer useful. It is also important to keep the IPCL small in order to allow fast lookup since it is fully-associative and is accessed whenever a store occurs.

Figure 5.7(c) illustrates the performance in parallel regions with varying PRB size. On average increasing the size of the PRB from 4 to 8 entries yield 1.3% performance gain. Most applications are not sensitive to the size of the PRB beyond 8-entry, however `VPR_ROUTE` performs much better with the 32-entry PRB than the 8-entry one. The speculative threads in `VPR_ROUTE` have relatively more writes than all benchmarks that we study, therefore it benefits

from a bigger RST and PRB. We found a small, 8-entry, PRB is sufficient for most applications, mainly because threads in TLS tend to be small (to avoid failed speculation) and modify only a small number of cache lines during speculative execution.

Since the additional hardware structures required for this technique are both small and decentralized, we expect it to scale well to CMPs with larger numbers of processors.

5.4 Exploiting Strided Miss Patterns

In the previous sections we focussed on techniques for exploiting read-only and write-based sharing miss patterns. According to our classification of data cache miss patterns in Section 4.7, the third major category of misses are *strided* misses, which comprise more than 6% of all data cache misses within parallel regions. As opposed to sharing misses which involve a single cache line, strided misses involve different cache lines with addresses that are separated by a constant distance. While schemes for prefetching based on such strided access patterns have been well studied [5, 11, 15, 32], there has been relatively little investigation into how stride-based prefetching interacts with TLS execution. Our stride prefetcher is to prefetch data from the shared level-two cache to the private level-one caches, but not from off-chip to on-chip caches.

To evaluate the potential impact of stride-based prefetching on TLS execution, we model an aggressive *adaptive* stride prefetcher [11] in each processor. In our implementation we use a history table with 512 entries, each of which consists of (i) the PC of the instruction that generates the miss, (ii) the miss address, (iii) the stride distance, and (iv) the state of this entry. This fully-associative history table is indexed by the PC and uses an LRU replacement policy. A stride is successfully identified after three cache misses are associated with the same PC, with

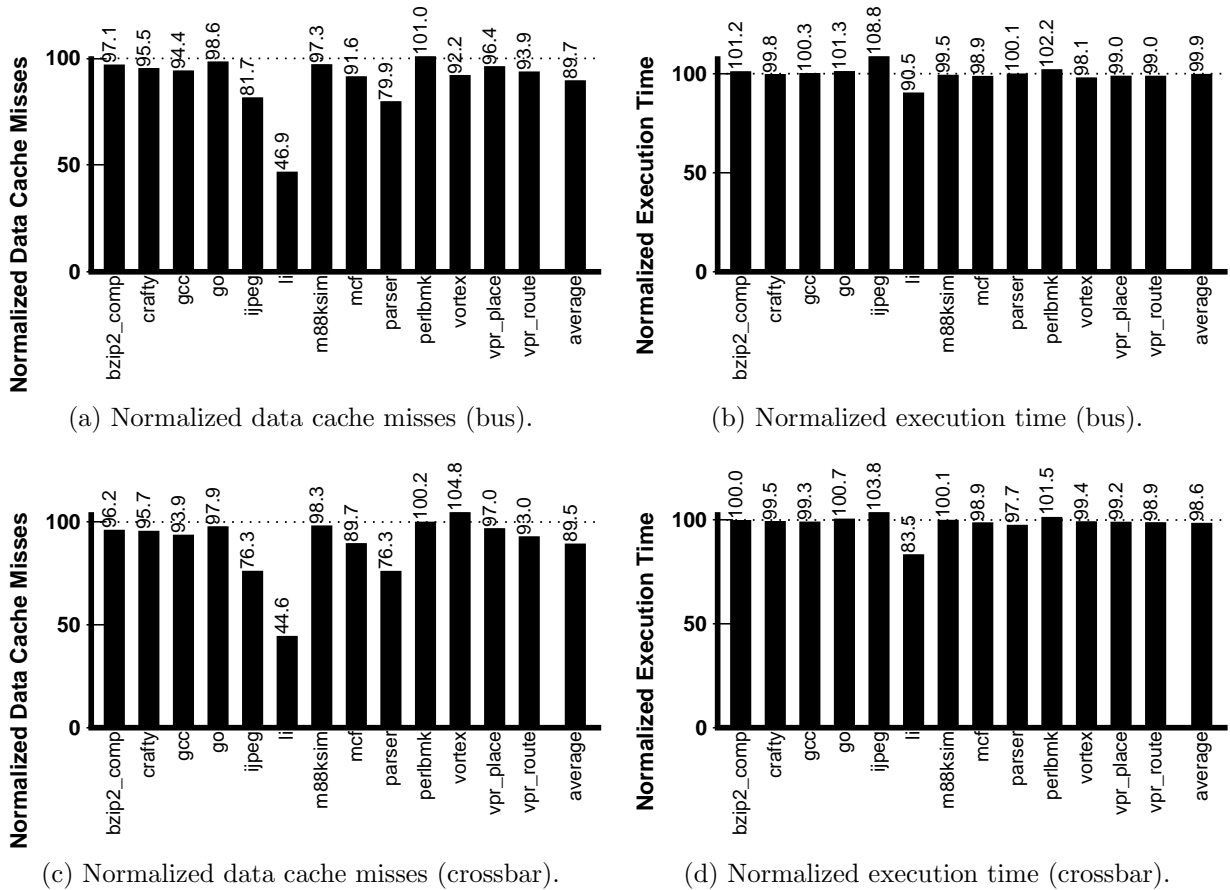


Figure 5.8: Impact of stride-based prefetching on parallel regions.

the consecutive referenced addresses differing by the same stride distance. When the prefetcher recognizes a stride, we queue 16 prefetches that continue the stride pattern and then reset the corresponding entry. The issuing of these prefetches is throttled to avoid unwanted bursts of interconnect traffic.

Figures 5.8(a) and 5.8(c) show the reduction in data cache misses in parallel regions for strided prefetching. For both bus and crossbar, data cache misses have been reduced by nearly 20% or more for three applications, LI, IJPEGE and PARSER, and by an average of 10% across

all applications—indicating that we have successfully eliminated most stride-based misses as identified in Section 4.7.

In Figures 5.8(b) and 5.8(d) we see mixed results for the corresponding performance when prefetching strided accesses. With a bus interconnect, LI, and VORTEX improve somewhat while IJPEG performs significantly worse. Although there is a significant decrease in the number of misses, strided prefetching produces bursts of traffic that hurt performance. On average, strided prefetching has no significant performance impact overall, indicating that the benefits of reduced cache misses are overwhelmed by the increase in interconnect traffic. Indeed, during parallel execution we find that interconnect contention increases eight-fold over that of sequential execution (even without strided prefetching) due to the increase in active processors running the parallel region, decreased locality, and the effects of failed speculation. Strided prefetching performs better with a crossbar, where it has 1.4% performance improvement on average across all benchmarks. The reason is that strided accesses usually involve cache lines that are located in different banks of the shared cache, and therefore the prefetching traffic is spread across all connections in the crossbar, hence reducing the performance impact of this extra traffic. To conclude, we do not recommend to use stride prefetching with a bus where extra traffic is prohibited, however stride prefetching is beneficial with a crossbar interconnect.

5.5 Summary

In this Chapter, we proposed four techniques for improving cache locality for TLS, and evaluated their impact on data cache misses and performance with both bus and crossbar interconnects. We found that using a fixed sequential processor is beneficial to both types of interconnect and implementing it is essentially free. The other three techniques target the three major categories

of data cache misses in parallel regions, as described in Chapter 4.

1. Read-based sharing—Broadcasting read misses eliminates 27.7% of data cache misses and improves performance by 7.3% with a bus; reduces 23.9% of misses and improves performance by 2.6% with a crossbar. Throttling broadcast avoids bad broadcasts and benefits a few benchmarks which fewer read-based sharing misses, however it yields little performance on average.
2. Write-based sharing—We proposed a scheme to predict which cache lines are involved in write based sharing at runtime, and at commit time we write-back, self-invalidate and push the cache line to the next processor. Our prediction scheme introduces only a small amount of extra hardware and does not require any modification to the existing hardware or coherence protocol. This technique reduces data cache misses by nearly 20% and improves parallel region performance by more than 7% with both bus and crossbar interconnects.
3. Strided access—We implement an adaptive stride prefetcher which can recognize 16 strides with arbitrary stride distance simultaneously. The stride prefetcher successfully eliminates most of the strided misses in parallel regions and reduces data cache misses by more than 10% in both bus and crossbar, however it does not have much impact on performance: 0.1% speedup with a bus and 1.4% speedup with a crossbar, due to the excess amount of traffic.

In the next Chapter we investigate the combination of all three techniques and evaluate their impact on scalability and parallel region selection.

Chapter 6

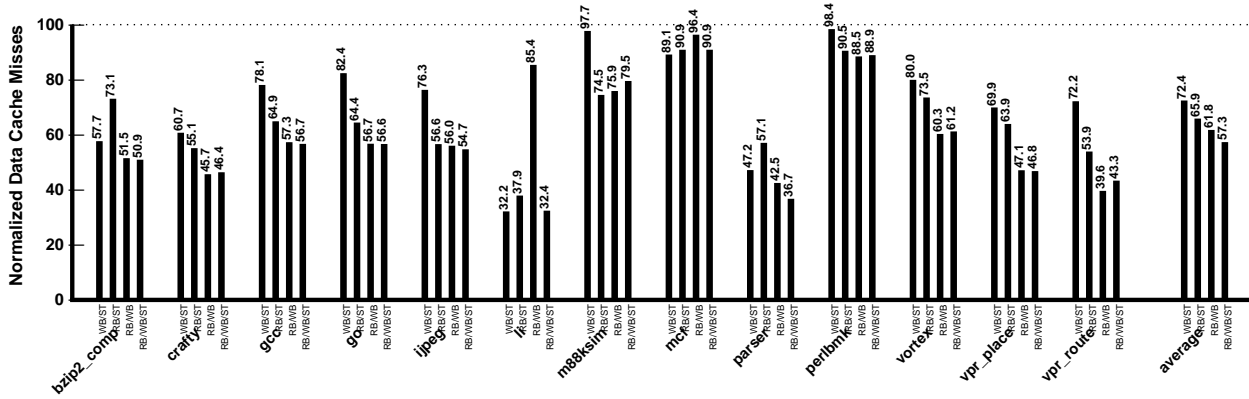
Combining the Techniques

In the previous chapter we proposed three techniques for improving cache locality for TLS execution that target the three important categories of data cache misses identified in Section 4.7. In this chapter we evaluate the impact of combining all three techniques and compare the performance against running the benchmarks sequentially. Then we investigate the impact of re-selecting parallel regions after applying our locality techniques. Finally, we show that our techniques enhance the scalability of TLS on up to 8-core CMPs.

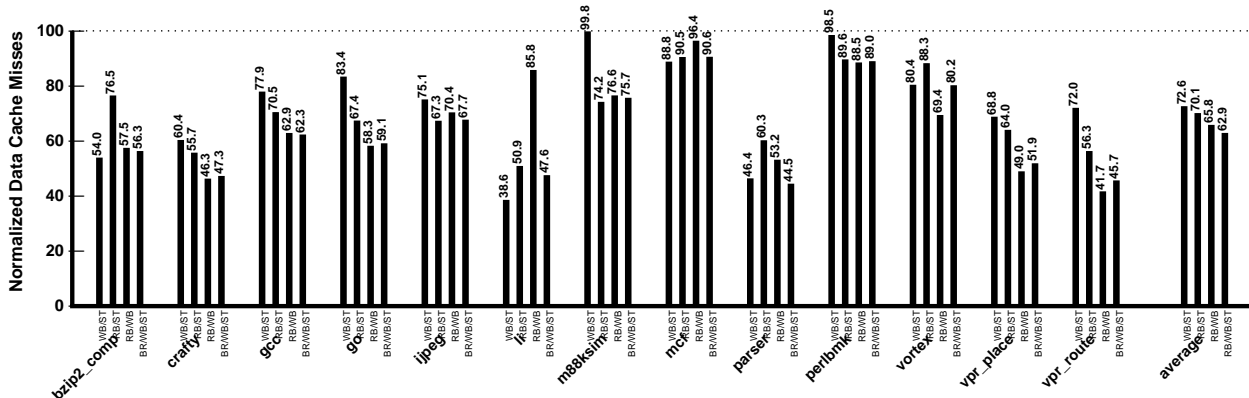
6.1 Performance Impact of Combining all Techniques

In Figure 6.1 we show the reduction in data cache misses in parallel regions for all combinations of the three techniques: exploiting read-based sharing by broadcasting all load misses (*RB*), exploiting write-based sharing through our technique for cache line self-invalidation and pushing (*WB*), and strided prefetching (*ST*).

Both bus and crossbar have 28% cache miss reduction with *WB/ST*, however crossbar has nearly 5% more misses than bus in three other cases due to the inefficiency of broadcasting with

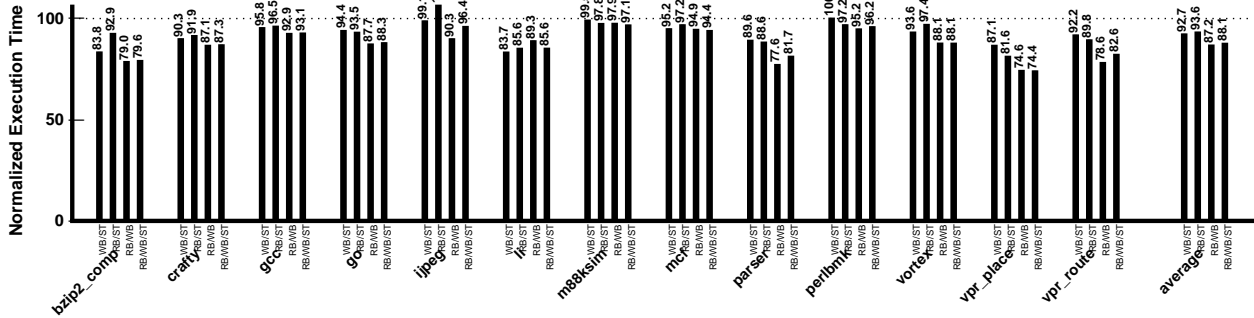


(a) Normalized data cache misses (bus).

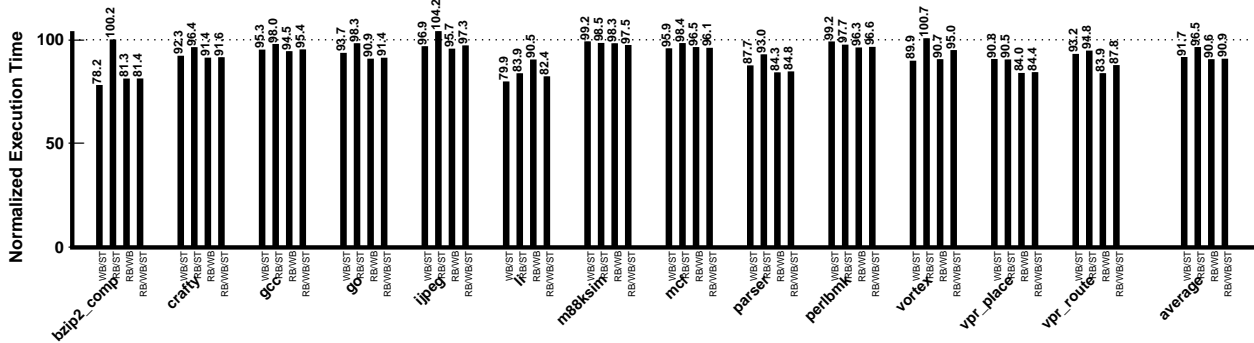


(b) Normalized data cache misses (crossbar).

Figure 6.1: Impact on the number of data cache misses within parallel regions of combining the three techniques, relative to the baseline model: exploiting read-based sharing (*RB*), write-based sharing (*WB*), and strided prefetching (*ST*).



(a) Normalized execution time (bus).



(b) Normalized execution time (crossbar).

Figure 6.2: Impact on the performance of parallel regions of combining the three techniques, relative to the baseline model: exploiting read-based sharing (*RB*), write-based sharing (*WB*), and strided prefetching (*ST*).

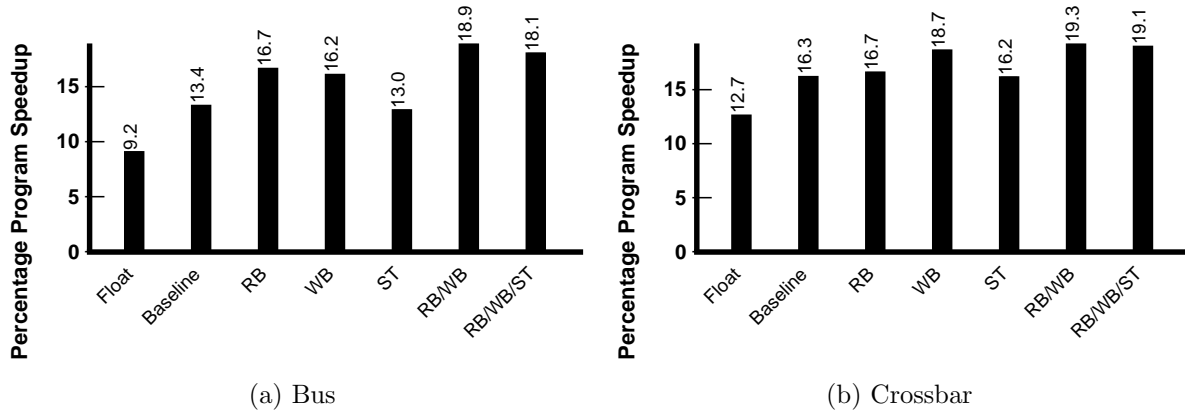


Figure 6.3: Summary of all our techniques, showing average program speedup of 13 SpecInt benchmarks relative to sequential execution. *Float* shows the performance of a “floating” sequential processor, *Baseline* includes a fixed sequential processor, *RB* exploits read-based sharing, *WB* exploits write-based sharing, and *ST* performs strided prefetching.

a crossbar interconnect.

Broadcast is the most effective miss-reduction technique for nine of the applications with a bus and eight with a crossbar. When our write-based technique is applied with a bus interconnect, BZIP2_COMP and PARSER have more than 20% decrease in cache misses. Three applications perform best with the invalidation scheme with a crossbar: BZIP2_COMP, PARSER and VORTEX. Strided prefetching provides a significant benefit for LI and MCF: LI has a huge 50% drop in the number of cache misses when stride prefetching is enabled (*RB/WB/ST*), while MCF has 4% less misses. Since the broadcast and stride schemes each target independent groups of misses, we expect that their respective impacts will be complementary. While previously we observed an average reduction in data cache misses in parallel regions of 27.7% for broadcast alone and 10.3% for stride alone, we see a 34.1% reduction when both are combined (*RB/ST*) as evidence of their complementary behaviour. This result is less than the sum of the individual reductions, indicating that some cache lines are impacted by both schemes over the full

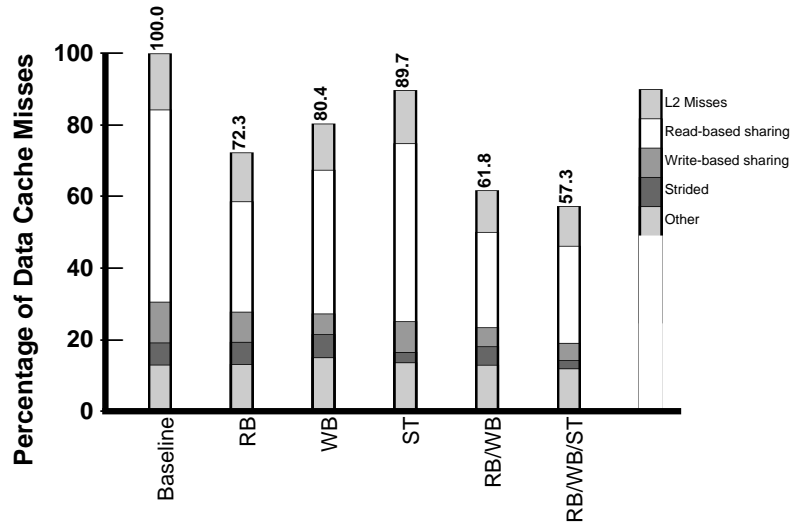


Figure 6.4: Impact on data cache misses of all our techniques within parallel regions, showing number of misses relative to the baseline. *Baseline* includes a fixed sequential processor, *RB* exploits read-based sharing, *WB* exploits write-based sharing, and *ST* performs strided prefetching.

execution (as opposed to our limited-window classification in Section 4.7). Since the technique for exploiting write-based sharing reduces traffic, it enables the traffic-limited strided prefetcher to further reduce misses. This is demonstrated by the 5% improvement of all three techniques (*RB/WB/ST*) over broadcast and strided prefetching alone (*RB/ST*).

Looking at the performance impact of combining the techniques, as shown in Figure 6.2, we see that the combination of read and write-based techniques (*RB/WB*) performs the best for eight applications for both bus and crossbar, with the combination of all three schemes performing the best for three applications. Unfortunately, it is evident that while the combination of all three schemes eliminates the most cache misses, the read/write-based combination actually performs the best on average. This indicates that the read/write-based schemes were unable to reduce traffic enough for the strided prefetcher to be effective. However, the read/write-based combination has still significantly improved the performance of parallel regions. With a bus

interconnect, performance is improved by nearly 10% or more for nine applications, and by an average of 12.8% across all applications. In the case with a crossbar interconnect, the performance of *WB/ST*, *RB/WB* and *RB/WB/ST* are indeed very close and are within around 1%. However *RB/ST* is 5% worse than the other three combinations, indicating that *WB* is most important for crossbar in which broadcasting does not yield any significant gain.

Figure 6.3 summarizes the performance impact on overall program of our locality techniques. The results are shown as program speedup relative to the original sequential execution of the un-modified binary. Our best scheme, *RB/WB* adds 5.5% program speedup to the baseline TLS hardware with a bus, and 3.3% in the case of crossbar. TLS baseline hardware has 13.4% and 16.3% speedup relative to sequential execution with a bus and a crossbar interconnect respectively, indicating that TLS performs better with a crossbar without our locality techniques. After applying our techniques, both bus and crossbar yield nearly 19% program speedup since cache locality is improved and hence interconnect contention is reduced.

In Figure 6.4 we show data cache miss patterns within parallel regions after applying our locality techniques with a bus interconnect. *RB* significantly reduces read-based sharing misses, while *WB* eliminates both read-based and write-based misses. *ST* reduces most of the strided misses. *RB* reduces some of the write-based misses since broadcasting also benefits any write miss that follows a read miss (the cache line is broadcasted to all processors after the read miss). As shown in Figure 6.2(a), *RB/WB* is the scheme that gives most performance gain and it eliminates a large amount of read-based and write-based miss. Although adding stride prefetching does not yield additional performance speedup, *RB/WB/ST* has the smallest number of data cache misses among all our schemes since it targets all three major miss categories. L2 cache misses have not been significantly reduced as our locality techniques do not intend to prefetch into the level-two cache.

6.2 Impact of Re-selecting Parallel Regions

So far, we have evaluated our techniques for improving cache locality on applications whose parallel regions were chosen based on their measured performance on baseline hardware support: the parallel regions which contribute the greatest product of speedup and coverage (which we call *gain*) are selected, as described in Chapter 3. However, it is possible that our techniques themselves could change the relative gains of parallel regions, resulting in an even better potential selection of parallel regions and enhanced performance. To the best of our knowledge, this crucial step in the evaluation of optimizations for TLS has not been taken previously.

In Figure 6.5 we show the impact of reselecting regions with a bus interconnect, showing only those applications for which the selection has changed significantly (as measured by the coverages of the changed parallel regions). We show program speedup relative to the sequential execution where B is the baseline, B/RW improves on the baseline by supporting both read and write-based techniques, and R/RW reselects parallel regions after applying those techniques. Note that the versions of the benchmarks used in this experiment are slightly different than those used in the rest of the thesis, because they allow us to easily change the selection of parallel regions through guarded code (rather than recompiling).

Applying our techniques to the original selection of parallel regions (B/RW) improves the program performance of BZIP2_COMP by 5%. Reselecting parallel regions for this application (R/RW) further improves program performance by 4%. For COMPRESS, LI and TWOLF, reselecting parallel regions makes the difference between achieving no speedup at all and achieving modest program speedups of 5%, 3% and 5% respectively. However, reselecting parallel re-

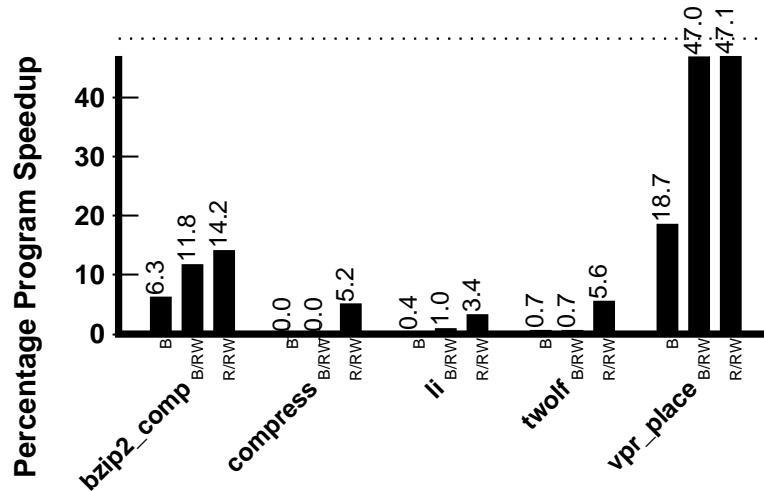
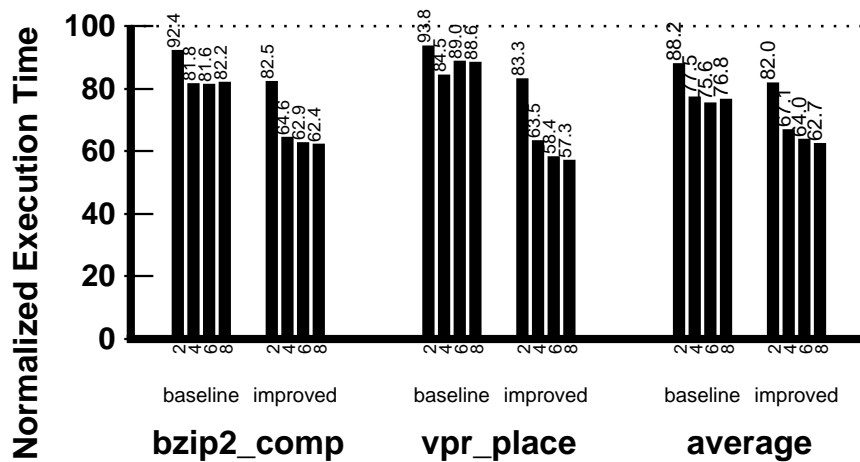


Figure 6.5: Impact of re-selecting parallel regions after applying our techniques with a bus interconnect, showing program speedup relative to the sequential execution: *B* is the baseline, *B/RW* implements both broadcast all load misses and aggressive writeback techniques, and *R/RW* reselects parallel regions after applying those techniques.

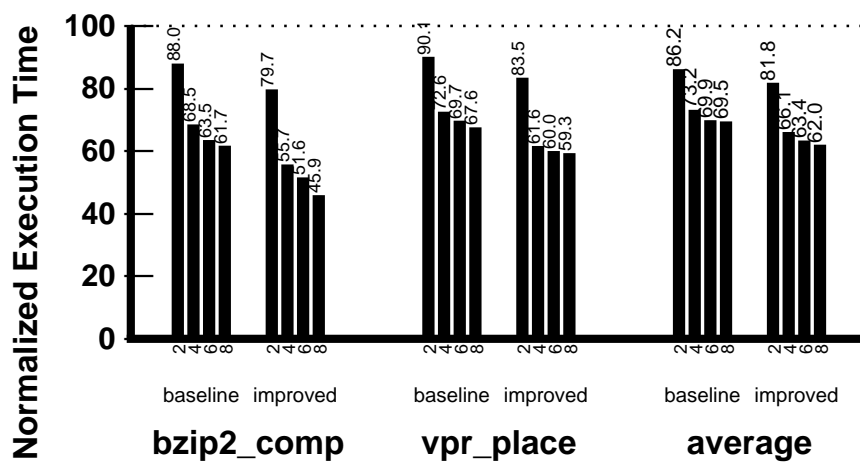
regions has no performance impact on VPR_PLACE, even though the selection itself did change significantly (by coverage). Overall, these results confirm that reselecting parallel regions can offer compelling additional performance improvements when evaluating an optimization of TLS execution.

6.3 Impact on Scalability

With baseline TLS hardware support, as we increase the number of processors the cache locality problem is exacerbated: more processors means more caches which in turn means decreased cache locality when compared with the original sequential execution. This trend is evident in Figure 6.6, where we show the region execution time for a varying number of processors (from two to eight). Looking at the average of all benchmarks for the baseline TLS support with a



(a) bus



(b) crossbar

Figure 6.6: Impact of our techniques for improved locality on the scalability of parallel regions, as we vary the number of processors from 2 to 8. The improvement is most pronounced for BZIP2_COMP and VPR_PLACE, but is also significant on average across all benchmarks.

bus interconnect, we see that increasing the number of processors to eight actually hinders performance (as compared with only using six processors). This negative trend is most pronounced in BZIP2_COMP and VPR_PLACE, as shown in Figure 6.6(a).

After applying our techniques for improving locality (labeled *improved*, which exploits both read-based and write-based sharing patterns), both BZIP2_COMP and VPR_PLACE show improved scaling, as well as significantly improved performance. Looking at the average case, we see that increasing processors from six to eight improves performance (although modestly). The amount of improvement over the baseline case grows with the number of processors (6.2%, 10.4%, 11.4%, and 14.1% for 2, 4, 6, and 8 processors respectively), demonstrating that the importance of dealing with cache locality issues for TLS grows with the number of processors.

On the other hand, the TLS baseline with a crossbar interconnect is able to scale well up to eight processors, as shown in Figure 6.6(b), while our techniques further improve the performance of parallel regions.

Chapter 7

Conclusions

Thread-Level Speculation (TLS) is a promising way to exploit chip multiprocessors (CMPs) to improve the performance of an individual program through speculative parallelization. However, the cache locality of the original program is significantly disrupted by TLS execution, resulting in nearly a four-fold increase in the data cache miss rate. This dissertation proposes techniques which improve cache locality and hence performance for TLS systems. These techniques are evaluated with SPEC integer benchmarks. In this section, we review the contributions of this dissertation and discuss possible directions for future work.

7.1 Contributions

This thesis makes the following contributions. First, we provide a thorough classification of the cache locality problem for TLS, and use this classification to divide and conquer the problem. Second, we perform a detailed evaluation of techniques for addressing read-only sharing, write-based sharing, and strided-based miss patterns. We show that these simple techniques can significantly reduce the number of cache misses for TLS programs and improve overall per-

formance. Third, we demonstrate the importance of re-selecting parallel regions *after* applying our techniques for improving cache locality, as opposed to only evaluating the original selection which is biased towards baseline TLS hardware support. Finally, we show that our techniques significantly improve the ability of TLS execution to scale to larger numbers of processors.

Our investigation into the TLS cache locality problem revealed that scheduling the sequential portion of execution to a single processor is much better for cache locality than a floating sequential processor. We also discovered that given this proper scheduling instruction cache locality was not an issue, and that the majority of performance problems come from the data cache locality and interconnect traffic during parallel execution. Finally, we observed that a vast majority of misses were for miss patterns exhibiting read-only sharing, with write-based sharing and stride-based patterns being the next two most significant.

These observations motivated us to suggest several schemes for improving data cache locality. With respect to baseline TLS hardware support, we can further reduce cache misses by 38.2% with a bus and 34.2% with a crossbar, improve parallel region and program performance by 12.8% and 5.5% respectively for a bus, 9.5% and 3% for a crossbar, through our techniques for exploiting read and write based sharing. While stride-based prefetching can significantly reduce data cache misses, we found that the additional traffic incurred during parallel regions is prohibitive. We showed the importance of re-selecting parallel regions after applying any optimization with a bus interconnect: for five applications a different selection of parallel regions resulted in a significant additional performance improvement, making the difference between achieving no speedup at all and achieving modest program speedups. Finally, we demonstrated that our techniques facilitate scaling, and that the importance of dealing with cache locality issues for TLS grows with the number of processors.

Extracting thread-speculative parallelism from general purpose programs is challenging. However, by maximizing the efficiency of every aspect of the system, including repairing cache locality, we can use CMPs to automatically extract significant speculative parallelism from a broad range of applications.

7.2 Future Work

Improving the cache locality for TLS is a relatively new topic and there is no other similar work as far as we know. There is plenty of research to be done beyond this dissertation, including the following,

Future compiler research: Although we have shown our hardware techniques are able to eliminate locality misses and improve performance significantly, we believe that since the compiler has the knowledge of the program structure, it could be used to analyze the regions and suggest the hardware with the best potential locality optimizations. Compiler transformations for improving locality on speculative parallel would be beneficial, especially for array based strided accesses.

Investigate into the possibility of selectively activating a combination of locality technique for individual region: Since the cache access patterns for each parallel region can be very different, it is highly possible that any particular locality technique would perform well in one region while hurting the performance of another region. The invocation path of the region may also have an impact on cache behaviour, and affect the locality techniques. A system that is able to monitor cache behaviour online and select the optimal combination of optimizations for each region invocation would be beneficial.

Evaluation of our locality techniques on other multithreaded systems: It would be interesting to see the performance impact of our techniques on other hardware-based TLS systems [17, 18] (as described in Chapter 2) and helper threads prefetching schemes [28, 46].

Bibliography

- [1] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the third International Conference on High-Performance Computer Architecture*, February 1997.
- [2] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st International Symposium on Microarchitecture*, December 1998.
- [3] J. Gregory Steffan Antonia Zhai, Christopher B. Colohan and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [4] J. Gregory Steffan Antonia Zhai, Christopher B. Colohan and Todd C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, March 2004.
- [5] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. In *IEEE Transactions on Computers, Volume 44 , Issue 5*, May 1995.

- [6] S. E. Breach, T. N. Vijaykumar, Sridhar Gopal, J. E. Smith, and G. S. Sohi. Single-program speculatives multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. Technical Report RC19928, IBM Research Division, T.J. Watson Research Center, February 1995.
- [7] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, November 2001.
- [8] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, 1994.
- [9] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. Technical report. <http://www.spechbench.org>.
- [10] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, 1993.
- [11] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, 1993.
- [12] Yiannakis Sazeides Eric Rotenberg, Quinn Jacobson and Jim Smith. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.
- [13] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, May 1996.

- [14] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, December 1993.
- [15] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, 1992.
- [16] S. Breach G. S. Sohi and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.
- [17] Sridhar Gopal, T.N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.
- [18] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, and Kunle Olukotun. The stanford hydra cmp. In *IEEE MICRO Magazine*, March 2000.
- [19] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [20] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture*, 2003.
- [21] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, 1997.
- [22] Stefanos Kaxiras, Stein Gjessing, and James R. Goodman. A study of three dynamic approaches to handle widely shared data in shared-memory multiprocessors. In *ICS '98*:

- Proceedings of the 12th international conference on Supercomputing*, pages 457–464, New York, NY, USA, 1998. ACM Press.
- [23] Iffat H. Kazi and David J. Lilja. Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors. In *Proceedings of the 12th international conference on Supercomputing*, 1998.
- [24] Dongkeun Kim, Steve Shih wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. Physical experimentation with prefetching helper threads on intel’s hyper-threaded processors. In *CGO ’04: Proceedings of the international symposium on Code generation and optimization*, page 27, Washington, DC, USA, 2004. IEEE Computer Society.
- [25] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.
- [26] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA ’00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 139–148, New York, NY, USA, 2000. ACM Press.
- [27] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *ISCA ’95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 48–59, New York, NY, USA, 1995. ACM Press.
- [28] Chi-Keung Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51, July 2001.

- [29] Pedro Marcuello and Antonio Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*, June 1999.
- [30] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, 1991.
- [31] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 62–73, New York, NY, 1992. ACM Press.
- [32] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [33] Antonio Gonzalez Pedro Marcuello and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*, July 1998.
- [34] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops With Privatization and Reduction Parallelization. In *Proceedings of PLDI '95*, pages 218–232, June 1995.
- [35] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.
- [36] Amir Roth and Gurindar S. Sohi. A quantitative framework for automated pre-execution thread selection. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.
- [37] J. Gregory Steffan. *Hardware Support for Thread-Level Speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, April 2003.

- [38] J. Gregory Steffan, Christopher B. Antonia, Colohan Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.
- [39] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, February 2002.
- [40] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA 22*, pages 392–403, June 1995.
- [41] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, January 1998.
- [42] Zhenlin Wang, Doug Burger, Steven K. Reinhardt, Kathryn S. McKinley, and Charles C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.
- [43] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.
- [44] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.
- [45] Xiaotong Zhuang and Hsien-Hsin S. Lee. A hardware-base cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 32nd International Conference on Parallel Processing*, October 2003.
- [46] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, July 2001.