Improving Value Communication for Thread-Level Speculation

J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{*steffan,colohan,zhaia,tcm*}@*cs.cmu.edu*

Abstract

Thread-Level Speculation (TLS) allows us to automatically parallelize general-purpose programs by supporting parallel execution of threads that might not actually be independent. In this paper, we show that the key to good performance lies in the three different ways to communicate a value between speculative threads: speculation, synchronization, and prediction. The difficult part is deciding how and when to apply each method.

This paper shows how we can apply value prediction, dynamic synchronization, and hardware instruction prioritization to improve value communication and hence performance in several SPECint benchmarks that have been automatically-transformed by our compiler to exploit TLS. We find that value prediction can be effective when properly throttled to avoid the high costs of misprediction, while most of the gains of value prediction can be more easily achieved by exploiting silent stores. We also show that dynamic synchronization is quite effective for most benchmarks, while hardware instruction prioritization is not. Overall, we find that these techniques have great potential for improving the performance of TLS.

1 Introduction

Microprocessors which can simultaneously execute multiple parallel threads are becoming increasingly commonplace. Processors such as the Sun MAJC [34], IBM Power4 [18], and the Sibyte SB-1250 [8] are *single-chip multiprocessors* (CMPs), while the Alpha 21464 was designed to support *simultaneous-multithreading* [36]. Using this multithreaded hardware to improve the throughput of a workload is straightforward, but improving the performance of a single application requires parallelization.

How can we parallelize all of the applications that we care about? Writing parallel software can be a daunting task; we would much rather have the compiler parallelize our code for us. Traditionally, compilers have parallelized by proving that potential threads are independent [3, 17, 33]—but this is extremely difficult if not impossible for many general purpose programs due to their complex data structures and control flow, and use of pointers and runtime inputs. One promising alternative for overcoming this problem is *Thread-Level Speculation* (**TLS**) [2, 7, 14, 15, 16, 20, 23, 26, 32, 35] which allows the compiler to create parallel threads without having to prove that they are independent.

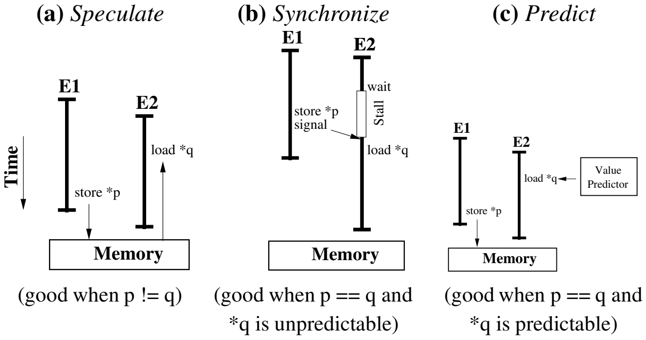


Figure 1. A memory value may be communicated between two epochs (E1 and E2) through (a) speculation, (b) synchronization, or (c) prediction.

1.1 The Importance of Value Communication for Thread-Level Speculation

In the context of TLS, value communication refers to the satisfaction of any true (read-after-write) dependence between *epochs* (sequential chunks of work performed speculatively in parallel). From the compiler's perspective, there are two ways to communicate the value of a given variable. First, the compiler may speculate that the variable is not modified (Figure 1(a)). However, if at run-time the variable actually *is* modified then the underlying hardware ensures that the misspeculated epoch is re-executed with the proper value. This method only works well when the variable is modified infrequently, since the cost of misspeculation is high. Second, if the variable is frequently modified, then the compiler may instead synchronize and forward[1] the value between epochs (Figure 1(b)). Since a parallelized region of code will contain many variables, the compiler will employ a combination of speculation and synchronization as appropriate.

To further improve upon static compile-time choices between speculating or synchronizing for specific memory accesses, we can exploit dynamic run-time behavior to make value communication more efficient. For example, we might exploit a form of *value prediction* [2, 13, 22, 24, 28, 29, 37], as illustrated in Figure 1(c). To get a sense of the potential upside of enhancing value communication under TLS, let us briefly consider the ideal case. From a performance perspective, the ideal case would correspond to a value predictor that could perfectly predict the values of any inter-thread dependences. In such a case, speculation

[1] This is also known as *doacross* [10, 27] parallelization.

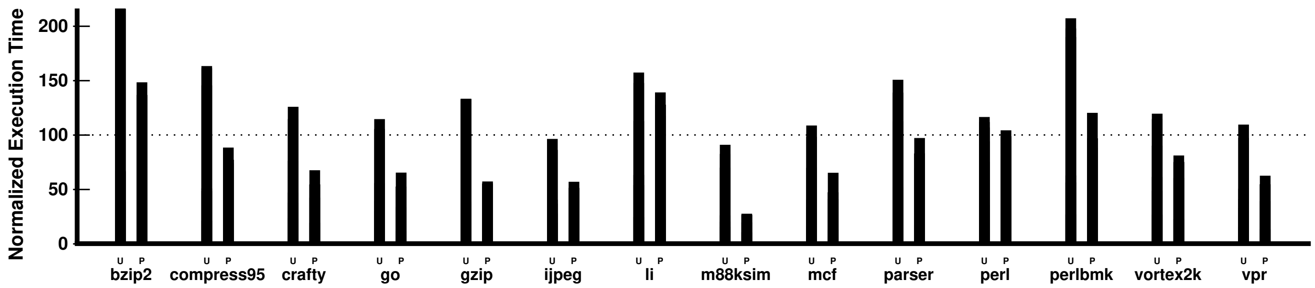


Figure 2. Potential impact of optimizing value communication. Relative to the normalized, original sequential version, U shows the unoptized speculative version and P shows perfect prediction of all inter-thread data dependences.

would never fail and synchronization would never stall. While this perfect-prediction scenario is unrealistic, it does allow us to quantify the potential impact of improving value communication in TLS. Figure 2 shows the impact of perfect prediction on several speculatively-parallelized SPECint [9] benchmarks, running on a 4-processor CMP that implements our TLS scheme [32] (details are given later in Section 3.2). Each bar shows the total execution time of all speculatively-parallelized regions of code, normalized to that of the corresponding original sequential versions of these same codes. As we see in Figure 2, efficient value communication often makes the difference between speeding up and slowing down relative to the original sequential code. Hence this is clearly an important area for applying compiler and hardware optimizations.

1.2 Techniques for Improving Value Communication

Given the importance of efficient value communication in TLS, what solutions can we implement to approach the ideal results of Figure 2? Figure 1 shows the spectrum of possibilities: i.e. speculate, synchronize, or predict. In our baseline scheme, the compiler synchronizes dependences that it expects to occur frequently (by explicitly "forwarding" their values between successive epochs), and speculates on everything else. How can we use hardware to improve on this approach? Hardware support for efficient *speculation* has already been addressed in a number of papers on TLS [2, 6, 14, 15, 16, 20, 23, 26, 32, 35]. Therefore our focus in this paper is how to exploit and enhance the remaining spectrum of possibilities (i.e. *prediction* and *synchronization*) such that they are complementary to speculation within TLS. In particular, we explore the following techniques:

Value Prediction: We can exploit *value prediction* by having the consumer of a potential dependence use a predicted value instead, as illustrated in Figure 1(c). After the epoch completes, it will compare the predicted value with the actual value; if the values differ, then the normal speculation recovery mechanism will be invoked to squash and restart the epoch with the correct value. We explore using value prediction as a replacement for both *speculation* and *synchronization*. In the former case (which we refer to later as "*memory value prediction*"), successful value prediction avoids the cost of recovery from an unsuccessful speculative load. In the latter case (which we refer to later as "*forwarded value prediction*"), successful prediction avoids the need to stall waiting for synchronization. Because the implementation issues and performance impact differ for these two cases, we evaluate them separately.

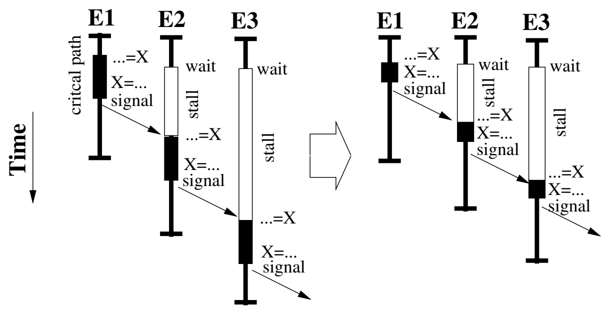


Figure 3. Reducing the critical forwarding path.

Silent Stores: An interesting program phenomenon that was recently discovered [21] is that many stores have no real side-effect since they overwrite memory with the same value that is already there. These stores are called *silent stores*, and we can exploit them when they occur to avoid failed speculation. Although one can view silent stores as a form of value prediction, the mechanism to exploit them is radically different from what is shown in Figure 1(c) since the changes occur with the *producer* of a communicated value, rather than the consumer.

Hardware-Inserted Dynamic Synchronization: In cases where the compiler decided to speculate that an ambiguous store/load dependence between speculative threads was not likely to occur, but where the dependence does in fact occur frequently and the communicated value is unpredictable, the best option would be to explicitly synchronize the threads (Figure 1(b)) to avoid the full cost of failed speculation. However, since the compiler did not recognize that such synchronization would be useful, another option is for the *hardware* to automatically switch from speculating to synchronizing when it dynamically detects such bad cases.

Reducing the Critical Forwarding Path: Once synchronization is introduced to explicitly forward values across epochs, it creates a dependence chain across the threads that may ultimately limit the parallel speedup. We can potentially improve performance in such cases by using scheduling techniques to reduce the critical path between the first use and last definition of the dependent value, as illustrated in Figure 3. We implement both compiler and hardware methods for reducing the critical forwarding path.

1.3 Contributions and Overview

This paper makes the following contributions. First, we evaluate a comprehensive set of techniques for enhancing value communication within a system that supports thread-level speculation, and demonstrate that many of them can result in significant performance gains. While we evaluate these techniques within the context of our own implementation of TLS, we expect to see similar trends within other TLS environments since the results are largely dependent on application behavior rather than the details of how speculation support is implemented. Second, and perhaps most importantly, we evaluate these techniques *after* the compiler has eliminated obvious data dependences and scheduled any critical forwarding paths, thereby removing the "easy" bottlenecks to achieving good performance. This leads us to very different conclusions than previous studies on exploiting value prediction within TLS [24, 26]. Third, we demonstrate the importance of throttling back value prediction to avoid the high cost of misprediction, and propose and evaluate techniques for focusing prediction on the dependences that matter most. Fourth, we present the first exploration of how *silent stores* can be exploited within TLS; compared with using traditional value prediction mechanisms to predict speculative memory loads, our silent stores approach yields comparable (if not better) performance while requiring considerably less hardware support. Finally, we evaluate two novel hardware techniques for enhancing the performance of synchronized dependences across speculative threads, but find mixed or disappointing results compared with what our compiler can do to optimize such cases in software.

The remainder of this paper is organized as follows. In Section 2 we describe our approach to TLS support, including our hardware implementation and compiler infrastructure. We take a closer look at the potential for improving value communication in Section 3, and show that our compiler optimizations have a large impact. In Section 4 we investigate techniques for improving value prediction, and explore methods to improve synchronization. Finally, in Section 5 we evaluate the combination of all techniques, and conclude in Section 6.

2 Our Support for Thread-Level Speculation

This section describes the goals of our approach, how we implement support for TLS in hardware, our compiler support, and our experimental framework. While this study is within the context of our approach to TLS support [31, 32], the techniques that we suggest for improving value communication are applicable to other approaches as well.

2.1 Goals of Our Approach

Before we begin our investigation of value communication for TLS, it is important to understand the philosophy behind our approach. First and foremost, our goal is to parallelize general-purpose programs. Our scheme supports parallelization of scientific codes, but for now we focus on the more difficult problem of parallelizing integer applications. Second, we want to keep the hardware support simple and minimal: we avoid large structures that are specialized for speculation, and preserve the performance of non-TLS workloads. Third, we take full advantage of the compiler which selects the regions of code to speculatively parallelize, eliminates data dependences where possible, and otherwise inserts synchronization and schedules the critical paths.

2.2 Underlying Hardware Support

Our scheme [31, 32] is applicable to shared-cache architectures, but for now we focus on single-chip multiprocessors where each processor has its own physically-private first-level data cache, connected to a unified second-level cache by a crossbar switch. TLS hardware support must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. In our scheme we implement this support by using the data caches and an extended version of standard invalidation-based cache coherence.

In a nutshell, our coherence scheme works by tracking which cache lines have been speculatively loaded or modified, and piggybacking a sequence number on coherence messages to detect when an epoch has violated a data dependence. We buffer speculative modifications from regular memory by ensuring that speculatively-modified cache lines are not evicted[2] so that only committed, non-speculative modifications are visible to the rest of the memory hierarchy. We also provide support for *multiple writers*, where two epochs can each speculatively modify their own copy of the same cache line: the coherence mechanism uses the sequence numbers to properly combine the cache lines when they are committed.

2.3 Compiler Support

In contrast with hardware-only approaches to TLS, we rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [33], and performs the following phases when compiling an application to exploit TLS.

Deciding Where to Speculate: For this paper, we focus solely on loops. With the help of automatically-gathered profile information, the compiler selects loops to maximize coverage while meeting heuristics for epoch size and loop trip counts: each loop must comprise at least 0.1% of overall execution time and have an average of at least 1.5 epochs per instance, as well as an average of at least 15 instructions per epoch. Once the key loops are selected, the compiler automatically applies loop unrolling to small loops to help amortize the overheads of speculative parallelization.

Transforming to Exploit TLS: Once speculative regions are chosen, the compiler inserts new TLS-specific instructions into the code that interact with the TLS hardware to create and manage the speculative threads (aka "epochs") [31]. We must also satisfy register dependences between speculative threads; to accomplish this, the compiler "forwards" register values between successive epochs by accessing a

[2] If a speculative cache line must be evicted, we simply cause speculation to fail for the corresponding epoch.

special portion of the stack called the *forwarding frame* which allows hardware to manage synchronization and communication for these values. Before the first use of a forwarded value, the compiler inserts a `wait` instruction, and then reads the value from the forwarding frame. After the last definition, the value is written back to the forwarding frame and a `signal` instruction allows the next epoch to proceed.

Optimization: Without optimization, execution can be unnecessarily serialized by synchronization (through `wait` and `signal` operations). A pathological case is a "`for`" loop in the `C` language where the loop counter is used at the beginning of the loop and then incremented at the end of the loop—if the loop counter is synchronized and forwarded then the loop will be serialized. However, scheduling can be used to move the `wait` and `signal` closer to each other, thereby reducing this critical path. Our compiler schedules these critical paths by first identifying the computation chain leading to each `signal`, and then using a dataflow analysis which extends the algorithm developed by Knoop [19] to schedule that code in the earliest safe location. We can do even better for any loop induction variable that is a linear function of the loop index; the scheduler hoists the associated code to the top of the epoch and computes that value locally from the loop index, avoiding any extra synchronization altogether. These optimizations have a large impact on performance, as we show later in Section 3.1.

Code Generation: Our compiler outputs `C` source code which encodes our new TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements. This source code is then compiled with `gcc` v2.95.2 using the "`-O3`" flag to produce optimized, fully-functional MIPS binaries with TLS instructions.

2.4 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 R10000 [38]. 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 simulate up to the first billion instructions[3] of SPECint95 and SPECint2000 benchmarks [9].[4]

Table 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
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

Table 2. Benchmark statistics.

Application Name	Portion of Dynamic Execution Parallelized (Coverage)	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance
BZIP2	98.1%	1	251.5	451596.0
CRAFTY	36.1%	34	30.8	1315.7
GZIP	70.4%	1	1307.0	2064.8
MCF	61.0%	9	206.2	198.9
PARSER	36.4%	41	271.1	19.4
PERLBMK	10.3%	10	65.1	2.4
VORTEX2K	12.7%	6	1994.3	3.4
VPR	80.1%	6	90.2	6.3
COMPRESS95	75.5%	7	188.2	68.4
GO	31.3%	40	2252.7	56.2
IJPEG	90.6%	23	1499.8	33.8
LI	17.0%	3	176.4	124.9
M88KSIM	56.5%	6	840.4	50.2
PERL	43.9%	4	137.3	2.2

3 A Closer Look at Improving Value Communication

In this section, we evaluate the impact of compiler optimization on performance and then show the potential for further improvement by hardware techniques.

3.1 Impact of Compiler Optimization

We begin by analyzing the performance impact of our compiler on TLS execution. Table 2 shows some statistics on our benchmarks. We observe that our *coverage* (i.e. the portion of dynamic execution time that has been parallelized using TLS) is reasonably good for most benchmarks: 51.4% on average, and as high as 98.1%. Figure 4 shows the performance on a single-chip, 4-processor multiprocessor. For each application in Figure 4, the leftmost bar (*S*) is the original sequential version of the code, and the next bar (*T*) is the TLS version of the code run on a single processor. For each experiment, we show region execution time normalized to the sequential case (*S*); hence bars that are less than 100 are speeding up, and bars that are larger than 100 are slowing down, relative to the sequential version. Comparing the TLS version run sequentially (*T*) with the original sequential version (*S*) isolates the overhead of TLS transformation. In all cases, this is roughly 10%. When we run the TLS code in parallel, it must overcome this overhead in order to achieve an overall speedup.

Each bar in Figure 4 is broken down into six segments explain-

[3] 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.

[4] At the time of publication, our infrastructure could not yet handle GCC, TWOLF, GAP, nor EON.

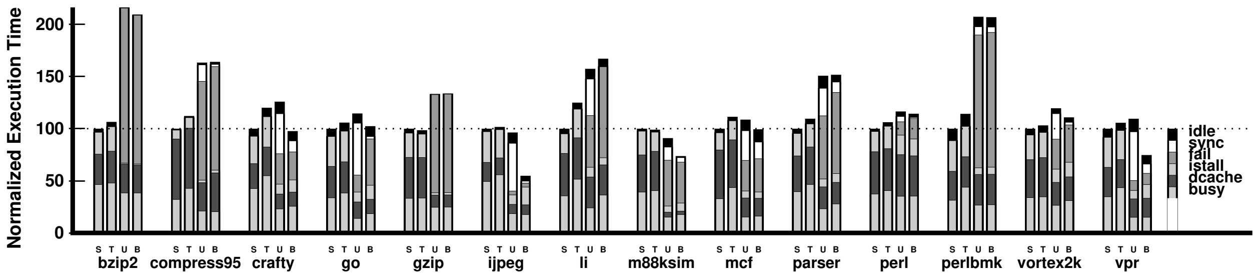


Figure 4. Performance impact of our TLS compiler. For each experiment, we show normalized region execution time scaled to the number of processors multiplied by the number of cycles (smaller is better). S is the sequential version, T is the TLS version run sequentially. There are two versions of TLS code run in parallel: U and B are without and with compiler scheduling of the critical forwarding path, respectively. Each bar shows a detailed breakdown of how time is being spent.

ing what happened during all potential graduation slots.[5] The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining five segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *dcache* segment is the number of non-graduating slots attributed to data cache misses; the *sync* portion represents slots spent waiting for synchronization for a forwarded location; the *istall* segment is all other slots where instructions cannot graduate; the *idle* segment represents slots where the reorder buffer is empty.

We consider two versions of TLS code running in parallel: the *B* case includes all of the compiler optimizations described earlier in Section 2.3, and the *U* case is the same minus the aggressive compiler scheduling to reduce the critical forwarding path. In nearly every case, the "unoptimized"[6] version (*U*) slows down with respect to the sequential version. The additional impediments include decreased data cache locality, synchronization, and failed speculation. Many benchmarks spend a significant amount of time synchronizing on forwarded values (as shown by the *sync* portion). Some benchmarks suffer from non-negligible *idle* segments; in general, this indicates load imbalance and in many cases it is due to regions that have fewer epochs than there are processors. If the compiler optimizes forwarded values by removing dependences due to certain loop induction variables and scheduling the critical path (*B*, our baseline), we observe that the performance of several benchmarks (CRAFTY, GO, IJPEG, M88KSIM, MCF, VORTEX2K, and VPR) improves substantially through decreased synchronization (*sync*), indicating that this is a crucial optimization.

3.2 The Potential for Further Improvement by Hardware

To illustrate that the performance of many of our benchmarks is limited by the efficiency of value communication, we show in Figure 5 the impact of ideal prediction on performance. First, in the *F* experiment we see the impact of perfect prediction of forwarded values. In effect, this means that there will be no time spent waiting for synchronization of forwarded values. Most benchmarks improve slightly, while M88KSIM, MCF, and PARSER show a substantial improvement: this makes sense since the baseline experiment (*B*) shows these benchmarks to be somewhat limited by synchronization. All benchmarks except for IJPEG and VPR suffer from a significant amount of failed speculation in the *B* and *F* experiments.

In the *M* experiment, we measure the impact of perfect memory value prediction, which means that no epoch will suffer from failed speculation. In this case, we see a great improvement in most benchmarks. CRAFTY, MCF, and PARSER show a significant synchronization portion for the *M* experiment, indicating that synchronization is still a limiting factor for these benchmarks.

Finally, in the *P* experiment we evaluate the impact of perfect prediction of both memory and forwarded value values. In this case, MCF and PARSER show a significant benefit compared with perfect memory value prediction alone, while the other benchmarks show only modest improvements. Evidently, avoiding failed speculation is the main bottleneck to good performance, while improving synchronization may still be important for some benchmarks. Also, if we cannot fully eliminate failed speculation then improving synchronization will still be important. Note that BZIP2, LI, PERL, and PERLBMK do not speed up, even with perfect prediction of memory and forwarded values. For these four benchmarks, the decreased data cache locality of executing on four processors is the limiting factor. IJPEG will not speed up further even under perfect prediction of both forwarded and memory values (*P*).

4 Evaluation of Techniques for Improving Value Communication

In this section we first focus on the benefits of predicting values for TLS: we describe the issues related to value prediction in the midst of speculation, describe how to predict memory values and how to predict forwarded values, and then discuss how to apply the technique of optimizing silent stores. Then, we focus on improving synchronization and automatically applying synchronization when speculation and prediction are ineffective.

4.1 Techniques for When Prediction Is Best

Value prediction in the context of a uniprocessor is fairly well understood [13, 22, 29, 37], while value prediction for thread-speculative architectures is relatively new. Gonzalez *et al.* [24]

[5] The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors.

[6] Note that the "unoptimized" case still includes the `gcc` "`-O3`" flag, and is optimized in every way except for the aggressive critical forwarding path scheduling.

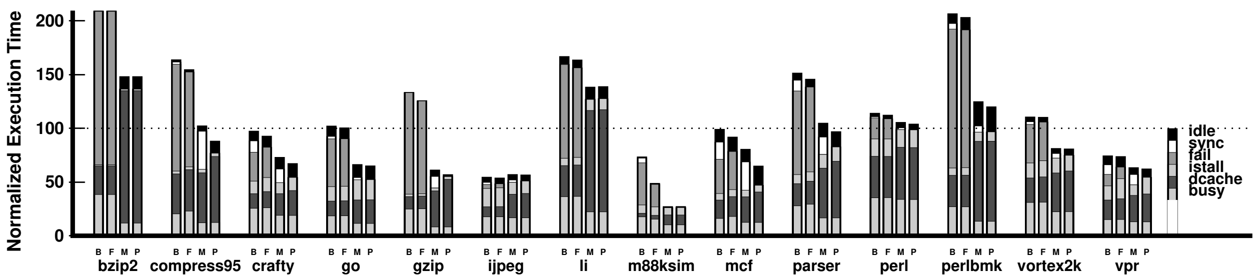


Figure 5. Potential for improved value communication. For each experiment, we show normalized region execution time (smaller is better). B is the baseline TLS version run speculatively in parallel, M shows perfect prediction of memory values, F shows perfect prediction of forwarded values, and P shows perfect prediction of both forwarded and memory values.

evaluated the potential for value prediction when speculating at a thread-level on the innermost loops from SPECint95 [9] benchmarks, and concluded that predicting synchronized (forwarded) register dependences provided the greatest benefit, and that predicting memory values did not offer much additional benefit. The opposite is true of our results for two reasons. First, our compiler has correctly scheduled easily-predictable but frequently-synchronized loop-induction variables so that they cannot cause dependence violations, and has also scheduled the code paths of forwarded values to minimize the impact of synchronization on performance. Second, we have selected much larger regions of code to speculate on, resulting in a greater number of memory dependences between threads. Concurrent with our work, Cintra *et al.* [7] investigated the impact of value prediction after the compiler has optimized loop induction variables in floating-point applications. Several other works evaluate the impact of value prediction without such compiler optimization. Oplinger *et al.* [26] evaluate the potential benefits to TLS of memory, register, and procedure return value prediction, and Akkary *et al.* [2] and Rotenberg *et al.* [28] also describe designs that include value prediction.

Predicting values for TLS has similar issues to predicting values in the midst of branch speculation, but at a larger scale. With branch speculation, we do not want to update the predictor for loads on the mispredicted path. Also, when a value is mispredicted we need only squash a relatively small number of instructions, so the cost of misprediction is not large. Similarly, in TLS we only want to update the predictor for values predicted in successful epochs, but this will require either a larger amount of buffering or the ability to back-up and restore the state of the value predictor. Furthermore, the cost of a misprediction is high for TLS: the entire epoch must be re-executed if a value is mispredicted because a prediction cannot be verified until the end of the epoch when all modifications by previous epochs have been made visible.[7] Finally, for TLS we require that each epoch has a logically-separate value predictor. For SMT or other shared-pipeline speculation scheme, this does not mean that each requires a physically separate value predictor, but that the prediction entries must be kept separate by incorporating the epoch context identifier into the indexing function. This is necessary since multiple epochs may need to simultaneously predict different versions of the same location.

For this paper, we model an aggressive hybrid predictor that combines a $1\text{K}\times 3$-entry context predictor with a 1K-entry stride predictor, using 2-bit, up/down, saturating confidence counters to select between the two predictors. We found that the number of mispredictions can be minimized by simply predicting only when the prediction confidence is at the maximum value. Finding the smallest and simplest predictor that produces good results is beyond the scope of this paper. It is important to note that we also model misprediction by re-executing any epoch that has used a mispredicted-value. A misprediction is not detected until the end of the epoch when the prediction is verified.

4.1.1 Memory Value Prediction

One potential way to eliminate data dependence violations between speculative threads is through the prediction of memory values. But which loads should we predict? A simple approach would be to predict every load for which the predictor is confident. Previous work [4, 11] shows that focusing prediction on critical path instructions is important for uniprocessor value prediction when modeling realistic misprediction penalties. Similarly, the cost of misprediction in TLS is very high, so we instead want to focus only on the loads that can potentially cause misspeculation. Fortunately, this information is available from the speculative cache line state: only loads that are *exposed*[8] can cause speculation to fail. Since our scheme tracks which words have been speculatively-modified in each cache line, we can decide whether a given load is exposed.

Table 3 shows some statistics for the prediction of exposed loads using the predictor described above. M88KSIM and PERL are quite predictable, while the remaining benchmarks also provide a significant fraction of correct predictions. We see that the amount of misprediction is quite small—fewer than 4% of predictions are incorrect for all benchmarks. Hence we expect memory value prediction to work well.

In Figure 6, the *E* experiment shows the impact of predicting all exposed loads for which the predictor is confident. In almost every case, performance is worse due to an increased amount of failed speculation caused by misprediction. The problem is that it only takes a single misprediction to cause speculation to fail.

Rather than predict all exposed loads, we can be more selective by only predicting loads that are likely to cause dependence violations. We can track these loads with the following two devices. First, we keep a 16-entry table (called the *exposed load table*) that is indexed by the cache line tag, and stores the PC of the

[7] Some schemes support selective-squashing of instructions that have used a mispredicted value [2], but this requires a large amount of buffering.

[8] A load that is not proceeded in the same epoch by a store to the same location is considered to be exposed [1].

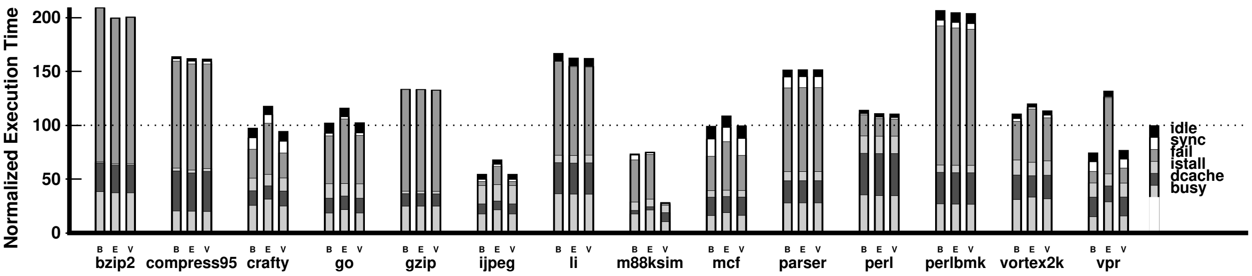


Figure 6. Performance with memory value prediction. B **is the baseline experiment,** E **predicts all** *exposed* **loads, and** V **only predicts loads that have caused violations.**

Table 3. Memory Value Prediction statistics.

Application	Avg. Exposed Loads per Epoch	Incorrect	Correct	Not Confident
BZIP2	9.5	0.2%	63.4%	36.3%
CRAFTY	4.5	3.0%	48.6%	48.3%
GZIP	66.6	1.4%	52.8%	45.7%
MCF	2.5	1.7%	34.9%	63.3%
PARSER	3.6	3.2%	48.7%	48.0%
PERLBMK	1.6	0.9%	17.9%	81.0%
VORTEX2K	25.4	2.8%	64.9%	32.2%
VPR	6.3	3.6%	49.8%	46.4%
COMPRESS95	12.0	0.3%	31.8%	67.9%
GO	7.0	2.5%	41.2%	56.2%
IJPEG	4.4	1.6%	35.4%	62.8%
LI	2.3	2.1%	50.8%	46.9%
M88KSIM	7.5	1.2%	90.9%	7.7%
PERL	12.3	1.1%	79.7%	19.1%

Table 4. Forwarded Value Prediction statistics.

Application	Incorrect	Correct	Not Confident
BZIP2	0.0%	0.0%	0.0%
CRAFTY	5.5%	24.6%	69.7%
GZIP	0.2%	98.0%	1.6%
MCF	2.5%	48.5%	48.9%
PARSER	2.8%	11.6%	85.5%
PERLBMK	2.9%	61.7%	35.2%
VORTEX2K	2.2%	81.9%	15.7%
VPR	2.8%	26.4%	70.7%
COMPRESS95	3.7%	31.2%	65.1%
GO	3.7%	28.3%	67.9%
IJPEG	5.8%	72.4%	21.6%
LI	1.0%	18.7%	80.1%
M88KSIM	5.4%	91.0%	3.4%
PERL	1.5%	91.4%	7.0%

corresponding exposed load. Subsequent exposed loads simply overwrite the appropriate entry—hence we track the most recent exposed loads. Second, whenever a dependence violation occurs it is associated with a cache line, so we can use the cache line tag to index the exposed load table and retrieve the PC of the offending load. Hence we can keep a list of load PCs which have caused violations (the *violating loads list*), and can now use this list to decide which loads we should predict. In Figure 6, the V experiment shows the impact of predicting only loads that have caused violations, as given by the violating loads list. Compared with the baseline B, we see that every benchmark either improves slightly or at least remains unchanged, except for VORTEX2K which degrades slightly, and M88KSIM which improves significantly by eliminating much failed speculation. Hence with proper throttling, memory value prediction can be used to improve the performance of some applications.

Having explored value prediction for the sake of avoiding failed speculation, we now turn our attention to using value prediction to mitigate the performance impact of explicit synchronization.

4.1.2 Prediction of Forwarded Values

Recall that forwarded values are those that are frequently modified and so they are synchronized between epochs by the compiler. Just as we did with memory values, we can also predict forwarded values. However, while we predict memory values to decrease the amount of failed speculation, we predict forwarded values to decrease the amount of time spent synchronizing. Table 4 shows some statistics for the prediction of forwarded values. We see that the fraction of incorrect predictions is somewhat higher than for memory values, and the fraction of correct predictions is not as high.

In Figure 7, the F experiment shows the impact of predicting forwarded values: MCF improves by 4.2%, GZIP by 6.0%, and M88KSIM by 43.6%. The remaining benchmarks are not greatly affected, except for CRAFTY and VPR which become slightly worse due to mispredictions. In an attempt to remedy this problem, we applied a similar technique to that used in memory value prediction: we track which forwarded loads that cause the pipeline to stall waiting for synchronization, and only predict those values. The S bars shows the results of this experiment, which maintains the performance of the F experiment in every benchmark, and solved the problem for CRAFTY but not for VPR. In summary, the prediction of forwarded values is an effective way to reduce the amount of time spent synchronizing for some applications.

4.1.3 Silent Stores

Often, a store does not actually modify the value of its target location. In other words, the value of the location before the store is the same as the value of the location after the store. This occurrence is known as a *silent store* [21], and was first exploited to reduce coherence traffic. A store that is expected to be silent is replaced with a load, and the loaded value is compared with the value to be stored. If they are not the same, then the store is executed after all, otherwise we save the coherence traffic of gain-

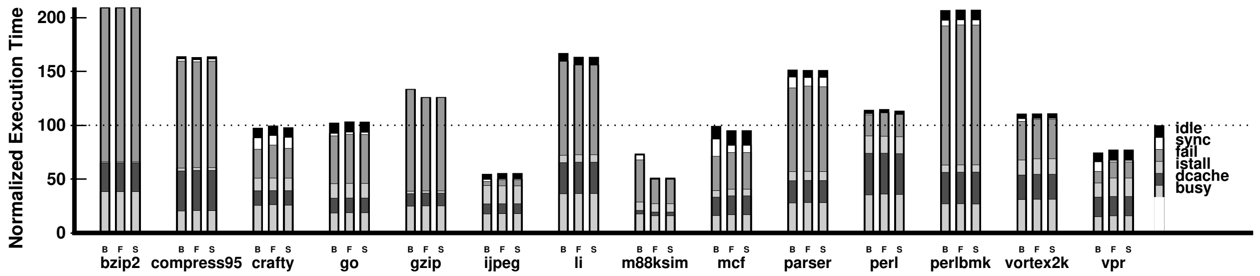


Figure 7. Performance of forwarded value prediction. B **is the baseline experiment,** F **predicts all forwarded values** S **predicts forwarded values that have caused stalls.**

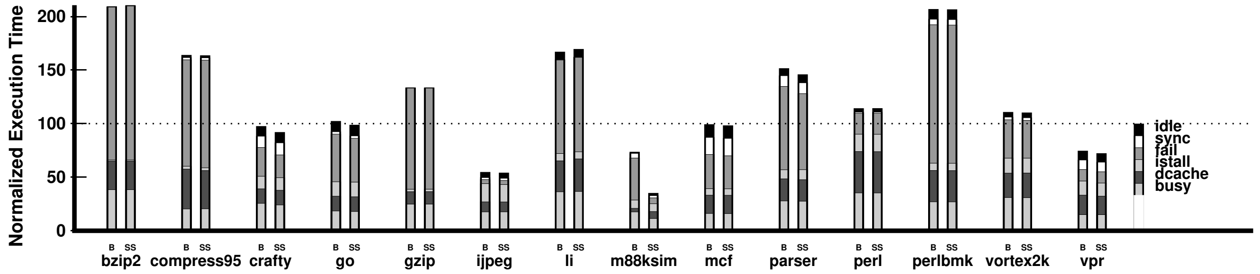


Figure 8. Performance of silent stores optimization. B **is the baseline experiment, and** SS **optimizes silent stores.**

Table 5. Percent of Dynamic, Non-Stack Stores That Are Silent.

Application	Dynamic, Non-Stack Silent Stores
BZIP2	11%
CRAFTY	16%
GZIP	4%
MCF	19%
PARSER	12%
PERLBMK	7%
VORTEX2K	84%
VPR	26%
COMPRESS95	80%
GO	16%
IJPEG	31%
LI	19%
M88KSIM	57%
PERL	36%

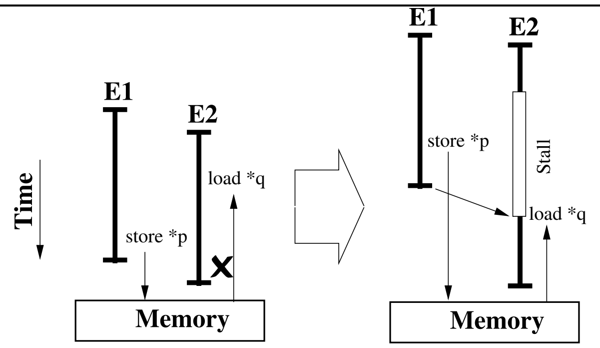


Figure 9. Dynamic synchronization, which avoids failed speculation (left) by stalling the appropriate load until the previous epoch completes (right).

ing exclusive access to the cache line and eliminate future update traffic. This same technique can be applied to TLS to avoid data dependence violations so that a dependent store-load pair can be made independent if the store is silent.

In Table 5 we see that silent stores are abundant, ranging from 4% to 80% of all dynamic non-stack stores within speculative regions. However, what matters is whether the stores which cause dependence violations are silent. Figure 8 shows that optimizing silent stores results in a slight improvement for most benchmarks, and a large improvement in M88KSIM; only LI performs slightly worse when optimizing silent stores. Compared with using value prediction to avoid potential memory dependences (as we explored earlier in Section 4.1.1), this silent stores approach yields similar if not better performance, but requires significantly less hardware support (e.g., no value predictor is needed). Hence this appears to be a very attractive technique for enhancing TLS performance.

4.2 Techniques for When Synchronization Is Best

We now turn focus on the scenarios when synchronization is the right thing to do. We investigate techniques for dynamic synchronization of dependences, and prioritization of the critical path.

4.2.1 Hardware-Inserted Dynamic Synchronization

For many of our benchmarks, failed speculation is a significant performance limitation; and as we observed in Section 4.1, prediction alone cannot eliminate all dependence violations. For dependences with unpredictable values that occur frequently, the only remaining alternative is to synchronize. Our compiler has already inserted synchronization for local variables. Still many dependences remain, as demonstrated in Section 3.2, which can be synchronized dynamically by hardware.

Dynamic synchronization has been applied to both uniprocessor and multiprocessor domains. Chrysos *et al.* [5] present a

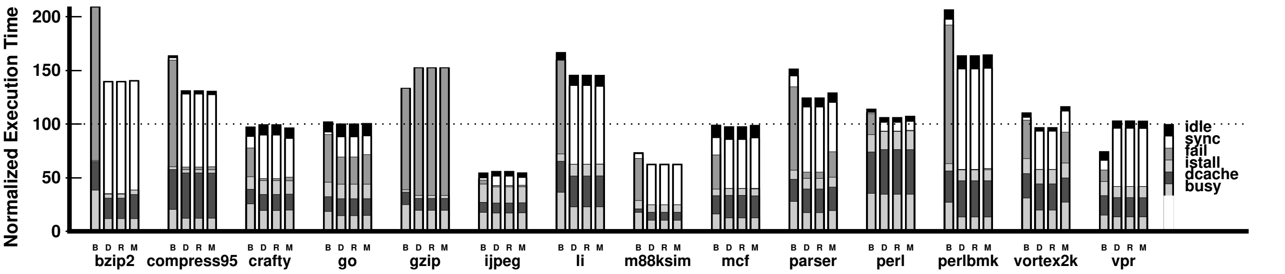


Figure 10. Performance of dynamic synchronization. B is the baseline experiment, D automatically synchronizes all violating loads, R builds on D by periodically resetting the violating loads list, and M builds on R by requiring a load to have caused at least 4 violations since the last reset before synchronizing it.

design for dynamically synchronizing dependent store-load pairs within the context of an out-of-order issue uniprocessor pipeline, and Moshovos *et al.* [25] investigate the dynamic synchronization of dependent store-load pairs in the context of the Multiscalar architecture [12, 30].

Both of these works differ from ours because they have the ability to forward a value directly from the store to the load in a dynamically-synchronized store-load pair: this is trivial in a uniprocessor since the store and load issue from the same pipeline; for a multiprocessor like the Multiscalar, this requires that the memory location in question is implicitly forwarded from the producer to the consumer—functionality that is provided by the Multiscalar's *address-resolution buffer* [12]. Our scheme does not provide this support because it would require the memory coherence protocol to perform complex version management.

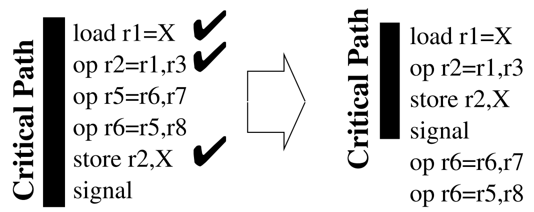
Figure 9 illustrates how we dynamically synchronize. When a load is likely to cause a dependence violation, we can prevent speculation from failing by instead stalling the load until the previous epoch is complete: at that point, all modifications by previous epochs will be visible and the load can safely issue. We can use the *violating loads list* described in Section 4.1.1 to identify the loads most likely to cause a violation that should therefore be synchronized.

In Figure 10, experiment *D* shows the performance of dynamic synchronization where we have synchronized every load in the violating loads list. By inspecting both result graphs, we see that failed speculation has been replaced with synchronization as expected, resulting in improved performance for 10 of the 14 benchmarks. However, CRAFTY, GZIP, and VPR are now over-synchronized: we have unwittingly replaced successful speculation with synchronization as well. In an attempt to mitigate this effect, we periodically reset the violating loads list in experiment *R*, and build on that in experiment *M* by requiring that a load be responsible for at least 4 violations since the last reset before synchronizing. The *M* experiment solves the problem for CRAFTY but not for VPR, and performance is degraded for PARSER and VORTEX2K. Overall, this technique has a greater benefit than cost (an average improvement of 9%), and is a promising technique for improving the performance of TLS.

4.2.2 Prioritizing the Critical Path

In Section 4.1.2 we observed that even after aggressive prediction of forwarded values, synchronization is still an impediment

(a) *Prioritization*



(b) *Statistics*

Application	Issued Insts That Are High Priority and Issued Early	Improvement in Avg. Start-to-Signal Time (cycles)		
		Unprioritized	Prioritized	Speedup
BZIP2	0.0%	0.0	0.0	0.0
CRAFTY	6.8%	118.5	117.7	0.99
GZIP	3.6%	1622.6	1636.2	1.00
MCF	9.9%	86.7	80.2	0.92
PARSER	9.7%	110.9	105.6	0.95
PERLBMK	18.1%	45.5	44.0	0.96
VORTEX2K	3.6%	304.8	290.4	0.95
VPR	4.7%	81.8	79.7	0.97
COMPRESS95	7.1%	136.7	135.9	1.01
GO	12.9%	70.2	70.4	1.00
IJPEG	11.2%	28.1	26.0	0.92
LI	27.5%	37.4	33.1	0.88
M88KSIM	9.1%	212.8	218.4	1.02
PERL	16.6%	42.4	44.5	1.04

Figure 11. Prioritization of the critical path. We show (a) our algorithm, where we mark the instructions on the input chain of the critical store and the pipeline's issue logic gives them high priority; (b) some statistics, namely the fraction of issued instructions that are given high priority by our algorithm and issue early, and also the improvement in number of cycles from the start of the epoch until each signal.

to good speedup for some benchmarks. We call the instructions between the first use and the last definition of a forwarded value the *critical path*. A possibility for improving performance when it is not possible to eliminate synchronization is to instead prioritize instructions to help reduce the size of the critical forwarding path. Our compiler already performs this optimization to the best of its ability, but there may be more that can be done dynamically by hardware at run-time.

Our hardware prioritization algorithm works as shown in Figure 11(a). We mark all instructions with registers on the input-chain of the critical store. We also track the critical path through memory, so that a critical load also depends on the store which

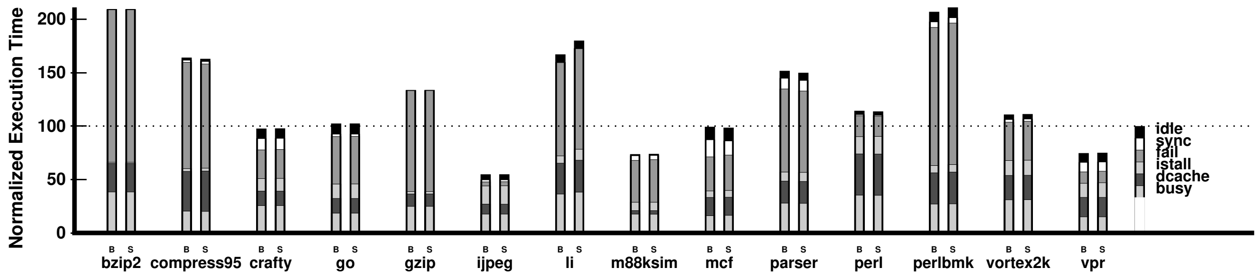


Figure 12. Performance impact of prioritizing the critical path: B is the baseline experiment, and S prioritizes the critical path.

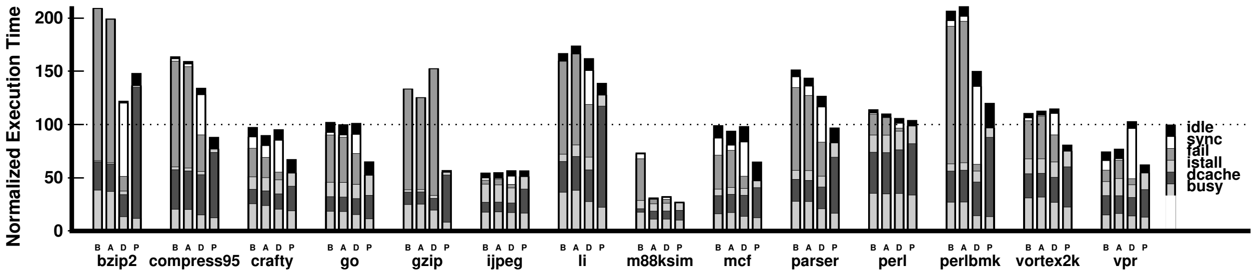


Figure 13. Performance of all techniques combined. B is the baseline experiment, A performs all optimizations except dynamic synchronization, D performs all optimizations, and P is the perfect prediction result from Section 3.2 for comparison.

produced the value for the given memory location. Ideally, we would also mark any instructions on the input-chain of an unpredictable conditional branch as being on the critical path, but this beyond the scope of this paper. The pipeline issue logic then gives priority to marked instructions so that the associated signal may be issued as early as possible. This algorithm could be implemented using techniques described by Fields *et al.* [11], but for now we focus on the potential impact.

The impact of prioritizing the critical path is shown in Figure 12. Note that we model a 128-entry reorder buffer (see Table 1), so the issue logic has significant opportunity to reorder prioritized instructions. Despite this fact, all benchmarks remain relatively unchanged, while MCF and PARSER improve slightly, and the performance of LI and PERLBMK degrades slightly. To clarify whether our prioritization has had any impact, Figure 11(b) shows the fraction of issued instructions that are given high priority by our algorithm and also issue early. This is between 4% and 28% of issued instructions, an average of 10.8% across all benchmarks with the exception of BZIP2 (which does not have forwarded values). Figure 11(b) also shows the change in the average number of cycles from the start of an epoch to the issue of each signal, for which the results are mixed: COMPRESS95, M88KSIM, and PERL have improved somewhat, GZIP and GO are unchanged, and the remaining benchmarks have slowed-down slightly. Given the potential complexity for implementing this technique and the resulting uncompelling performance, we do not advocate the use of this technique.

5 Combining the Techniques

In this Section, we evaluate the impact of all of our techniques combined. Most techniques are orthogonal in their operation with the exception of memory value prediction and dynamic synchronization: we only want to dynamically synchronize on memory values that are unpredictable. This cooperative behavior is implemented by having the dynamic synchronization logic check the prediction confidence for the load in question, and synchronizing only when confidence is low.

Figure 13 shows the performance of all techniques combined, where *A* performs all optimizations except dynamic synchronization, *D* performs all optimizations, and *P* is the perfect prediction result from Section 3.2 for comparison. We achieve very close to the ideal speedup for M88KSIM, and we have improved CRAFTY and MCF significantly. After all of our optimizations, we observe that failed speculation remains a problem for many benchmarks. For BZIP2 the *D* experiment shows how some techniques can be complementary since its performance is better than that of any one technique alone.[9] Since including dynamic synchronization (*D*) degrades performance for more than half of the benchmarks, we do not advocate this technique in its current form.

6 Conclusions

We have shown that improving value communication in TLS can yield large performance benefits, and examined the techniques for taking advantage of this fact. Our analysis provides several important lessons. First, we discovered that prediction cannot be applied liberally when the cost of misprediction is high: predictors must be throttled to target only those dependences that limit performance. We observed that silent stores are prevalent, and squashing them can greatly improve the performance of TLS execution. We found that dynamic synchronization improves performance for many applications but can degrade performance for others—this technique requires further throttling before it can be

[9] The *D* bar out-performs the perfect prediction estimate (*P*) because that estimate does not account for the coherence traffic savings of memory value prediction and silent stores—only for the savings in failed speculation.

applied liberally. We also found that hardware prioritization to reduce the critical forwarding path does not work well, even though a significant number of instructions can be reordered. Finally, we have shown that compiler transformations can impact conclusions about hardware TLS support, and demonstrated that the compiler can be quite effective at improving the performance of TLS.

7 Acknowledgments

This research is supported by grants from Compaq, IBM, and NASA. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship. We thank Joel Emer for his many helpful comments regarding this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *MICRO-31*, December 1998.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, 1996.

[4] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture*, 1999.

[5] G. Chrysos and J. Emer. Memory dependency prediction using store sets. June 1998.

[6] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of ISCA 27*, June 2000.

[7] M. Cintra and J. Torrellas. Learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA02*, 2002.

[8] Broadcom Corporation. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[9] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. Technical report. http://www.spechbench.org.

[10] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, 1986.

[11] Brian A. Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *ISCA 2001*, 2001.

[12] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[13] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report EE Department TR #1080, Technion–Israel Institute of Technology, 1996.

[14] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[15] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*, November 1998.

[16] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

[17] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*, 1993.

[18] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[19] Jens Knoop and Oliver Ruthing. Lazy code motion. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 92.

[20] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[21] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA 27*, June 2000.

[22] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, 1996.

[23] P. Marcuello and A. Gonzlez. Clustered Speculative Multithreaded Processors. In *Proc. of the ACM Int. Conf. on Supercomputing*, June 1999.

[24] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *International Symposium on Microarchitecture*, November 1999.

[25] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. June 1997.

[26] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[27] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computing*, September 1980.

[28] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of Micro 30*, 1997.

[29] Y. Sazeides and J. E. Smith. The Predictability of Data Values. *Proceedings of Micro 13*, pages 248–258, December 1997.

[30] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA 22*, pages 414–425, June 1995.

[31] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[32] J. G. Steffan, C. B. Colohan, A. Zhaia, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of ISCA 27*, June 2000.

[33] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[34] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99*, August 1999.

[35] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[36] 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.

[37] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, 1997.

[38] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.