The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization

J. Gregory Steffan and Todd C. Mowry
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{steffan,tcm}@cs.cmu.edu

Abstract

As we look to the future, and the prospect of a billion transistors on a chip, it seems inevitable that microprocessors will exploit having multiple parallel threads. To achieve the full potential of these "single-chip multiprocessors," however, we must find a way to parallelize non-numeric *applications. Unfortunately, compilers have had little success in parallelizing non-numeric codes due to their complex access patterns. This paper explores the potential for using* thread-level data speculation *(TLDS) to overcome this limitation by allowing the compiler to view parallelization solely as a cost/benefit tradeoff, rather than something which is likely to violate program correctness. Our experimental results demonstrate that with realistic compiler support, TLDS can offer significant program speedups. We also demonstrate that through modest hardware extensions, a generic single-chip multiprocessor could support TLDS by augmenting its cache coherence scheme to detect dependence violations, and by using the primary data caches to buffer speculative state.*

1. Introduction

As advances in integrated circuit technology continue to provide more and more transistors on a chip, processor architects are faced with the pleasant challenge of finding the best way to translate these additional resources into improved performance. A number of options have been proposed, including integrating main memory onto the processor chip [17], supporting wider instruction issue, and executing multiple threads of control in parallel [2, 14, 24]. While these options may be mutually exclusive in the short term due to transistor constraints, in the long term we will eventually have enough resources to potentially combine several of these options.

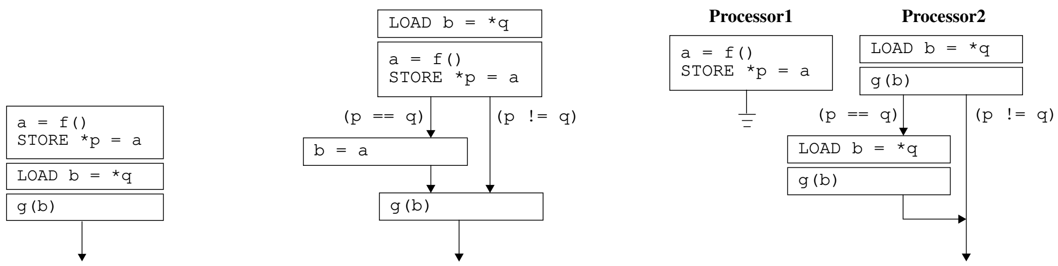
[0] To appear in *HPCA-4*, February 1-4, 1998.

Due to the diminishing returns of wider instruction issue and more memory on the chip, we believe that it is simply a matter of time until processors also exploit the performance benefits of having multiple parallel threads. The coarser-grained *thread-level parallelism* exploited by these "single-chip multiprocessors" is largely orthogonal to the more fine-grained *instruction-level parallelism* which wide-issue machines attempt to exploit within a single thread. Hence, with enough transistors, it may be possible to aggressively exploit both forms of parallelism in a complementary fashion.

What are the performance benefits of single-chip multiprocessing? For a multiprogrammed workload, computational throughput will clearly improve as the independent tasks execute in parallel on multiple processors rather than time-sharing a single processor. However, to reduce the execution time of a *single* application, that application must contain parallel threads. Unfortunately, despite the fact that conventional multiprocessors have been commercially available for quite some time, only a small fraction of the world's software has been written to exploit parallelism. Hence if single-chip multiprocessing is going to succeed and become ubiquitous, the key question is how do we convert the applications we care about into parallel programs?

Expecting programmers to only write parallel programs from now on is unrealistic. Instead, the preferred solution would be for the compiler to parallelize programs automatically. Unfortunately, compilers have only been successful so far at parallelizing the *numeric* applications commonly run on supercomputers [1, 7, 16]. For single-chip multiprocessing to have an impact on most users, we must also find a way to automatically parallelize *non-numeric* applications.

One of the primary challenges in automatic parallelization is determining whether data dependences exist between two potential threads that would prevent them from running safely in parallel. To address this problem in numeric codes, a considerable amount of research has focused on analyzing array accesses within `DO` loops [7]. Although progress has been made in this area, the problem is considerably more



(a) Original Execution **(b)** Instruction-Level Data Speculation **(c)** Thread-Level Data Speculation

Figure 1. Examples of data speculation.

difficult for non-numeric codes due to their complex access patterns, including pointers to heap-allocated objects and complex control flow. Given the size and complexity of real non-numeric programs, automatic parallelization appears to be an unrealistic goal if the compiler must statically prove that threads are independent. Instead, we would prefer to relax the constraints on the compiler such that if it simply believes that two threads are *likely* to be independent, it can optimistically parallelize them without worrying about violating program correctness. With such a model, the compiler can view parallelization solely as a cost/benefit tradeoff, rather than something which may break the program. In this paper, we consider a technique which provides this flexibility: *thread-level data speculation*.

1.1. Thread-Level Data Speculation

To maximize parallelism, we want to perform loads as early as possible so that operations which depend on them can be executed concurrently. Hence it is often desirable to move a load ahead of an earlier store, which is safe provided that they access different memory locations. Since analyzing memory addresses in non-numeric applications is difficult, a potentially attractive option is for the compiler to *speculatively* move a load ahead of a store, and resolve whether this was safe at run-time. If the speculative load turns out to have been unsafe, then a recovery action is taken to restore the correct program state. This technique is known as *data speculation*, and it works well when the unsafe cases are sufficiently rare that the overhead of recovery is small relative to the benefit of increased parallelism.

Figure 1(a) shows an example code fragment, and Figure 1(b) illustrates how it might be optimized to exploit instruction-level data speculation. In this example, the compiler is uncertain whether the pointers `p` and `q` point to the same address, but nevertheless it has speculatively moved the load ahead of the store. At run-time, we can verify the safety of this speculation through either a simple software check or with special hardware support [5].

Thread-**L**evel **D**ata **S**peculation (**TLDS**) is analogous to instruction-level data speculation, except that the load and store are executed by separate threads of control which run in parallel, as illustrated in Figure 1(c). A given speculative load is safe provided that its memory location is not subsequently modified by another thread such that the store should have preceded the load in the original sequential program. When such dependence violations are detected, a recovery action is taken such as partially re-executing the thread which performed the failed speculative load.

While instruction-level data speculation has received much attention [5, 9, 20], the only relevant work on thread-level data speculation for non-numeric codes when we performed our study was the Wisconsin Multiscalar architecture [3, 4, 21]. This tightly-coupled ring architecture assigns threads around the ring in program order, provides a hardware mechanism for forwarding register values between processors, and uses a centralized structure called the "address resolution buffer" (ARB) [4, 21] to detect data dependences through memory. When an unsafe speculation is detected, a purely hardware-based mechanism squashes computation in reverse order around the ring until it can be safely restarted. While the Multiscalar architecture supports a form of thread-level data speculation with relatively little software support, we believe that this hardware-centric approach has some important disadvantages. First, the ARB is a relatively large, centralized structure which must be checked on all loads and stores. A centralized approach has the danger of increasing load latency due to long wire delays—instead, we would prefer a more distributed approach where loads and stores can be satisfied directly from their own primary caches. Second, the ring architecture limits our ability to optimize data cache locality and to perform efficient multiprogramming. Instead, we would prefer a more flexible topology where computation can be placed wherever we wish. (Concurrent with our work, researchers in the Stanford Hydra and Wisconsin Multiscalar projects have also explored distributed approaches to TLDS [8, 15].)

1.2. Objectives and Overview

Our goals in this paper are threefold: to quantify the potential performance advantages of TLDS, to propose cost-effective hardware support for TLDS, and to gain insight into the compiler support necessary to effectively exploit

TLDS. The remainder of the paper is organized as follows. Section 2 illustrates how TLDS works, and Section 3 presents our experimental results. Sections 4 and 5 discuss architectural and compiler support for TLDS, respectively. Finally, we conclude in Section 6.

2. An Example: `Compress`

Before we illustrate how TLDS applies to a real application, we briefly introduce some terminology. We say that TLDS parallelism is extracted from a *speculative region*, which consists of a collection of dynamic instruction sequences called *epochs*. For example, with loop-level parallelism, we would say that the loop is a speculative region, and the individual loop iterations would be epochs. Since TLDS parallelism also applies to structures other than loops (e.g., recursion), we have adopted this more general terminology. For TLDS to be effective, each epoch must contain enough work to amortize the costs of thread management and data communication, and the speculative regions must constitute a significant fraction of overall execution time.

The `compress` application in the SPEC92 and SPEC95 benchmark suites is a good candidate for exploiting TLDS. Over 99% of execution time is spent in a single `while` loop which reads each input character and performs the compression. The control flow within the loop body is quite complicated, and on average takes roughly 90 dynamic instructions. While this loop may appear to be an abundant source of data parallelism, a compiler cannot statically prove that loop iterations are independent because they are not. The input characters are used to index a hash table which is modified; hence when two character sequences hash to the same entry, there is a true *read-after-write* (RAW) data dependence. Figure 2(a) shows a pseudo-code representation of this code. Fortunately, due to the nature of a hash table, consecutive characters rarely access the same hash table entry—therefore there is an opportunity to extract parallelism during the iterations *between* actual dependences. Since a single-chip multiprocessor has a relatively small number of processors, we do not need a large gap between data-dependent iterations to keep the machine busy.

Figure 2(b) illustrates how `compress` can be parallelized using TLDS, where each epoch (i.e. loop iteration) is executed as a separate thread. Since the threads are speculative, any stores which they perform must be buffered until it is certain that their results can be safely committed to memory. In this example, the first three epochs successfully complete without any problems. The fourth epoch, however, reads the value `hash[10]` *before* it is modified by epoch 1—since this violates the original program semantics, we must recover by re-executing epoch 4. As we can see in Figure 2(b), the number of epochs between RAW data dependences dictates the amount of parallel speedup that can be achieved—we will quantify this number for

(a) Pseudo-code representation of `compress`

```
while ((c = getchar()) != EOF) {
    /* perform data compression */
    in_count++;
    ...
    ...  = hash[hash_function(c)];
    ...
    hash[hash_function(...)]  = ...;
    ...
    if (...)  {out_count++; putchar(); ...}
    if (free_entries < ...)  {free_entries = ...  }
    ...
}
```

(b) TLDS execution of `compress`

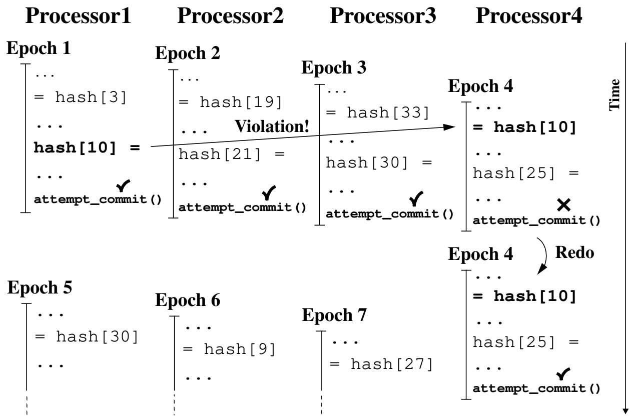


Figure 2. Example of TLDS execution.

`compress` and other applications later in Section 3.

In addition to the hash table accesses, `compress` contains several other sources of RAW data dependences, but fortunately we can either eliminate or at least mitigate their impact with the proper compiler support. For example, the `in_count` variable is incremented on each iteration to count total input characters—the compiler can recognize this as an induction variable, and eliminate it since it is implicit in the epoch number. The `out_count` variable is conditionally incremented inside the loop to count total output characters; since it is not used otherwise within the loop, the compiler could recognize this as a reduction operation and optimize it accordingly (i.e. each processor's partial sum is added together at the end of the loop). Loop-carried dependences also exist inside `getchar()` and `putchar()`—these could also be eliminated through parallel implementations of the I/O routines. Finally, the scalar variable `free_entries` is always read and sometimes modified. Although this dependence cannot be eliminated, the compiler could explicitly forward this value directly to consuming threads whenever it is modified, thereby allowing us to at least *partially* overlap execution of the epochs. Thus we see that a range of different compiler optimizations can potentially enhance TLDS, and we will quantify their benefit in the next section.

Table 1. Benchmarks

Suite	Name	Input Data Set *(size:name)*	Region	Average Dynamic Instrs per Epoch	% of Total Dynamic Instrs
SPEC92	compress	ref:in	r1	89	99.9
	gcc	ref:stmt.i	r1	1092	8.1
			r2	1593	4.0
	espresso	ref:bca.in	r1	32	19.4
	li	ref:li-input.lisp	r1	19	21.9
			r2	286	51.2
	sc	ref:loada1	r1	36	69.3
SPEC95	m88ksim	test:ctl.raw	r1	1232	99.3
	ijpeg	train:vigo.ppm	r1	9406	15.3
	perl	test:primes.pl	r1	67	35.8
	go	train:2stone9.in	r1	80	6.8
NAS Parallel	buk	N = 65536	r1	26	16.5
			r2	18	11.4

3. Performance Impact of TLDS

This section presents our experimental results which quantify the potential performance gain offered by TLDS. There are two separate effects which limit performance: (i) data dependences through memory caused by ambiguous load and store addresses, and (ii) synchronization and communication latency when we explicitly forward scalar values between epochs to partially overlap execution. We start by considering each of these effects in isolation in Sections 3.2 and 3.3, and then later combine both effects to produce overall speedup numbers in Section 3.4.

3.1. Finding and Simulating Speculative Regions

Table 1 summarizes the ten non-numeric applications we study, which are taken from the SPEC92, SPEC95, and NAS Parallel benchmark suites. We compiled these applications with `-O2` optimization using the standard MIPS compilers under IRIX 5.3, and did not modify the source code or object files in any way throughout this study. Our simulator reads traces generated by the MIPS `pixie` utility [19], and models a perfectly-pipelined single-issue processor where each instruction completes in a single cycle.

To locate *speculative regions*—i.e. sections of code that we wish to parallelize using TLDS—we first used the IRIX `prof` utility to identify regions that account for a large portion of total execution time. We then inspected these regions by hand to determine whether they were good candidates for exploiting TLDS. If so, we explicitly identified these regions to our simulator through their instruction addresses. The simulator then measures the exact data dependences between epochs in each speculative region. (For further details on our experimental methodology, see [23].) Since identifying speculative regions by hand was a time-consuming process, we were not able to explore all possible regions, particularly in large programs such as `gcc`. We believe that an automated tool would have located a larger set of speculative regions, and therefore our results may underestimate the potential program speedups.

3.2. Relaxing Memory Data Dependences

Data dependences between epochs can occur either through registers or through memory. Register dependences are generally easier for the compiler to analyze since the storage locations are not ambiguous (in contrast with memory references). Therefore we begin in this section by focusing only on the more difficult case of memory dependences.

As illustrated earlier in Figure 2, TLDS can exploit parallelism whenever gaps exist between data-dependent epochs. We quantify this potential by computing the *run lengths*, which are the number of consecutive epochs within a speculative region delimited by *performance-limiting* read-after-write (RAW) data dependences. (Note that although write-after-read (WAR) and write-after-write (WAW) data dependences might also exist between epochs, our proposed hardware eliminates them through a form of renaming, and hence they do not limit performance.)

While RAW dependences can potentially disrupt parallelism (forcing a processor to re-execute an epoch), this is not always the case. Given that a single-chip multiprocessor will only support T outstanding speculative threads, we know that when epoch E_i is executing, any epoch E_j where $j \leq (i - T)$ must have committed its state already. Hence a RAW dependence of distance d, where $d \geq T$, will not limit our ability to exploit parallelism. For example, assuming that $T = 4$ in Figure 3(a), the RAW data dependence shown between epoch E_8 and epoch E_3 will not limit performance. Therefore there are only two performance-limiting data dependences within these nine epochs, thus resulting in run lengths of two, three, and four. The average run length size of three corresponds roughly to the maximum speedup we might expect on four processors, as illustrated in Figure 3(b).

Figure 3(c) shows the average run lengths for each region given a threshold (T) of ten outstanding epochs.[1] Starting with the base case (**B**)—i.e. the original code—we see that seven of the thirteen regions have natural run lengths of two or more under TLDS. To further exploit TLDS, we can apply compiler optimizations to eliminate RAW data dependences whenever possible. Case "**O**" in Figure 3(c) shows how much the run lengths increase if the compiler eliminates dependences due to induction variables and reduction operations, and uses parallel library routines (as described earlier in Section 2). These optimizations yield significant improvements in the `compress` and `buk` regions. Finally, case "**F**" in Figure 3(c) shows that the scalar dependences

[1] Note that the run length can exceed the threshold. Hence run length does not translate directly into speedup, which is something we take into account later in Section 3.4 when we compute region and program speedups.

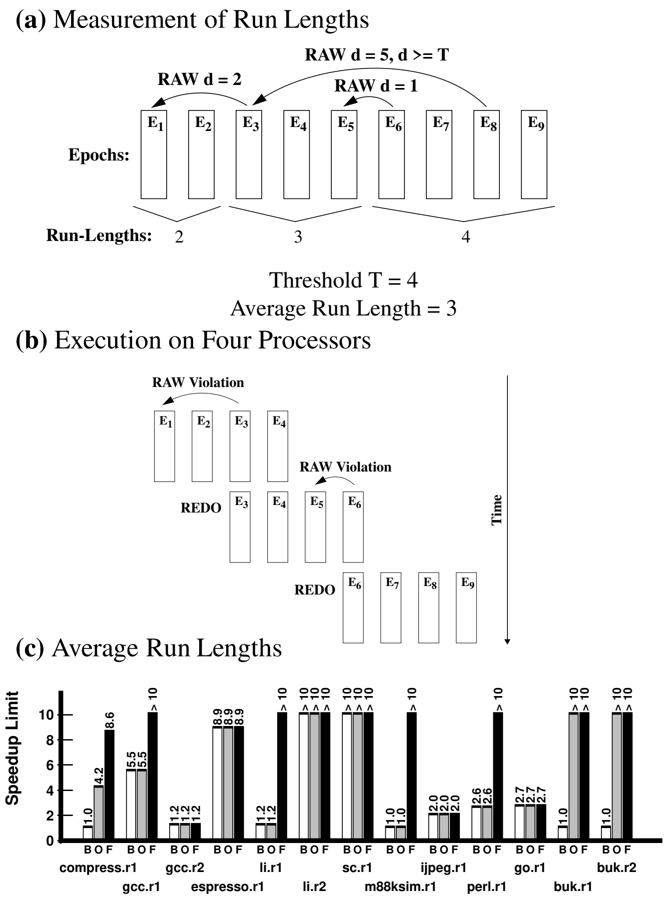


Figure 3. Impact of TLDS on run lengths caused by memory data dependences (B = base case, O = compiler optimizations applied to eliminate dependences, and F = dependences due to forwardable scalars also removed).

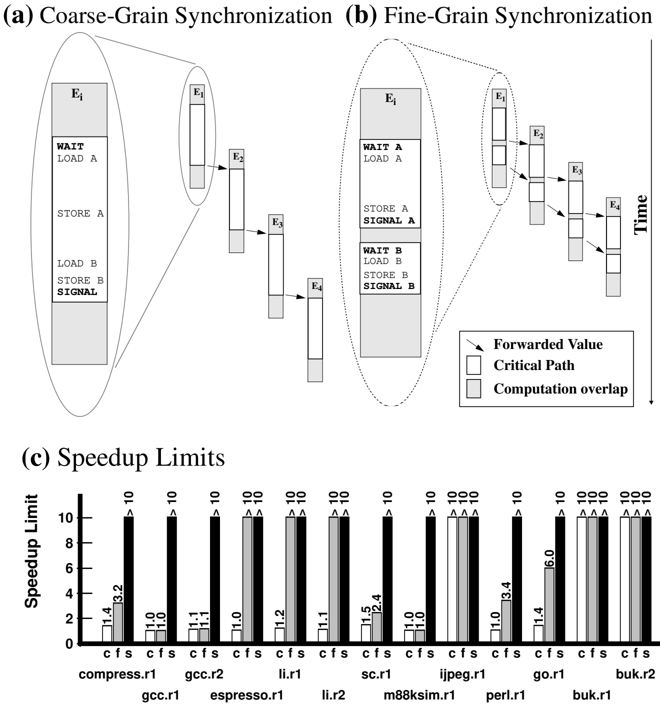


Figure 4. Forwarding scalar values between epochs. (c = coarse-grain synchronization, f = fine-grain synchronization, s = fine-grain synchronization with aggressive instruction scheduling.)

between epochs represent the final barrier to achieving average run lengths of eight or more in `compress`, `gcc.r1`, `li.r1`, `m88ksim`, and `perl`. Although these scalar dependences cannot be fully eliminated (unlike the **B** and **O** cases), we can potentially accelerate these cases by explicitly *forwarding* the values between epochs, as we discuss in the next section.

3.3. Forwarding Data Between Epochs

In cases where RAW dependences frequently occur between consecutive epochs and these dependences cannot be eliminated, we can still potentially accelerate performance by explicitly forwarding data and therefore *partially* overlapping the epochs. In addition to forwarding scalar memory dependences, we must also forward any register dependences. In contrast with the Multiscalar architecture [3, 4, 21], we do not assume any register forwarding hardware—instead, we forward register values explicitly through memory. Although the run lengths in Figure 3(c) ignored register dependences between epochs, we take them into account throughout the remainder of this study as follows. We eliminate two classes of RAW register dependences which the compiler can easily optimize away: those due to induction variables (e.g., updating a loop index), and those which can be trivially eliminated by rescheduling the loop test condition. Otherwise, all remaining cross-epoch register dependences are explicitly forwarded through memory.

In addition to a data transfer mechanism, forwarding data also requires a form of producer-consumer synchronization (e.g., `wait`/`signal` or full/empty bits [2, 18]) so that the receiving processor knows when the value is ready. If multiple values are to be forwarded, the synchronization can occur at either a *coarse granularity* (once per epoch) or a *fine granularity* (once per value), as illustrated in Figures 4(a) and 4(b).

The performance of a speculative region that requires forwarding is limited by the *critical path length*, which is the sum of the non-overlapped portions of each epoch plus the latency of forwarding these values between epochs. With coarse-grain synchronization, the critical path length is straightforward to compute, as illustrated in Figure 4(a). With fine-grain synchronization, there can be *multiple* forwarding paths through the epochs, as shown in Figure 4(b). In this latter case, the overall critical path is simply the longest of these forwarding paths (e.g., the forwarding path for `A` in Figure 4(b)). Roughly speaking, the maximum potential speedup for a speculative region in the presence of forwarding can be computed by dividing the total sequential execution time of all epochs by the critical path length.

The performance with forwarding depends on how aggressively we attempt to minimize the non-overlapped portions of each epoch. In addition to using fine-grain rather than coarse-grain synchronization, we can potentially improve performance further by rescheduling the code to move as many instructions out of the non-overlapped portion of an epoch as possible. To evaluate the potential benefit of improved instruction scheduling, we simulated aggressive instruction scheduling by tracking the dynamic dependence chain depth between instruction pairs that consume and produce forwarded data values. In other words, we measured the minimum possible sizes of the non-overlapped portions within epochs. We do not claim that the compiler will be able to schedule the code this aggressively, but we want to show that proper scheduling can reduce the length of the critical path.

Figure 4(c) shows the speedup limits due to forwarding. In these initial experiments, we assume that forwarded data can be consumed immediately (e.g., through a shared register file); later, in Section 3.4, we consider a more realistic forwarding latency. Three regions (`ijpeg.r1`, `buk.r1`, and `buk.r2`) do not require any forwarding, and hence are not limited by it. Focusing on the other ten regions, we see that coarse-grain synchronization (**c**)—i.e. forwarding data once per epoch—yields speedups above 35% in only three cases (`compress.r1`, `sc.r1`, and `go.r1`), and no speedups above 50%. By using fine-grain synchronization to forward values as soon as they are produced (**f**), the speedup limit increases to over twofold for seven of the ten regions that require forwarding—in several of these cases, the improvement is dramatic. Finally, by combining fine-grain synchronization with aggressive instruction scheduling (**s**) to minimize non-overlapped sections within epochs, we can potentially achieve large speedups in all regions. The benefits of rescheduling are particularly pronounced in `gcc.r1` and `m88ksim.r1`, where speedups in the original code are limited to under 3%, but by rescheduling these relatively large epochs (over 1000 instructions each), we can potentially achieve speedups of tenfold or more.

3.4. Potential Speedups

Having gained insight into how TLDS can relax memory and register data dependences and exploit forwarding, we now translate the run length and critical path metrics into an estimate of actual speedups on a single-chip multiprocessor with four processors. To estimate speedups, we combine the limitations imposed both by memory data dependences and by forwarding data—both of these effects were shown in isolation in Figures 3 and 4.

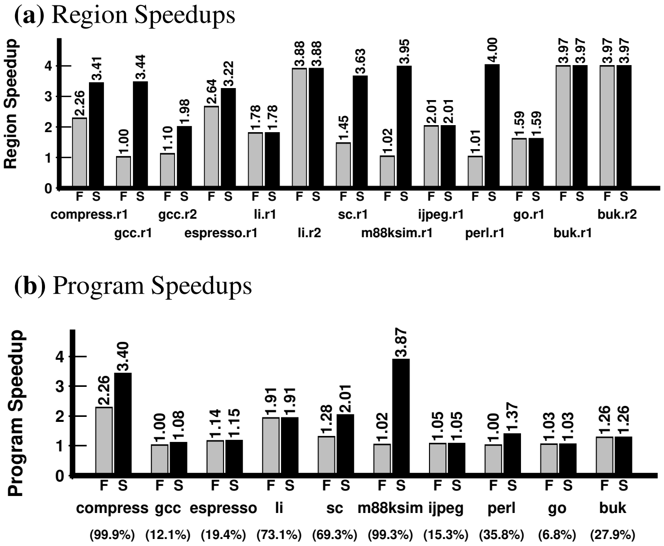


Figure 5. Potential region and program speedups on four processors assuming a 10-cycle forwarding latency, 32B cache lines, and accounting for recovery time (F = forwarding with fine-grain synchronization, S = forwarding with fine-grain synchronization and aggressive instruction scheduling). The coverage is shown below each benchmark.

For these results, we take several new factors into consideration. First, in contrast with Sections 3.3 and 3.2 where we assumed an ideal communication latency of a single cycle, we now assume a *ten* cycle communication latency between processors, which corresponds roughly to communicating through a shared L2 cache. Second, rather than tracking data dependences at a word granularity (as in Sections 3.3 and 3.2), we now track dependences at a *cache line* granularity of 32 bytes. We find that the effects of false dependences due to this increased granularity are not detrimental to the performance of TLDS [23]. Third, we account for the time required to recover from unsuccessful speculation. In contrast with the Multiscalar architecture [21], our model of TLDS involves software in the recovery process; hence we include an estimate of the time required to restore the initial state of the epoch and restart parallel execution, assuming that a violating epoch will wait to become the oldest epoch before attempting to restart. Finally, we ensure that the parallelism does not exceed the number of physical processors (four, in this case) during execution.

Figure 5 shows the potential region and program speedups for the following two cases. Case "**F**" includes the compiler optimizations mentioned in Section 2 to eliminate data dependences, and also performs fine-grained synchronization to forward scalar values. Case "**S**" includes these same optimizations, but also reschedules instructions to maximize parallel overlap. As we see in Figure 5(a), eight of the thirteen regions enjoy significant speedups (50% or more) using TLDS when memory dependences are eliminated and scalars are forwarded under case "**F**". By rescheduling the code, all but one region potentially achieves an overall speedup of roughly twofold or more on four processors, indicating that code scheduling will result in large performance gains for TLDS.

Given the fraction of total execution time spent in each region (shown in Table 1), we can estimate the potential overall speedup for each application, as shown in Figure 5(b). To a large extent, the overall speedup depends directly on our ability to find regions that constitute a large fraction of overall program execution time. The coverage is shown in parentheses below each benchmark. In four applications (`compress`, `li`, `sc`, and `m88ksim`), we found regions covering roughly 70% or more of execution time, and all of these cases can potentially enjoy speedups of nearly twofold or more on four processors. Three other applications (`espresso`, `perl`, and `buk`) achieve more modest speedups of 15-37%, and the remaining three applications improve by less than 10%.

Overall, we find the speedups to be quite good, considering the difficulty of improving the performance of these benchmarks. As we mentioned earlier, we believe that our region coverages (and hence program speedups) are pessimistic in many cases since finding regions by hand was a very time-consuming process, and we could not do justice to large applications such as `gcc`. With an automated tool, we may see even better results.

4. Architectural Support for TLDS

Having demonstrated the potential performance benefits of TLDS, we now discuss how TLDS might be implemented. Our goals are twofold. First, we would like to support an aggressive form of TLDS while requiring only minimal hardware modifications to a generic single-chip multiprocessor. Second, we do not want to sacrifice performance in single-threaded applications or applications that do not exploit TLDS—hence we will avoid complex, centralized structures which can increase primary data cache access latencies. Therefore the starting point for our design is a single-chip multiprocessor where the L2 cache is physically shared and the individual L1 caches are kept coherent to provide a shared memory abstraction. The motivation for having separate L1 caches is that they provide high bandwidth and low latency relative to a single shared L1 cache. The motivation for keeping the L1 caches coherent is that without a shared-memory abstraction, the job of the compiler becomes too difficult—i.e. just as ambiguous memory addresses prevent the compiler from statically proving the independence of threads, they also prevent the compiler from successfully decomposing the data into separate address spaces.

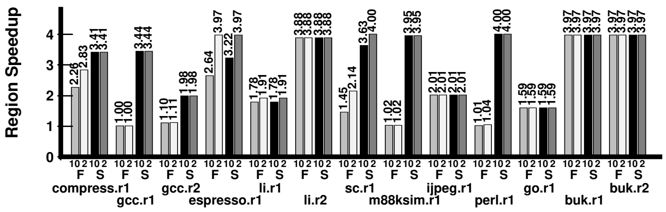


Figure 6. Impact of communication latency (F = forwarding with fine-grain synchronization, S = forwarding with fine-grain synchronization and aggressive instruction scheduling, "10" = 10 cycles, "2" = 2 cycles).

We begin by investigating the importance of communication latency between processors for forwarding values between epochs. We then discuss the issues involved in managing threads, including software's interface to the TLDS hardware support. Next, we illustrate how cache coherence protocols can be extended to detect data dependence violations. Finally, we demonstrate that the cache itself can be used to buffer speculative side effects until they can be safely committed to memory.

4.1. The Importance of Communication Latency

One implication of having separate L1 caches and a shared L2 cache is that by default, the fastest path for communicating between processors is through the shared L2 cache, and this will not occur instantaneously. Hence an important question is whether communicating through a shared L2 cache is fast enough, or whether faster communication mechanisms (e.g., direct register-to-register forwarding [21]) are strictly necessary.

To address this question, we measured the impact of communication latency on our region speedups. Figure 6 includes the same cases shown earlier in Figure 5(a), where the communication latency was assumed to be *ten* cycles (corresponding roughly to communicating through a shared L2 cache). In addition, Figure 6 also shows the impact of a faster *two* cycle communication latency. As we see in Figure 6, the vast majority of the regions show little or no improvement from the faster communication latency. In the three cases where there is a noticeable difference (`compress.r1`, `espresso.r1`, and `sc.r1`), we observe that aggressive instruction rescheduling (i.e. case "**F**") often reduces this sensitivity by making the epochs more latency-tolerant. In summary, although fast communication is helpful in some cases, communicating through the shared L2 cache is a viable approach.

4.2. Thread Management

A number of mechanisms are required by TLDS to manage and coordinate the parallel threads. In many cases, there is considerable flexibility in how these mechanisms might be implemented. Due to space constraints, our goal here is simply to raise the important issues rather than presenting a complete design (additional detail can be found in earlier publications [22, 23]). First, we need a way to create parallel threads and schedule the epochs onto them. One option is to dynamically create a new thread per epoch (perhaps using a lightweight `fork` instruction [13]), and another is to statically create one thread per processor and have them execute multiple epochs.

Second, since dependence violations are detected by comparing epoch numbers, a mechanism is needed such that each thread's epoch number will be visible to the hardware. One way to accomplish this is for software to explicitly pass epoch numbers to the hardware through a new instruction. However, there are three important things to note. First, hardware's representation of epoch numbers does not necessarily need to coincide with epoch numbers in software. Second, epoch numbers represent a *partial* ordering rather than a total ordering, since epochs across unrelated threads (e.g., separate applications) are unordered; hence a portion of an epoch number might be a *thread ID*, which must match exactly for two epochs to be considered ordered with respect to each ther. Finally, in some cases the hardware may be able to implicitly maintain epoch numbers, and hence software would not need to be aware of them.

Third, we need to distinguish speculative versus non-speculative memory accesses, since only speculative operations must be buffered or checked for dependence violations. Rather than creating new flavors of all memory references in the instruction set, we can instead use explicit instructions to dynamically indicate whether a *thread* is speculative or not—when a thread is speculative, all of its memory references will be interpreted as being speculative. A thread should become speculative prior to its first speculative load, and can become non-speculative again once it confirms that its speculation was safe. (Note that the hardware distinguishes the "oldest" thread, and always interprets it as being non-speculative.)

Finally, we need a mechanism for recovering from failed speculation. In contrast with the Multiscalar approach of performing rollback entirely in hardware [21], we propose that *software* performs the bulk of the recovery process, and that hardware simply provides two key pieces of functionality: (i) detecting data dependence violations and notifying software when they occur, and (ii) buffering speculative stores so that software does not have to explicitly roll back their side effects on memory. We discuss these hardware mechanisms in greater detail in the next two subsections.

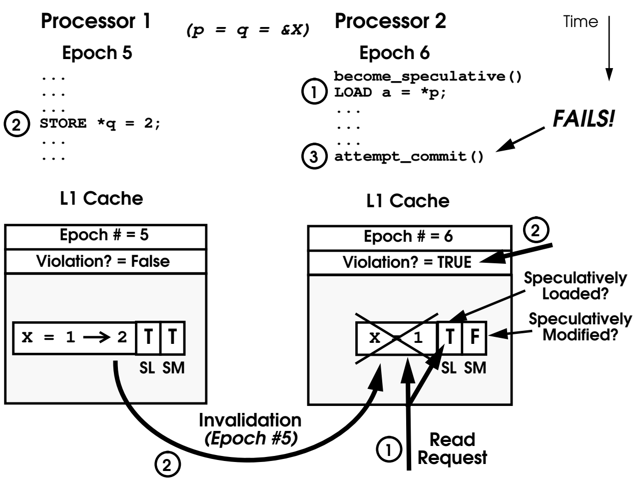


Figure 7. Illustration of an augmented cache coherence scheme which supports TLDS.

4.3. Extending Cache Coherence to Detect Data Dependence Violations

At the heart of TLDS is a mechanism for detecting RAW data dependence violations and recovering the correct program state whenever they occur. Given the potentially large number of addresses that must be compared against each other to determine safety, and given the fact that the exact interleaving of accesses between threads is unknown *a priori* since they run asynchronously, a purely software-based approach of explicitly comparing memory addresses [12] would appear to be impractical. Instead, we propose extending a basic invalidation-based writeback cache coherence protocol to allow hardware to detect potential dependence violations with little overhead.

The basic intuition behind our scheme is as follows. Consider two epochs, E_i and E_{i+1}, where E_i precedes E_{i+1} in the original sequential program. Assume that these epochs execute in parallel, and imagine that E_{i+1} violates a RAW data dependence by speculatively loading a location A before it is modified by E_i. Under normal cache coherence, E_i must first invalidate A from E_{i+1}'s cache to obtain an exclusive copy before its store to A can proceed. If we extend this protocol by piggybacking E_i's epoch number along with the invalidation request, we can compare epoch numbers to determine that the speculative load of A by E_{i+1} was in fact a RAW dependence violation. At this point, we can either set a flag in hardware which E_{i+1} will subsequently check, or else interrupt E_{i+1} to notify it that its speculation has failed.

Figure 7 illustrates how the coherence scheme will detect a data dependence violation. Note that each cache line is augmented with two bits indicating whether the line has been speculatively loaded or modified, and each processor maintains an epoch number and a flag indicating if specu-

(a) Original Code

```
// Execute epoch i.
epoch_body(i);
```

$\Longrightarrow$

(b) Steps Involved in TLDS

```
set_epoch_number(i);
become_speculative();

epoch_body(i);

// Now attempt to commit the results.
wait_until_oldest_epoch();
become_nonspeculative();
if (speculation_succeeded) {
   make_speculative_stores_visible();
} else { // speculation failed
   squash_any_child_threads();
   // Recover by re-executing the epoch
   epoch_body(i);
}
commit_speculatively_forwarded_values();
make_child_oldest_epoch();
```

Figure 8. Further details on TLDS execution.

lation has failed. Assume that epoch 5 is the oldest epoch and therefore is not speculative. First, PROCESSOR 2 loads the value from p, causing the *speculatively loaded* bit for that cache line to be set. PROCESSOR 1 then stores to q, which points to the same location as p, causing an invalidation to be sent out along with the epoch number. When PROCESSOR 2's L1 cache receives the invalidation, it sets the violation flag because it notices that the following three conditions are true: the given cache line is present in the cache, the line has been speculatively loaded, and the invalidation came from a sequentially earlier epoch. When PROCESSOR 2 eventually attempts to commit its speculative work, it will notice that the violation flag has been set, and will therefore initiate the recovery process.

Figure 8 provides further detail on the steps involved in TLDS execution. Although the amount of pseudo-code in Figure 8(b) may appear to be substantial, these steps would in fact be implemented with just a small number of new machine instructions. Also note that Figure 8(b) is overly conservative in several ways, and is intended simply for the purpose of illustration.

Before executing the epoch body, the thread first sets its epoch number and indicates that it is speculative. (In reality, the thread can postpone becoming speculative until just before its first speculative memory access.) After completing the epoch body, the thread must wait until all previous epochs which this epoch might depend upon have successfully committed their results to memory before it can be certain that its speculation was successful. We refer to this state as being the "oldest" epoch.[2] By testing the violation flag at this point, the thread can determine whether its speculation was successful.

To prevent speculative threads from corrupting memory, and to eliminate write-after-write (WAW)[3] and write-after-read (WAR) dependences, we postpone making speculative stores globally visible until we are certain that speculation succeeds. After an epoch has confirmed successful speculation, it must obtain exclusive access to each line it has speculatively modified, at which point the lines may be allowed to safely leave the L1 cache and propagate throughout the rest of the memory hierarchy. (In reality, this process can be optimized to avoid sending out a burst of invalidation requests—e.g., if the speculatively modified line is already in an exclusive state, there is no need to send out any further requests.) All speculative stores must be globally performed before the given epoch can signal the next epoch that it has become the "oldest". Since the synchronization indicating the "oldest" epoch is serialized, the memory consistency of the original sequential program will be preserved. Also, since the acts of waiting upon and signaling the "oldest" state are *acquire* and *release* operations, respectively, we can exploit relaxed memory consistency models [6, 11].

Figure 8(b) shows that in response to failed speculation, the thread immediately squashes any subsequent epochs and then recovers by re-executing the epoch body non-speculatively. This process could be optimized in several ways. First, an epoch could be interrupted and begin the recovery process immediately upon a data dependence violation. Second, it is not necessary to squash all subsequent epochs—instead, one might selectively squash only those which are affected by the dependence violation. Finally, one could re-execute only the portion of the epoch body which is dependent upon speculative loads.

Note that in order for a speculative thread to forward data to another processor (as discussed in Section 3.3), it must explicitly perform a *non-speculative* store. To prevent this store from corrupting program state, the compiler will explicitly create shadow copies in memory of all forwarded values—once speculation succeeds, a thread will explicitly copy its shadow state to the real location. Since we only forward scalars, the memory overhead of creating shadow copies should be relatively small.

If a speculatively-loaded line is replaced from the cache while a thread is still speculative, the violation flag is set immediately since we can no longer track dependence violations. In general, whenever the hardware panics, it can always conservatively set the violation flag, since this only impacts performance and not correctness.

A potential drawback of tracking data dependences at a cache line rather than a word granularity is performance loss due to "false" dependence violations—i.e. when separate parts of a line are read and written, thereby triggering the recovery mechanism unnecessarily. To quantify the impact of false dependences, we measured how the run lengths

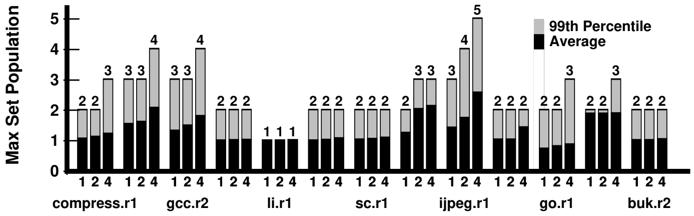
[2] Note that the term "oldest" epoch is overly simplistic in this context—e.g., we can have multiple "oldest" epochs at the same time.

[3] For WAW dependences, a bit must be kept per word in the cache line which indicates whether the word has been speculatively modified. This will allow two speculatively modified versions of a cache line to be combined properly, given the epoch number ordering between them.

(a) Average storage required for buffering speculative accesses (both loads and stores)

Application	Region	Unique 32B Lines Accessed	Total Storage (kB)
compress	r1	10.6	0.33
gcc	r1	43.7	1.37
	r2	34.2	1.07
espresso	r1	4.1	0.13
li	r1	1.6	0.05
	r2	7.1	0.22
sc	r1	4.4	0.14
m88ksim	r1	46.9	1.47
ijpeg	r1	139.6	4.36
perl	r1	11.0	0.34
go	r1	8.6	0.27
buk	r1	4.0	0.13
	r2	4.0	0.13

(b) Degree of associativity needed to avoid cache replacements



(c) Number of victim cache entries needed to avoid failed speculation

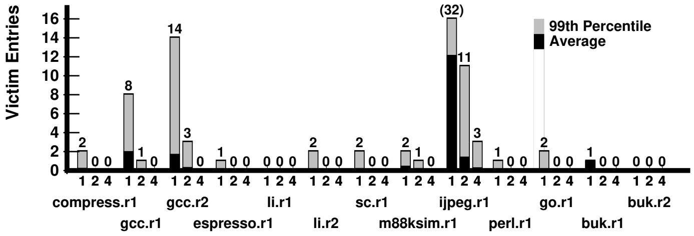


Figure 9. Statistics on whether the primary data cache is sufficient to buffer speculative state. A 16KB cache with 32B lines is used in all cases. The numbers below the bars indicate the associativity.

shown in Figure 3(c) degraded with 128 byte granularities rather than 32 byte granularities. While some decrease was observed for the very long run lengths, they remained sufficiently large that we would expect no performance loss on up to eight processors. Hence our approach is a viable technique for detecting unsafe data speculation.

4.4. Using the Cache to Buffer Speculative State

To simplify software's job of recovering from unsafe speculation, we rely on hardware to buffer speculative store results until they can be safely committed to memory. Rather than building a separate buffer devoted entirely to data speculation, it would be attractive to use the *cache* itself as the speculative buffer. The basic idea is that any speculatively modified lines in the primary data cache will be specially marked (e.g., using the "speculatively modified bits shown in Figure 7) such that their side effects will be prevented from propagating to the rest of the memory system until speculation succeeds. In effect, the coherence protocol treats speculatively modified lines as though they are not "dirty", and therefore they will not be written back; this can be implemented in a number of different ways. (Note that if we attempt to speculatively modify a line which is in fact dirty, we must write it back to memory before the speculative store can proceed.)

Since data speculation fails (and thus invokes recovery) if any lines which have been speculatively loaded or modified are forced out of the cache, a key question is whether the primary data cache has sufficient capacity to hold all of the lines accessed by a typical epoch. (Note that such evictions will not result in deadlock since the memory operations of the oldest epoch are always treated as being non-speculative, and hence it always makes forward progress.) As we see in Figure 9(a), all of our regions require less than 5KB of buffering on average. While this is encouraging, the real question is what degree of associativity is required in a realistic cache to avoid mapping conflicts. Figure 9(b) shows the maximum number of lines accessed which map to a given *set* of a 16KB primary data cache with 32B lines and various associativities, both for the average epoch and the 99th percentile case. If this maximum exceeds the associativity, then at least one line is forced out and speculation fails. As we see in Figure 9(b), a direct-mapped cache does not appear to be sufficient, but a two-way set-associative cache is far more successful at capturing the data: the average epoch almost always fits within the two-way sets, and even the 99th percentile case fits in nine of thirteen regions. Finally, rather than giving up when a speculatively accessed line is displaced, we could instead capture these lines within a small fully-associative *victim cache* [10]. Figure 9(c) shows that by adding a relatively small victim cache (e.g., with four entries) to a 16KB two-way set-associative cache, we can retain nearly all speculatively accessed lines, thus avoiding unnecessary recovery.

In summary, we have seen that instruction scheduling may eliminate the need for fast communication, and that TLDS can be supported through modest hardware modifications by extending the cache coherence algorithm to detect unsafe data speculation, involving software in the recovery process, and enhancing the role of the primary data cache to buffer speculative accesses. Further details on architectural support for TLDS can be found in a technical report [22].

5. Compiler Support for TLDS

The compiler clearly plays a crucial role in exploiting TLDS. In addition to selecting regions of the code to speculatively parallelize and inserting the appropriate TLDS primitives, we have also seen that the compiler has an important role in *optimizing* the code by removing data dependences and maximizing parallel overlap if we are to achieve the full potential of TLDS. We briefly discuss some of these compiler issues in this section.

5.1. Choosing Speculative Regions

The first step in compiling for TLDS is choosing the appropriate speculative regions to parallelize. Our goals here are twofold: (i) maximizing the fraction of total execution which is parallelized, and (ii) achieving the best speedups within each speculatively parallelized region. We performed this step by hand in our experiments as follows: we used profiling information to identify where the program was spending most of its time, and we then attempted to find the largest surrounding regions which did not contain obvious data dependences that would prevent TLDS from working. To maximize program coverage in a cost-effective manner, the compiler could also make use of control-flow profiling information to focus its efforts on the sections of code which account for the largest fractions of total execution time. Achieving the best region speedups automatically involves a number of different issues.

First, we would like to choose regions where the epochs are large enough to amortize the costs of thread management and communication, but not so large that the amount of space for buffering speculative state becomes a problem. In our experiments, we observed that epoch sizes ranging between 20 and 10,000 dynamic instructions worked effectively. The compiler has considerable flexibility in adjusting the epoch sizes within a region. If the epochs are too large, the compiler can statically split them up into smaller pieces (e.g., divide a loop body in half). If the epochs are too small, the compiler can merge consecutive epochs to form larger epochs (e.g., unroll a loop and fuse consecutive iterations).

Second, we would like to maximize the probability of successful speculation by avoiding regions with problematic data dependences—i.e. cases where RAW dependences occur frequently between consecutive epochs which can neither be eliminated nor explicitly forwarded to yield significant parallel overlap of epochs. In our experiments, we observed that increasing the scope of a region (e.g., moving from an inner to an outer loop nest) can either increase or decrease the number of problematic data dependences. To understand this behavior, the compiler should analyze data dependences to the extent possible. Our experiments relied only on understanding data dependences due to *scalar* variables, which should be feasible to analyze. Although understanding pointer addresses is not a requirement of TLDS, having more dependence information will help the compiler make better decisions regarding cost-benefit tradeoffs.

5.2. Optimizing Speculative Regions

After selecting candidate regions to speculatively parallelize, the compiler should then *optimize* their performance in the following ways. The first step is eliminating as many dependences as possible—e.g., induction variables (which can be expressed as a function of the epoch number), reduction operations (which can be replaced by local operations), and dependences inside library routines (which can be explicitly parallelized). We saw the importance of this step earlier with case "**O**" in Figure 3(c). Next, for any repeated scalar dependences which the compiler can recognize but not eliminate, it should insert code to explicitly forward these values between epochs, and (most importantly) *reschedule* the epochs to minimize the critical path. As we saw earlier in Figure 5, the performance gain offered by rescheduling can be dramatic.

Finally, the compiler must insert calls to the TLDS primitives and create the recovery code, such as the example in Figure 8 illustrates. Note that although the hardware restores the memory state during recovery by discarding any speculative stores, it is the responsibility of software to restore any necessary register state. The compiler can reduce the recovery overhead by re-executing only the portion of the epoch which depends on speculative load results, and by initiating the recovery process early either through an interrupt mechanism or by polling the violation flag early.

One exciting aspect of TLDS is that the compiler does not have to live with its mistakes. Since software directly observes every dependence violation at run-time, it is straightforward to collect statistics on the rate of unsafe speculation and use this information to adapt the parallelization strategy either on-the-fly or else during the next recompilation.

In summary, although automatically parallelizing non-numeric codes is still a non-trivial task, it is at least *feasible* with TLDS, in contrast with the hopelessly restrictive model of statically proving that threads are independent.

6. Conclusions

To enable a potential breakthrough in the compiler's ability to automatically parallelize non-numeric applications, we have investigated *thread-level data speculation* (TLDS)—a technique which allows the compiler to safely parallelize code in cases where it believes that dependences are unlikely, but cannot statically prove that they do not exist. Our experimental results demonstrate that with realistic compiler support, TLDS can potentially offer compelling performance improvements—i.e. overall program speedups ranging from 15% to nearly fourfold on four processors in seven of ten cases—for applications where automatic parallelization would otherwise appear infeasible. Since our hand analysis was not exhaustive, we believe that even larger speedups may be possible by applying TLDS more extensively.

To translate the potential of TLDS into reality, we have investigated and quantified the tradeoffs in providing hardware and compiler support for TLDS. We find that only modest hardware modifications to a standard single-chip multiprocessor are needed: the cache coherence protocol can be extended to detect RAW dependence violations and inform software when they occur to invoke recovery actions; the cache itself can be used to buffer speculative memory accesses; and although extremely fast interprocessor communication will offer some benefit, we can still achieve good performance by communicating through a shared L2 cache. Due to the distributed nature of this hardware support, we do not expect it to degrade the performance of applications which do not exploit TLDS. We have also discussed and evaluated the compiler optimizations which are necessary to effectively exploit TLDS. Our goal now is to implement the full compiler support for automatically parallelizing non-numeric applications using TLDS, and to explore the architectural issues in more detail. Based on the encouraging results in this study, we advocate that future single-chip multiprocessors provide the modest support necessary for TLDS.

7. Acknowledgments

This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada, and by a grant from IBM Canada's Centre for Advanced Studies. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

References

[1] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-Directed Page Coloring for Multiprocessors. In *Proceedings of ASPLOS-VII*, pages 244–255, October 1996.

[2] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proceedings of ISCA 28*, December 1995.

[3] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin – Madison, 1993.

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

[5] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of ASPLOS-VI*, pages 183–195, October 1994.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA 17*, pages 15–26, May 1990.

[7] G. Goff, K. Kennedy, and C. W. Tseng. Practical dependence testing. In *Proceedings of PLDI '91*, pages 15–29, June 1991.

[8] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. Technical Report 1334, Computer Sciences Department, University of Wisconsin-Madison, July 1997.

[9] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative Disambiguation: A Compilation Technique For Dynamic Memory Disambiguation. In *Proceedings of ISCA 21*, pages 200–210, April 1994.

[10] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of ISCA 17*, pages 364–373, May 1990.

[11] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of ISCA 19*, pages 13–21, May 1992.

[12] A. Nicolau. Run-time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38:663–678, May 1989.

[13] R. S. Nikhil and Arvind. Can Dataflow Subsume Von Neumann Computing. In *Proceedings of ISCA 16*, pages 262–272, May 1989.

[14] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*, October 1996.

[15] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University, May 1997.

[16] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops With Privatization and Reduction Parallelization. In *Proceedings of PLDI '95*, pages 218–232, June 1995.

[17] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case For Processor/Memory Integration. In *Proceedings of ISCA 23*, May 1996.

[18] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[19] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[20] M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

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

[22] 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.

[23] J. G. Steffan and T. C. Mowry. The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors. Technical Report CSRI-TR-356, Computer Systems Research Institute, University of Toronto, February 1997.

[24] 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.