Improving Cache Locality for Thread-Level Speculation

Stanley L.C. Fung and J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, M5S 3G4 Canada
{sfung,steffan}@eecg.toronto.edu

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 (i.e., four-fold greater cache capacity).*

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%, improve performance by 12.8%, and significantly improve scalability—further enhancing the feasibility of TLS as a way to capitalize on future CMPs.

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 (combinable, 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. One promising possibility for automatically-parallelizing general-purpose programs is *Thread-Level Speculation* (TLS) [6, 7, 9, 14, 15] 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.

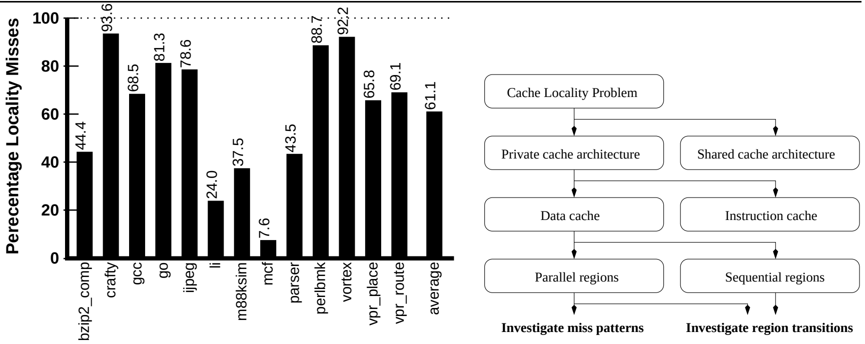
1.1 The TLS Cache Locality Problem

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. This tension between parallelism and locality is a key challenge for a single program to fully exploit the resources of a CMP.

To demonstrate the TLS cache locality problem, in Figure 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 (the details of our experimental framework are available in Section 2.1). As expected, the miss rate increases significantly in every case, with M88KSIM, PERLBMK, and VPR_ROUTE suffering the most. Overall, the average data cache miss rate for the TLS versions is nearly four times that of the original sequential versions. Intuitively, this makes sense since four processors are now accessing data that was originally accessed by only one processor. As further evidence that the culprit is reduced cache locality, in Figure 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.

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 miss rates.



(b) Percentage locality misses.

(c) Our investigation.

Figure 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); (c) our investigation of the cache locality problem.

On average 61.1% of all misses are such *locality misses*, indicating that locality has indeed been significantly disrupted.

In this paper we present a thorough investigation of the cache locality problem for TLS execution, as summarized by Figure 1(c). We discuss shared cache architectures, such as when an SMT processor supports TLS, although locality problems are the result of private cache TLS support—hence we focus only on private cache support in this paper. 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 behavior temporally, into regions (i) where 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. We find that read-only and write-based sharing patterns as well as strided access patterns represent the vast majority of misses during parallel execution. The results of this classification suggest several techniques for improving TLS cache locality which we evaluate in this paper.

1.2 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 [3, 16], in particular employing software transformations that adjust access patterns and data layout to improve locality. In this paper 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. Memory latency can be tolerated through prefetching [1, 4, 5]; since TLS itself has prefetching effects, many schemes have been proposed to speculatively execute *helper threads* which prefetch or precompute for a main thread [8, 12]. Most closely-related to this work, Brown *et al.* [2] propose broadcasting load-missed cache lines to all private data caches to improve the performance of speculative precomputation on a CMP. We find that similar support works well to improve locality and performance for TLS.

1.3 Contributions

This paper 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 performance. Finally, we show that these techniques significantly improve the ability of TLS execution to scale to larger numbers of processors.

2 Underlying Support for TLS

Before we investigate the cache locality problem, in this section we describe the support for TLS that this

Table 1. Simulation 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
Bus 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

work is based on, including both hardware and compiler support. While this study is within the context of a particular TLS implementation, the techniques that we suggest for improving cache locality would be applicable to other TLS systems as well.

Compiler Support: Our compiler infrastructure decides how to divide a program into speculative threads [15]. For now we consider only loops, since loops comprise a significant fraction of overall execution time. Using profile information, the compiler decides to speculatively parallelize the set of non-nested loops which maximize performance assuming baseline hardware support on a four-processor CMP. The compiler then transforms these loops by performing loop unrolling, inserting new instructions that interface with the underlying TLS hardware to manage speculative threads and perform synchronization, and scheduling code within synchronized portions to maximize parallel overlap. The compiler outputs modified `C` code which is then compiled with `gcc` 2.95.2 using the "-03" flag to produce optimized MIPS binaries.

Hardware Support: The underlying hardware support for TLS must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. Our underlying hardware support implements these features by using the data caches and an extended version of standard invalidation-based cache coherence [15]. In a nutshell, the extended coherence scheme tracks which cache lines have been speculatively loaded or modified, and piggybacks a sequence number on coherence messages to detect when a speculative thread has violated a data dependence. Cache lines which have been speculatively modified may not be evicted from the data cache until the speculative thread commits.

2.1 Experimental Framework

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, but modernized to have a 128-entry reorder buffer. We simulate a system with four processing cores, where each has its own physically private data and instruction caches, connected to a unified second level cache by a bus. 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 1.

We report results for all of the SPECint95 and SPECint2000 benchmarks 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 study. For each benchmark, after skipping over the initialization phases, we simulate approximately a billion instructions using the first input in the `ref` input set. Note that we have separated the compression phase of BZIP2, and the place and route phases of VPR.

3 Classifying TLS Cache Misses

Since the cache locality problem for TLS 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 1(c).

TLS support has been proposed for both shared [15] and private [6,7,15] cache architectures, and both have interesting cache behavior. For a shared cache architecture, the hardware-supported threads of execution (such as independent processors or the contexts of a *simultaneously-multithreaded* processor (SMT)) 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. In the private-cache case, the cache locality enjoyed by the original sequential program has definitely been disrupted (as demonstrated in Figure 1), hence we focus our efforts on this area.

3.1 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 behavior of a TLS program will therefore change significantly as the

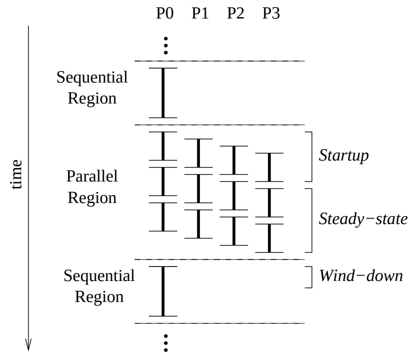
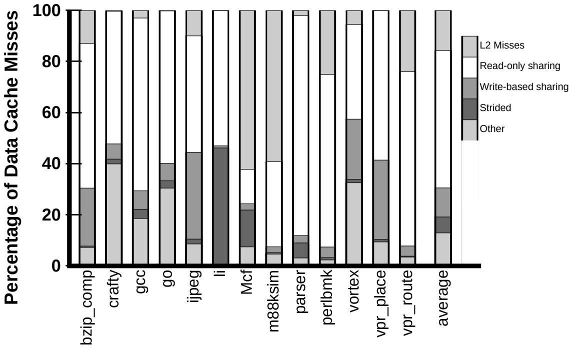


Figure 2. Execution stages of a TLS program.

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

(a) *Miss pattern breakdown*



(b) *Detailed miss pattern breakdown*

Figure 3. Data cache miss patterns within parallel regions.

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 2. These stages are repeated throughout the execution of the program, occurring for each dynamic parallel region instance, and exhibiting the following behavior.

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. Preliminary results showed that these *startup* misses do exist, but that addressing them directly would not provide significant performance gains.

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 3.3.

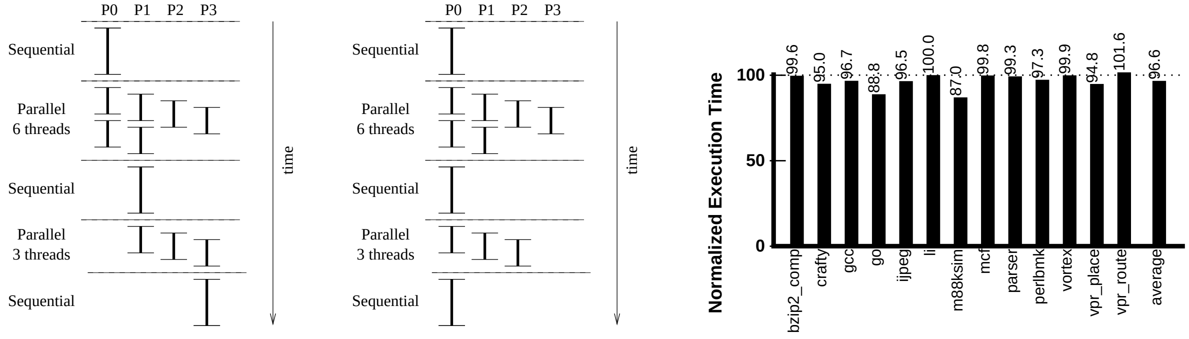
Wind-down: At the end of a parallel region we transition back to sequential execution, and expect that this sequential region will initially suffer a significant number of locality misses for data resident in the caches of temporarily inactive processors. In Section 4 we demonstrate how scheduling the sequential region can impact the number of locality misses observed during this *wind-down* stage.

3.2 Data and Instruction Cache Misses

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 behavior. Hence the memory hierarchy typically has separate first level caches for instructions and data, and we are obliged to investigate them separately. In an early experiment to clarify the potential benefits of improving cache locality for TLS execution, we modeled perfect private instruction and data caches where every reference is considered to be a hit. We found that the ideal data cache improved the performance of parallel regions by 23%, while the ideal instruction cache provided less than 1% improvement. This result is intuitive, since so far only loops have been speculatively parallelized in our benchmark applications; parallelized loops will have very good instruction cache locality, with the exception of possible cold misses during the *startup* stage of each parallel region. While there are simple methods for eliminating some of this small overhead, we do not discuss them further here and instead focus on improving cache locality for data.

3.3 Miss Patterns in Parallel Regions

To guide our efforts to improve cache locality for parallel regions, we analyzed traces of data cache miss addresses and observed the following five most common patterns of miss: (i) second-level cache (L2) misses; (ii) read-only sharing misses, where a cache line is read (but not written) by multiple processors; (iii) write-based sharing misses, where a cache line is written by at least one processor and possibly read by other processors; (iv) strided miss patterns, where the addresses



(a) Floating.

(b) Fixed.

(c) Program performance: fixed vs. floating.

Figure 4. 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. Finally, (c) shows the performance impact of a fixed sequential processor relative to floating.

of missing cache lines progress by a fixed stride; and (v) the remaining misses (*other*), for which we could find no pattern and are likely conflict and capacity misses.

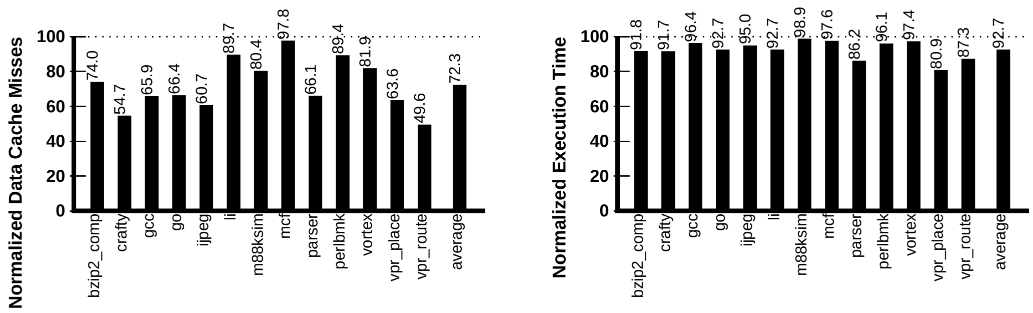
Classifying miss patterns based on a trace must be done carefully. To decide whether a set of misses belong in a given category, we analyze all of the misses within a fixed-size sliding window of 1000 cycles of the trace (experimentation showed that larger window sizes up to 16000 cycles did not significantly alter the classification). 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 but then read by multiple processors. For simplicity, we deal with overlap in our classification by attempting to fit each miss within the described categories in priority order. Figure 3(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 3(b).

From the figure, we observe that data cache misses within parallel regions exhibit interesting behavior. 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 addressed with known techniques for prefetching [1, 4, 5, 13]. Cache lines that exhibit read-only sharing are the most common, representing 53.7% of all misses. This is promising, since under TLS read-only cache lines are much easier to deal with than those which are modified. We address this category of misses in Section 5. Cache lines involved in *write-based sharing* represent 11.4% of misses, hence we address them in Section 6. Strided accesses comprise another 6.2% of misses, and we address them in Section 7. Finally, cache lines in the *other* category represent the remaining 13.0% of misses.

In the remainder of this paper 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.

4 Scheduling the Sequential Region

In the *wind-down* stage (as illustrated in Figure 2), 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 section 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 4. First, we consider a “floating” sequential processor, where the processor which executed the last 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.



(a) Number of data cache misses.

(b) Normalized execution time.

Figure 5. Impact of broadcasting all load misses on parallel 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 4(c) we show the program performance 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 region and the subsequent sequential region: three applications perform more than 5% better, and all benchmarks perform on average 3.4% better. While this improvement 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 Exploiting Read-Only Sharing

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 yet no speculative thread writes. 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.

5.1 Broadcasting for all Load Misses

The simplest scheme for addressing read-only 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 speculative 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). It is important to understand that this is not the same thing as a snoopy update, since every speculatively-read cache line will be pushed to every cache, regardless of whether each cache already has a copy.

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. Figure 5 shows the performance of this broadcasting scheme relative to our baseline model. On average across all benchmarks, this simple broadcasting scheme eliminates 27.7% of the data cache misses in speculative regions on average, and significantly more for several applications. This technique also improves execution time for every application, by 7.3% on average.

Throttling Broadcast: Although the scheme proposed above 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 subsequently 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 3.3) 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, indicating that pollution is not a problem for this broadcast approach.

6 Exploiting Write-Based Sharing

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.

In Figure 3 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 6(a). 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. Through experimentation, we found that limiting the size of each of the three structures to eight entries is sufficient.

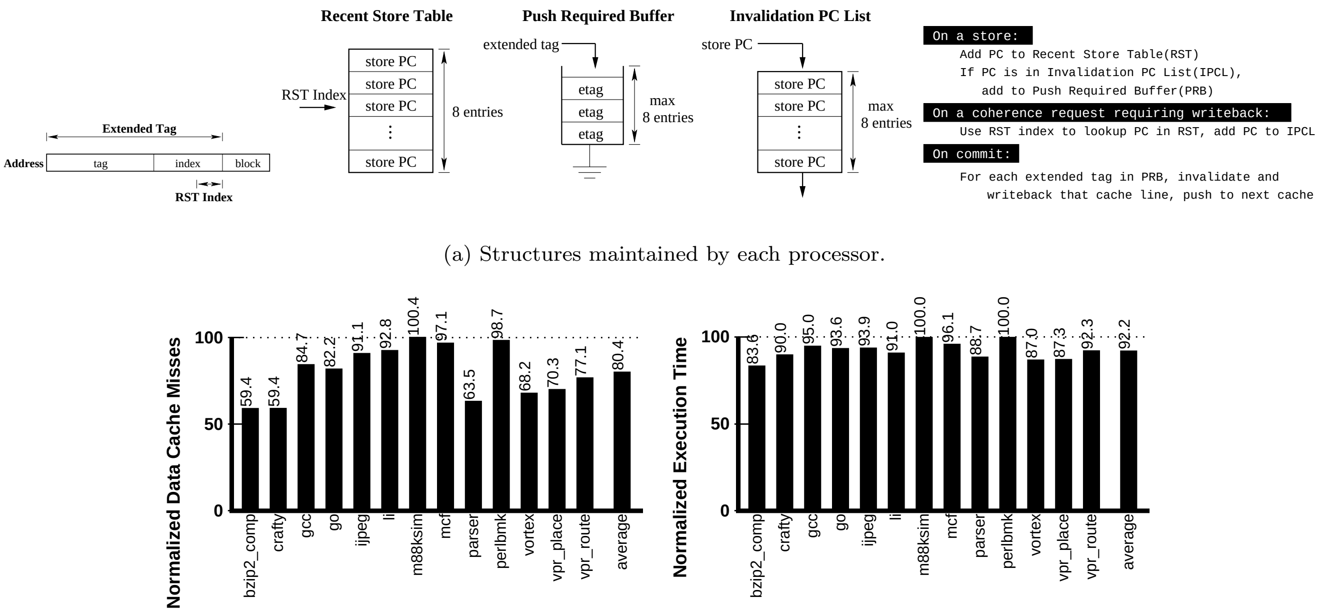
The operation of our technique is as follows. When a store executes, the store PC is saved in the RST (using three bits of the store address as an index). If that PC is currently in the IPCL, then it has been identified as being involved in write-based sharing in the past, and hence the extended tag 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 paper, 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 [11], although our technique does not need to implement versioning numbers to decide which blocks to self-invalidate. Furthermore, last-touch prediction [10] cannot be used in our approach since modified cache lines may not be propagated until the speculative thread commits.

Figure 6 shows the performance of our write-based sharing technique relative to our baseline model. On average across all benchmarks, 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. 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.

7 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 3.3, 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



```
On a store:
    Add PC to Recent Store Table(RST)
    If PC is in Invalidation PC List(IPCL),
       add to Push Required Buffer(PRB)
On a coherence request requiring writeback:
    Use RST index to lookup PC in RST, add PC to IPCL
On commit:
    For each extended tag in PRB, invalidate and
       writeback that cache line, push to next cache
```

(a) Structures maintained by each processor.

(b) Normalized data cache misses.

(c) Normalized execution time.

Figure 6. Our technique for exploiting write-based sharing.

access patterns have been well studied [1,4,5,13], there has been relatively little investigation into how stride-based prefetching interacts with TLS execution.

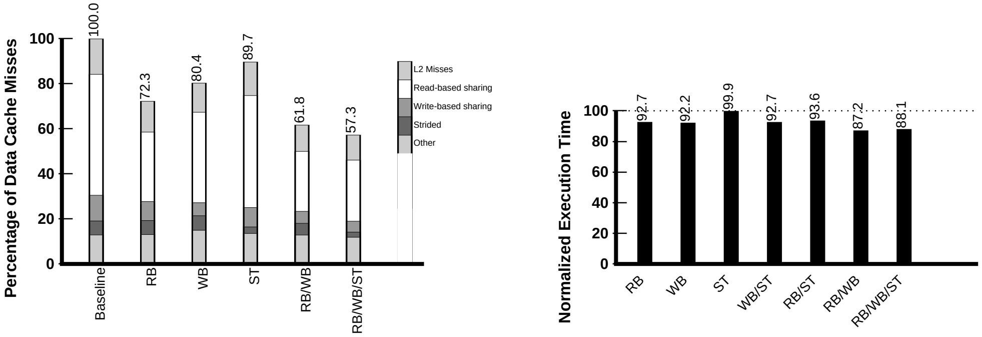
To evaluate the maximum potential impact of stride-based prefetching on TLS execution, we model an aggressive *adaptive* stride prefetcher [4] in each processor (fully associative, 512 entries, LRU replacement, queues 16 prefetches when a stride is recognized after 3 instances, and throttles the issuing of the prefetches to avoid traffic bursts). We do not allow prefetching beyond the L2, since this would lead to an unfair comparison with the sequential execution. On average, we found that strided prefetching has no significant performance impact—we do not show the full results here for this reason and due to space limitations. Prefetching strided accesses reduced data cache misses by nearly 20% or more for three applications, and by an average of 10% across all applications—indicating that we successfully eliminated most stride-based misses as identified in Section 3.3. However, the insignificant performance gained indicates that the benefits of reduced cache misses are overwhelmed by the increase in interconnect traffic.

8 Combining the Techniques

We have proposed three techniques for improving cache locality for TLS execution that target the three distinct categories of data cache misses identified in Section 3.3. In this section we evaluate the impact of combining all three techniques. In Figure 7(a) we show data cache miss patterns within parallel regions after applying the locality techniques. The read-based (*RB*) technique significantly reduces read-based sharing misses, as well as some write-based sharing misses: for a cache line which was just broadcast, a subsequent write to that line on another processor need only obtain obtain ownership of the line and not a copy. Similarly, the write-based (*WB*) technique eliminates write-based misses as well as some read-based misses. *ST* mainly reduces strided misses. The combination of *RB/WB* is indeed complementary, further reducing misses by an additional 10.5% over *RB* alone. Since it targets all three major miss categories, *RB/WB/ST* provides the greatest reduction in data cache misses of 42.7%.

Figure 7(b) shows the performance impact of combining the techniques. 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, by an average of 12.8% across all applications.

Figure 8 summarizes the performance impact of all



(a) Normalized data cache misses.

(b) Normalized execution time.

Figure 7. Impact on (a) data cache misses and (b) performance of all three techniques within parallel regions relative to the baseline. The baseline includes a fixed sequential processor.

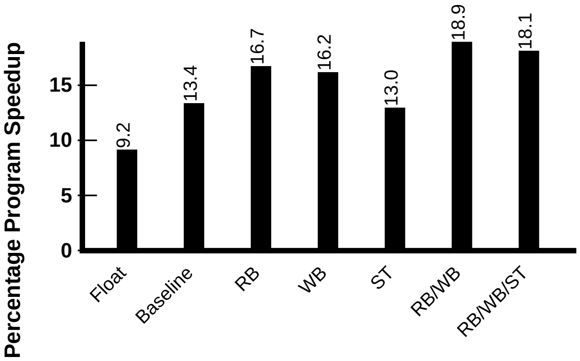


Figure 8. Summary of all techniques, showing program speedup relative to sequential execution. Float shows the performance of a "floating" sequential processor, Baseline includes a fixed sequential processor.

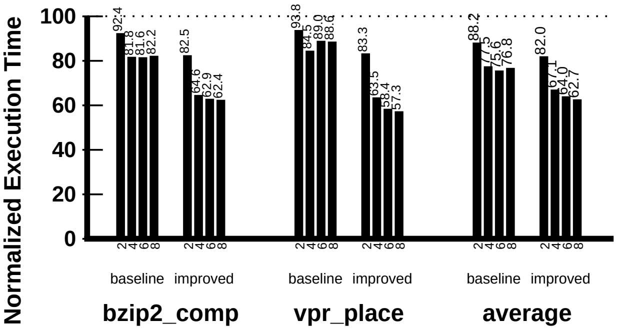


Figure 9. Impact of the techniques for improved locality on the scalability of parallel regions, as we vary the number of processors from 2 to 8.

techniques on overall program performance. The results are shown as program speedup relative to the original sequential execution of the un-modified binary. The baseline hardware support (including a fixed sequential processor) provides a 13.4% program speedup relative to sequential execution. The combination of read and write-based techniques (RB/WB) further improves program performance over the baseline by 5.5% on average across all benchmarks, demonstrating that the performance impact of the read and write-based techniques is also complementary. We again observe that the additional traffic of the stride-based technique (*RB/WB/ST*) is still detrimental to overall performance, even though it does further reduce cache misses.

8.1 Impact on Scalability

Without using the techniques proposed in this paper, 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 9, where we show the region execution time for a varying number of processors (from two to eight). Looking at the average of all benchmarks with baseline TLS support, 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.

After applying the 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.

9 Conclusions

Thread-Level Speculation (TLS) is a promising way to exploit chip multiprocessors (CMPs) and 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. Our investigation shows 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 is not an issue, and that the majority of performance problems come from data cache locality and interconnect traffic during parallel execution. Finally, we observed that a vast majority of misses are for miss patterns exhibiting read-only sharing, with write-based sharing and stride-based patterns being next most significant. These observations suggested several schemes for improving data cache locality that we implement. With respect to baseline TLS hardware support, we can further reduce cache misses by 38.2%, improve parallel region and program performance by 12.8% and 5.5% respectively 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. 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.

References

[1] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. In *IEEE Transactions on Computers*, volume 44, May 1995.

[2] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen. Speculative precomputation on chip multiprocessors. In *Proceedings of MEAC 6*, November 2001.

[3] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of ASPLOS 6*, San Jose, California, 1994.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of ICPP 93*, 1993.

[5] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of MICRO 25*, 1992.

[6] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of HPCA 4*, February 1998.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of ASPLOS 8*, October 1998.

[8] D. Kim, S. S. wei Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proceedings of CG 04*, 2004.

[9] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[10] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of ISCA 27*, 2000.

[11] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of ISCA 22*, 1995.

[12] C. Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *Proceedings of ISCA 28*, July 2001.

[13] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of ISCA 21*, April 1994.

[14] A. G. Pedro Marcuello and J. Tubella. Speculative multithreaded processors. In *Proceedings of Supercomputing 12*, July 1998.

[15] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 2005.

[16] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of PLDI 91*, 1991.