The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors

J. Gregory Steffan and Todd C. Mowry

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
{steffan,tcm}@eecg.toronto.edu

Technical Report CSRI-TR-350
February 1997

Abstract

To fully exploit the potential of single-chip multiprocessors, we must find a way to parallelize *non-numeric* applications. However, compilers have had little success in parallelizing non-numeric codes due to their complex data 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 may violate program correctness. Experimental results demonstrate that TLDS can offer significant program speedups. We also demonstrate that through modest hardware extensions, a standard single-chip multiprocessor could support TLDS by augmenting the cache coherence scheme to detect dependence violations, and by using the primary data caches to buffer speculative state. We quantify the impact of this implementation on performance, and we also evaluate the compiler support necessary to exploit TLDS.

1 Introduction

As the number of transistors that can be integrated onto a single VLSI chip continues its dramatic rate of increase, processor architects are faced with the pleasant challenge of finding the best way to translate these additional resources into improved performance. While there have been several proposals [6, 7, 14, 15, 23, 28, 27], perhaps one of the more compelling options is to integrate *multiple processors* onto a single chip [5, 20, 17, 14, 7]. From a VLSI perspective, single-chip multiprocessors are attractive because their distributed nature allows the bulk of the interconnections to be localized, thus avoiding the delays associated with long wires [20].

While single-chip multiprocessing will clearly increase computational throughput, it will only reduce the execution time of applications that can exploit parallelism. Hence 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 *numeric* applications [2, 10, 22]. For single-chip multiprocessing to have an impact on the majority of 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 ana-

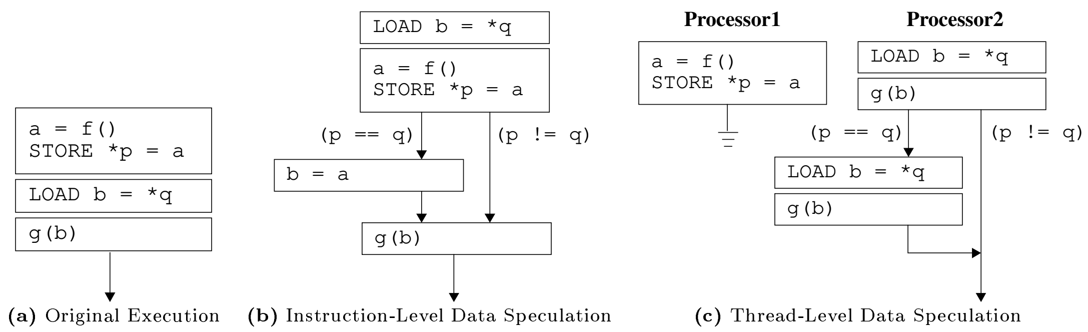


Figure 1: Examples of data speculation.

lyzing array accesses within `DO` loops [10, 16, 21]. Although progress has been made in this area, the problem is considerably more 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 by allowing it to view parallelization solely as a cost/benefit tradeoff—i.e. if the compiler believes it is *likely* that two threads are independent, it can go ahead and parallelize them without worrying about violating program correctness. In this paper, we quantify the performance advantages of a technique which provides this flexibility: *thread-level data speculation*.

1.1 Thread-Level Data Speculation

To maximize parallelism, we often want to perform loads early so that operations which depend on them can be executed concurrently. In contrast, there is less pressure to perform stores early, since they do not produce register results.[1] 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 it is difficult for the compiler to precisely analyze load and store addresses in non-numeric applications, 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 modified 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 speculative operation either through a simple software check (i.e. compare `p` with `q`), or with the help of hardware support such as the "memory conflict buffer" [9]. Previous studies have demonstrated that instruction-level data speculation can significantly increase the amount of instruction-level parallelism (ILP) within a program [9, 12].

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. Although

[1] Since the goal of the compiler is to keep values that will be used again soon in registers, stores should typically only occur when a value will not be loaded again in the near future.

supporting thread-level data speculation is more complicated than supporting instruction-level data speculation (as we discuss in detail later in Section 4.2), our results in this paper demonstrate that TLDS can significantly increase the amount of *thread-level parallelism* (TLP) within non-numeric programs, and can be implemented in a cost-effective manner.

1.2 Related Work

While instruction-level data speculation has received much attention [9, 12, 26], there has been relatively little work so far on thread-level data speculation. In this section, we briefly discuss the two most relevant works to this study, the *privatizing doall* test [22] and the Wisconsin multiscalar architecture [7, 27].

1.2.1 The Privatizing Doall Test

Padua et al. [22] have devised a method of parallelizing loops for numeric codes in the presence of ambiguous data dependences. Their approach, called the *privatizing doall* (PD) test is entirely software-based, allowing the compiler to parallelize loops without fully disambiguating all memory references. For a given loop, the PD test is performed on each shared variable with ambiguous references by creating corresponding *shadow arrays* to track read and write accesses. At the end of the parallel execution of the loop, the shadow arrays are examined. If any cross-iteration data dependences were violated, the loop is re-executed sequentially. Otherwise, we know that the parallel execution of the loop was correct.

Although a purely software-based approach is attractive, there are two shortcomings to the PD test. First, the PD test requires the creation of shadow storage for all shared data, and is therefore not applicable to most non-numeric codes due to their complex data structures and extensive use of heap-allocated objects. Second, the PD test does not extract any parallelism in the presence of a *single* cross-iteration RAW dependence, since the loop is re-executed sequentially in such cases. Because of this, the PD test is only effective in parallelizing a narrow class of loops.

1.2.2 The Multiscalar Architecture

The most relevant work to this study is the Wisconsin multiscalar architecture [7, 27, 8]. This architecture performs aggressive control and data speculation through the use of devoted hardware structures and complex forwarding mechanisms. This section describes the multiscalar architecture and how it executes an application, beginning with the compilation process.

During compilation for multiscalar execution, a program is broken up into small *tasks*. A task may consist of a only few instructions or perhaps several basic blocks. The compiler inserts a bit-vector called the *create mask* into every task denoting which registers are live at the end of the task. Hardware then uses this information to forward register values between tasks.

The multiscalar architecture speculatively executes tasks in parallel, performing control speculation by executing target tasks before the corresponding branches have been resolved. Data speculation is also supported, allowing tasks which are possibly data-dependent to be executed in parallel. We now discuss the hardware mechanisms which support these features.

Processors in the multiscalar architecture are arranged in a ring, and each processor is tightly-coupled with its two neighbouring processors. At any given time during execution, one of the processors is considered the *head* of the ring. This is the processor which is executing the oldest task, and its work is not considered speculative since it cannot depend on previous tasks.

There are two main benefits to the ring architecture. First, the ordering between tasks is implied by the order of the processors in the ring, thus making it trivial for the task manager to terminate the appropriate tasks when a data dependence violation occurs. Second, the tight-coupling of adjacent processors in the ring simplifies register forwarding, since register values must be forwarded between consecutive tasks which are executed by consecutive processors.

Register values are forwarded between tasks by hardware using the create mask described earlier. The last definition of each register which is live at the end of the task is forwarded to the next

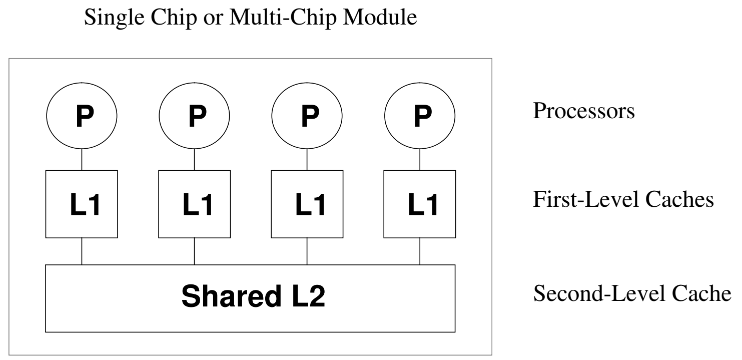


Figure 2: Block diagram of a single-chip multiprocessor.

processor in the ring. In the receiving processor, each register has a *busy bit* which is set when the register value arrives, thus synchronizing the communication of register values between tasks.

In order to support data speculation, the multiscalar architecture includes the *address resolution buffer* (ARB) [8] which performs dynamic memory disambiguation. The ARB sits between the processors and the first-level cache, and all memory accesses are filtered through it. When a store to memory occurs, the store address and the value are kept in one of the ARB's associative entries for the corresponding processor. When a memory access occurs, the ARB is searched for accesses to the same address–if the ARB notices that a store and load to the same address occur out of sequence, then all speculative tasks including and beyond the violating task are terminated. This mechanism allows tasks to be executed in parallel such that either memory dependences are satisfied or speculation fails.

Although the multiscalar paradigm allows control and data speculation, it does so at the cost of an architecture which is devoted to that mode of execution. The ring layout of processors is beneficial for forwarding data between consecutive tasks, but it would not be efficient in executing a conventional parallel program. The ARB is a large and complex structure. Since an associative search must be performed for most memory accesses, latency through the ARB will be longer than that of an ordinary first-level cache. This means that the multiscalar architecture will not be efficient at executing multiprogramming workloads or even conventional parallel programs. However, experimental results for the multiscalar architecture show that speculative execution is a promising way to improve the performance of non-numeric applications using multiple processors. An important question is whether speculative execution may be supported in a non-devoted architecture which is also effective on other types of workloads.

1.3 Objectives

Given a "generic" single-chip multiprocessor as a starting point, where processors share a second-level cache and the individual primary caches are kept coherent (as shown in Figure 2), this study attempts to answer the following questions. First, does adding support for TLDS significantly improve the performance of non-numeric applications through increased thread-level parallelism? We quantify this answer through a detailed study of integer applications taken from the SPEC92 and SPEC95 benchmarks suites, and find that significant improvements can be achieved. Second, can we provide the necessary hardware support for TLDS in a cost-effective manner—i.e. without a centralized structure such as the ARB? We propose a simple extension to an invalidation-based cache coherence protocol which allows us to detect unsafe speculation, and we rely on software (rather than just hardware) to recover the correct program state in such cases. We also propose using the cache to buffer speculative writes, and find that this is a feasible approach. Finally, what compiler support is needed to effectively exploit TLDS? We provide a detailed breakdown of what transformations are necessary, and how much they help performance.

1.4 Overview

The remainder of this paper is organized as follows. We begin in Section 1.5 with an example which illustrates TLDS. Section 2 describes the experimental methodology. Section 3 quantifies the potential performance impact of TLDS on idealized hardware, and investigates the minimum compiler and hardware support required to properly exploit TLDS.

Section 4 presents possible hardware support for TLDS, including thread management issues, extending the cache coherence mechanism to detect RAW dependence violations, and using the cache to buffer speculative state. The impact of this implementation on the results for idealized hardware is also discussed. The compiler support required to exploit TLDS is given in Section 5–basic transformations and support are described, as well as performance-improving optimizations.

Finally, Section 6 concludes the paper by summarizing the potential for extracting parallelism with TLDS in tightly-coupled multiprocessors, and describing possible future work for TLDS.

1.5 An Example: `Compress`

We have demonstrated how TLDS works for a small code fragment. In this section, we explore how TLDS applies to a real application, including the characteristics of the application which make it suitable for exploiting TLDS, the support required, and the optimizations which allow the application to obtain the full benefits of TLDS.

First, we briefly introduce some terminology. We say that TLDS extracts parallelism 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 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 a reasonable amount of work so that the overheads of thread creation and data communication will be relatively small. For TLDS to improve an application's execution time, the speculative regions must constitute a significant fraction of overall execution time.

The `compress` application in the SPEC92 benchmark suite is a good candidate for exploiting TLDS.[2] 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 89 dynamic instructions. While this 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 3(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.

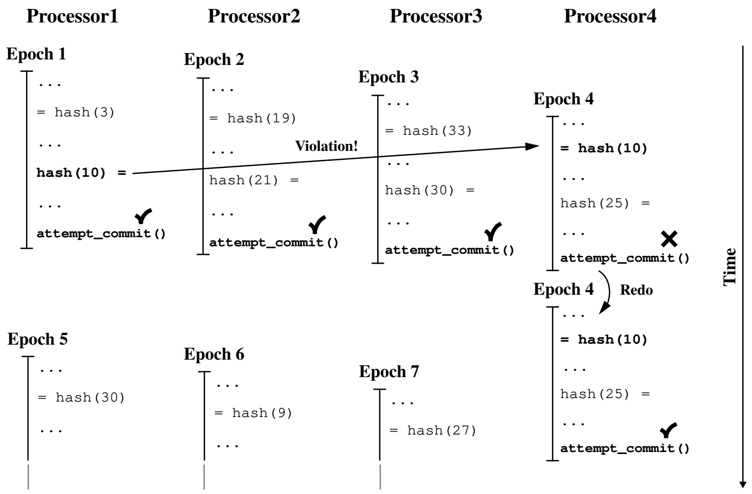
Figure 3(b) illustrates how `compress` can be parallelized using TLDS, where each epoch (i.e. loop iteration) is executed as a separate thread. Since the threads are speculative, they must buffer any stores to memory until they are certain that it is safe to commit their results. If a RAW dependence violation is detected, the thread can recover by re-executing its epoch. As we can see in Figure 3(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 `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 they can be eliminated 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. Loop-carried dependences also exist inside `getchar()` and `putchar()`—these could

[2] The same is true for the SPEC95 version of `compress`, which behaves quite similarly.

```
while (continue_cond){
  ...
  x = hash(index1);
  ...
  hash(index2) = y;
  ...
}
```

(a) `Compress` Pseudo-Code



(b) TLDS Execution of `Compress`

Figure 3: Example TLDS execution.

also be eliminated through parallel implementations of the I/O routines.[3] 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). Two other variables are sometimes modified within the loop and their values are often used: `offset`, which is a scalar, and `bp`, which is a pointer used to index a buffer. The compiler should be able to recognize the scalar data dependence, and with proper hardware support could forward this value directly to consuming threads whenever it is modified. While the pointer dependence could also be forwarded, recognizing it would be more difficult for the compiler.

Thus we see that a range of different compiler optimizations, with varying degrees of complexity, are useful for getting the full benefit of TLDS—we will break down the importance of each of these techniques later in Section 3.

2 Experimental Framework

This section describes our experimental methodology, including the applications we study, our method of finding speculative regions, the simulation environment, and the performance metrics we measure.

2.1 Finding and Simulating Speculative Regions

To quantify the potential for TLDS in non-numeric codes, we examine a set of real non-numeric applications in which potential *speculative regions* are identified by hand. Table 1 summarizes the ten non-numeric applications studied, which are taken from the SPEC92 [4], SPEC95 [3], and NAS Parallel [1] benchmark suites. These applications were compiled with `-O2` optimization using the standard MIPS compilers under IRIX 5.3, and the source code and resulting object files were not modified in any way.

Table 2 lists the speculative regions analyzed in this study. To locate speculative regions, the IRIX `prof` utility was used to identify regions that account for a large portion of total execution

[3]While parallel input routines are straightforward, parallel output routines are somewhat trickier. To reconstruct the output stream correctly, the epoch number may be passed to the `putchar()` routine to provide a total ordering.

Table 1: Integer benchmarks.

Suite	Benchmark	Input Data Set *(size:name)*	Description
SPEC92	compress	ref:in	Performs data compression
	gcc	ref:stmt.i	C compiler
	espresso	ref:bca.in	Boolean function minimizer
	li	ref.li-input.lisp	Lisp interpreter
	sc	ref:loada1	Spreadsheet calculator
SPEC95	m88ksim	test:ctl.raw	Simulator for a 88100 microprocessor
	ijpeg	train:vigo.ppm	Jpeg image processor
	perl	test:primes.pl	Perl script interpreter
	go	train:2stone9.in	Plays the game "go" against itself
NAS Parallel	buk	N = 65536	Implementation of the bucket-sort algorithm

Table 2: Benchmarks and speculative regions

Benchmark	Region	Region Description *(src_file:function():loop_type, src_line)*	Average Dynamic Instrs per Epoch	% of Total Dynamic Instrs
compress	r1	compress.c:compress():while, line 784	89	99.9
pppgcc	r1	reload1.c:reload():for, line 421	1092	8.1
	r2	flow.c:life_analysis():for, line 680	1593	4.0
espresso	r1	compl.c:compl_lift():for, line 269	32	19.4
li	r1	xldmem.c:sweep():for, line 417	19	21.9
	r2	xldmem.c:vmark():for, line 399	286	51.2
sc	r1	interp.c:RealEvalAll():for, line 1001	36	69.3
m88ksim	r1	go.c:goexec():while, line 118	1232	99.3
ijpeg	r1	jccolor.c:rgb_ycc_convert():while, line 132	7514	10.0
perl	r1	regexec.c:regmatch():while, line 544	67	35.8
go	r1	g29.c:getefflibs():for, line 636	80	6.8
buk	r1	buk.f:bucksort():do, line 137	26	16.5
	r2	buk.f:bucksort():do, line 124	18	11.4

time. These regions were then inspected by hand to determine whether they were good candidates for exploiting TLDS. If so, these regions were explicitly identified to the simulator through their instruction addresses. The simulator reads sequential execution traces generated by the MIPS `pixie` utility [25] to measure the exact data dependences between epochs in each speculative region. For the sake of simplicity and simulation speed, instructions are assumed to execute on an ideal single-issue processor–in other words, we assume that one instruction is issued every cycle.

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` (notice in Table 2 that the two regions in `gcc` account for only 12% of execution time). We believe that an automated feedback-directed tool would have located a larger set of speculative regions, thus improving the overall effectiveness of TLDS in improving performance.

2.1.1 Descriptions of Speculative Regions

To gain a better understanding of the types of regions which are suitable for TLDS, we now give a brief description of each speculative region.

`compress.r1`: This speculative region is the main `while` loop in the `compress()` routine, comprising 99.9 % of overall execution time, as described earlier in Section 1.5. The loop performs data compression using hash tables to maintain the compressed patterns.

`gcc.r1`: This `for` loop traverses a list of instructions to check for registers that may be available to use for re-loading spilled pseudo registers.

`gcc.r2`: This `for` loop traverses the basic block list, solving the live variable analysis dataflow problem. This implies that only iterations which propagate changes between basic blocks will have many data dependences between them. As the algorithm converges, data dependences become more sparse and the amount of parallelism available between data dependences increases.

`espresso.r1`: This region is a `for` loop which performs boolean set operations on a data space which is partitioned into two arrays of "cubes". A read-only operation is performed on each cube from one list, the result of which determines if the current cube from the other list is modified. Because of these cross-iteration RAW dependences, known methods are not be able to parallelize this loop.

`sc.r1`: This region is a `for` loop which calculates a value for each cell in a spreadsheet. Since it is not known at compile-time which cells are data dependent, the calculation of cells may not be parallelized using known techniques.

`m88ksim.r1`: `M88ksim` is an architecture simulator, and this `while` loop simulates the data path, calling the `Data_path()` function each simulated cycle and checking for interrupts and errors. Containing 99.3 % of overall execution time, this region of code operates mostly on statically allocated objects although there are several pointer dereferences which would be hard to disambiguate statically.

`ijpeg.r1`: This `while` loop converts rows of "rgb" samples to the JPEG colourspace. Although most of the data structures accessed in the loop are arrays, the use of indirection in the array subscript expressions prevents the loop from being parallelized using known techniques.

`perl.r1`: The `regmatch()` function matches regular expressions using this `while` loop, which occasionally calls `regmatch()` recursively to parse parentheses.

`go.r1`: This `for` loop checks board positions and performs some list operations. Since the lists are allocated dynamically, parallelization is not possible through known techniques.

`buk.r1`: This region is a `do` loop which computes the rank for each key in a set of key values stored in an array. The array which stores the ranks is indexed indirectly, thus making its accesses difficult to disambiguate.

`buk.r2`: This region is a `do` loop which counts the occurrence of each key in an array of key values, but may not be parallelized due to the indirection in the array subscript expressions.

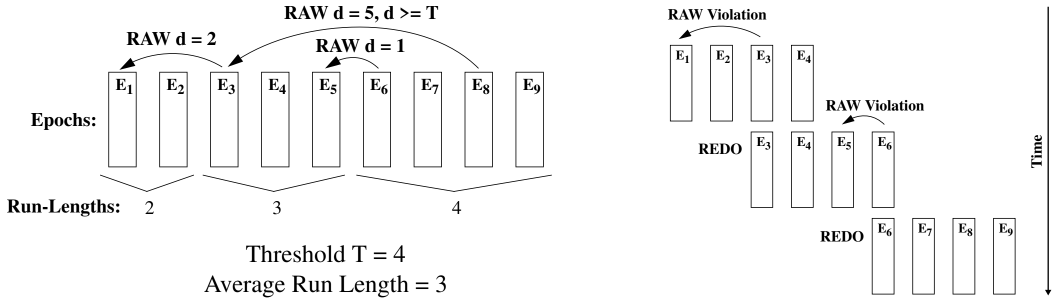
Although parallelism is available in many of these regions, it is evident that known techniques are not effective in parallelizing them due to complex data structures, ambiguous array subscript expressions, or the possibility of cross-iteration data dependences. In Section 3, we demonstrate that TLDS can exploit the available parallelism in these situations.

2.2 Performance Metrics

To estimate the potential performance gain resulting from TLDS, our simulator collects two performance metrics–*run lengths* and *critical path lengths*–which are described below.

2.2.1 Run Lengths

As illustrated earlier in Figure 3, 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. Write-after-read (WAR) and write-after-write (WAW) dependences between epochs may also exist, but these may be satisfied through memory renaming, and hence should not limit performance.



(a) Measurement of Run Lengths

(b) Execution on Four Processors Limited By Run Lengths

Figure 4: Example measurement and execution of run lengths.

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 we are executing epoch E_i, 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 4(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 to see on four processors, as illustrated in Figure 4(b). Hence we will use average run length as a metric for the potential thread-level parallelism.

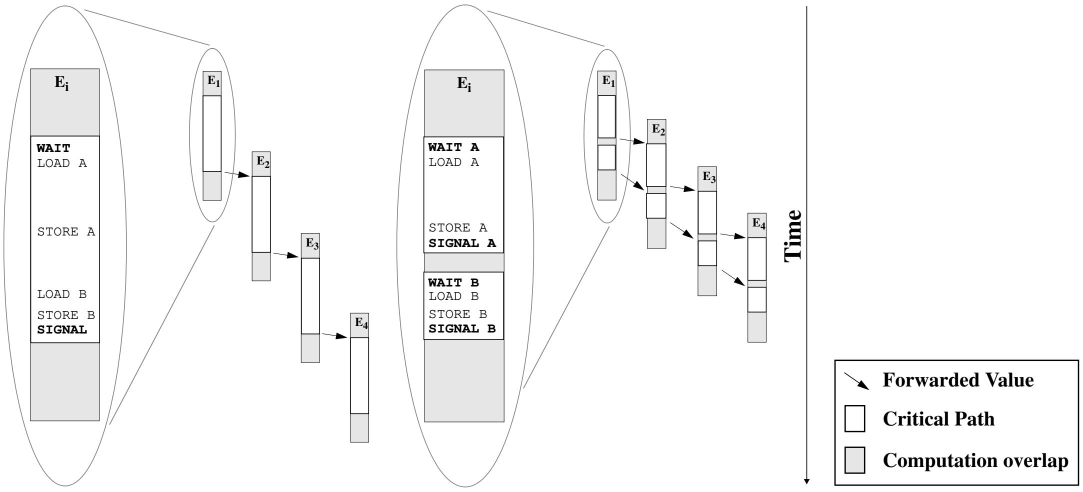
2.3 Critical Path Lengths

In some cases, we may find that RAW data dependences always exist between consecutive epochs, thus limiting the run lengths to one. For example, a scalar variable might be modified and used during each epoch such that it is neither an induction variable, reduction operation, nor anything else that the compiler can optimize away. Although the epochs cannot be fully overlapped in such cases, we may still be able to *partially* overlap their execution by directly forwarding the dependent values between epochs.[4]

Forwarding data requires both a data transfer mechanism and also some form of synchronization, so that the receiving processor knows when the value is ready. Data transfer can occur either through memory or through registers [7, 27], and synchronization can be performed either explicitly using `wait/signal` or implicitly using full/empty bits [6, 14, 24]. 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 Figure 5.

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 course-grain synchronization, the critical path length is straightforward to compute, as illustrated in Figure 5(a). With fine-grain synchronization, there can be *multiple* critical paths through the epochs, as shown in Figure 5(b). In this latter case, the critical path for the epoch is simply the longest of these critical paths (e.g., the critical path for `A` in Figure 5(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

[4]In numeric applications, this is referred to as "doacross" parallelism.



(a) Coarse-Grain Synchronization **(b)** Fine-Grain Synchronization

Figure 5: Examples of forwarding with coarse and fine grain synchronization.

epochs by the overall critical path length.

3 Impact of TLDS on Thread-Level Parallelism

In this section, we present the experimental results which quantify the potential parallelism that can be achieved using TLDS. We begin by focusing on how TLDS can relax memory data dependences. Next, we examine how forwarding data between epochs affects performance. Finally, these numbers are translated into overall program speedups. For an in depth description of how our results are computed, see Appendix A.

3.1 Relaxing Memory Data Dependences

To fully exploit TLDS, we should first apply compiler optimizations to eliminate RAW data dependences whenever possible. Table 3 describes the optimizations that we consider in these experiments, all of which are described in more detail in Section 5. The optimizations in Table 3 are listed roughly in ascending order of complexity. The *base* case (**B**) observes all RAW dependences, and represents the inherent run lengths due to memory dependences in the original code. The *induction variables* (**I**), *library routines* (**L**), and *reductions* (**R**) cases represent situations where the compiler can realistically recognize and eliminate dependences. These cases require only software support, and no special hardware beyond basic TLDS. Finally, in the *scalars* case (**S**), we identify scalar dependences which cannot be optimized away, but which can potentially be accelerated by explicitly forwarding their values between epochs. The level of hardware support for forwarding may vary, and we consider this issue in greater depth later in Sections 3.2 and 3.3.

Table 4 lists the occurrence of the different types of data dependences. Induction variables are present in four of the thirteen regions. Dependences due to library routines are rare, but we will demonstrate that the elimination of these dependences is crucial to the performance of the speculative regions which contain them. Reductions are relatively common, and scalars are abundant in several of the speculative regions.

Figure 6 shows the average run lengths for each region given a threshold (T) of ten outstanding

Table 3: Description of optimization levels used to relax RAW memory data dependences.

Name	Abbrev	Memory Data Dependences Removed	Compiler Support Needed
Base	**B**	None (all RAW dependences observed)	None
Induction Variables	**I**	RAW dependences caused by updating induction variables	Recognize induction variables and compute instead from epoch number
Library Routines	**L**	Case "**I**" plus RAW dependences in I/O and memory allocation library routines	Use parallel versions of library routines instead
Reductions	**R**	Case "**L**" plus RAW dependences caused by reduction operations	Recognize reductions and localize accordingly (e.g., partial sums)
Scalars	**S**	Case "**R**" plus RAW dependences caused by scalars communicated across epochs	Recognize communicated scalars and forward explicitly between epochs

Table 4: Occurrence of memory data dependence types in speculative regions.

Benchmark	Region	Induction Variables	Library Routines	Reductions	Scalars
compress	r1	1	4	2	2
gcc	r1	0	0	0	1
	r2	0	1	0	5
espresso	r1	0	0	0	0
li	r1	0	0	1	0
	r2	0	0	0	0
sc	r1	0	0	1	0
m88ksim	r1	1	0	15	36
ijpeg	r1	0	0	0	0
perl	r1	0	0	0	9
go	r1	0	0	0	5
buk	r1	1	0	0	0
	r2	1	0	0	0

epochs.[5] Starting with the base case (**B**), we see that seven of the thirteen regions have natural run lengths of two or more under TLDS. Of all thirteen regions, the only one which is naturally parallel (i.e. contains no dynamic data dependences) is `ijpeg.r1`—the other twelve regions would be unsafe to parallelize without TLDS. Even for `ijpeg.r1`, it appears unlikely that the compiler could statically prove that the epochs are independent, given the use of indirection in the array subscript expressions; with TLDS support, however, the compiler can safely parallelize this region despite this uncertainty.

Figure 6 also shows how the optimizations listed in Table 3 can enhance run lengths. By eliminating induction variable dependences (**I**), the average run lengths in `buk.r1` and `buk.r2` increase dramatically from one to over twenty. Replacing common library routines (e.g., `getchar`, `putchar`, etc.) with parallel versions (**L**) quadruples the average run length in `compress.r1` from one to four. Table 4 shows that RAW dependences due to library routines are present in `gcc.r2` as well, but the run lengths indicate that they are not performance-limiting. Optimizing reduction operations (**R**) provides a small but measurable improvement in `compress.r1`. Reductions are also found in `li.r1`, `sc.r1`, and most notably in `m88ksim.r1` (as shown in Table 4), but scalar variables must also be optimized to increase the run lengths for these regions. In fact, we see that scalar dependences between epochs (**S**) represent the final barrier to achieving average run lengths of eight or more in `compress.r1`, `gcc.r1`, `li.r1`, `m88ksim.r1`, and `perl.r1`. Although these scalar dependences cannot be fully eliminated (unlike the **I**, **L**, and **R** cases), we can potentially accelerate these cases by explicitly *forwarding* the values between epochs, as is discussed in the next section.

[5] Note that the run length can exceed the threshold, given our definition of run length in Section 2.2.1. Hence run length does not translate directly into speedup, which is something we take into account later in Section 3.3 when we compute region and program speedups.

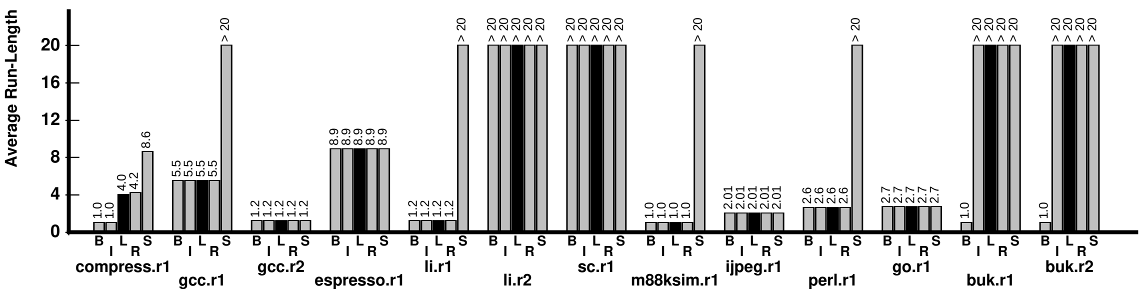


Figure 6: Average TLDS run lengths given various levels of memory dependence optimizations (see Table 3).

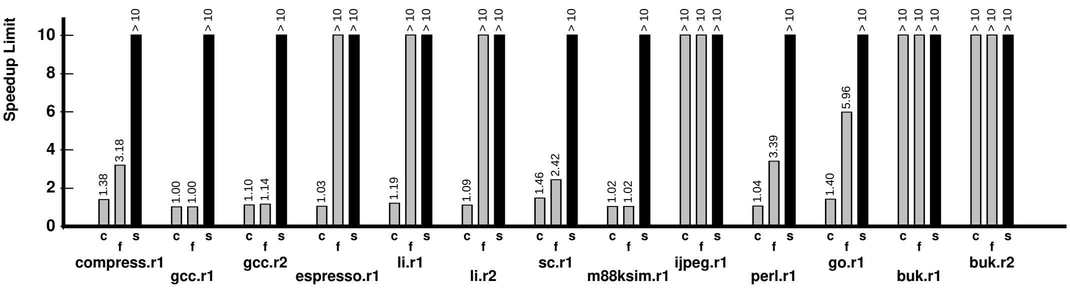


Figure 7: Speedup limits due to forwarding scalars both for memory and register dependences (**c** = coarse-grain synchronization, **f** = fine-grain synchronization, **s** = fine-grain synchronization with aggressive instruction scheduling).

3.2 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 *partially* overlapping the epochs, as described earlier in Section 2.3 and illustrated in Figure 7.

In addition to forwarding scalar memory dependences, we also need to forward any register dependences. Both types of forwarding can potentially be accomplished through the same mechanism. Although the run lengths in Figure 6 ignored register dependences, we take them into account throughout the remainder of this paper as follows. We eliminate two classes of RAW register dependences: those due to induction variables (directly analogous to the "**I**" case for memory), and simple cases where a dependence was only introduced by the compiler converting a `while` loop into a `do-while` loop to create a "landing pad".[6] All other cross-epoch register dependences are forwarded, and are similar to the "**S**" memory case described in Table 3.

3.2.1 Impact of the Synchronization Scheme and Scheduling

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 (as illustrated in Figure 7), 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

[6]For example, the loop test for the `while` loop in `compress.r1` calls `getchar()` to get the next character to be processed by the given iteration. When the compiler converts this `while` loop to a `do-while` loop, `getchar()` is called at the *end* of the loop, thus causing a cross-epoch RAW data dependence. However, this could be trivially fixed through instruction scheduling.

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 forwardable values. In other words, we measured the minimum possible sizes of the non-overlapped portions within epochs.

To estimate the upper bound on speedup given forwarding, we divided the total dynamic instructions in a speculative region by the *critical path length*, as described earlier in Section 2.3. For our initial experiments, we assume that forwarded data can be consumed immediately (e.g., through a shared register file); we consider more realistic forwarding latencies in the next section.

Figure 7 shows the speedup limits due to forwarding. 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 none of these speedups are 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.

3.2.2 Impact of Communication Latency

In addition to the synchronization scheme, another element of forwarding which may limit potential speedups is the communication latency. Extremely large communication latency will degrade performance, but how fast must communication be to obtain reasonable performance? Since we are considering a single-chip multiprocessor, a range of communication mechanisms and thus latencies should be available.

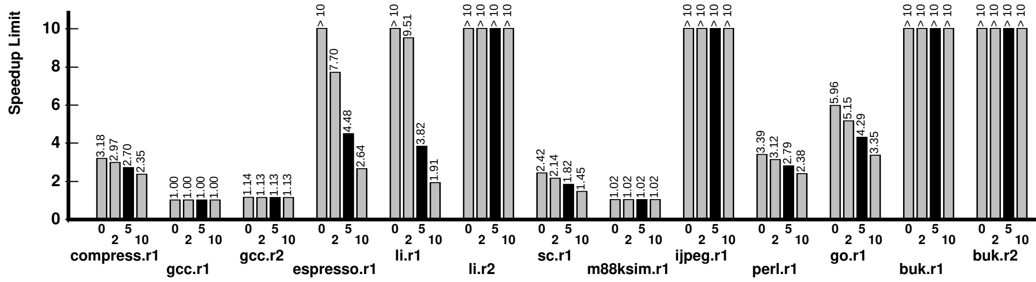
Figure 8 shows the speedup limits for fine-grain synchronization (**f**) and for fine-grain synchronization with aggressive instruction scheduling (**s**), each for a range of different communication latencies. Without instruction scheduling (Figure 8(a)), we see that there are three cases. For six of the regions, the communication latency is not large relative to the overlapped portion, resulting in little change in the speedup limit. Three of the regions show significant sensitivity to communication latency, due to the relatively small epoch sizes of these regions (see Table 2). Finally, the speedup limit due to forwarding may be large enough for communication latency not to be an issue, which is the case for the remaining four regions. With instruction scheduling (Figure 8(b)), all regions demonstrate encouraging speedup limits.

In summary, the importance of communication latency depends on the individual speculative region, although instruction scheduling may eliminate the need for fast communication in cases where it is an issue.

3.3 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 6 and 7. We also account for the time required to recover from unsuccessful speculation by adding the average time to execute an epoch for every speculative epoch which fails. Finally, we account for the fact that parallelism cannot exceed the number of physical processors (four, in this case) at any given time.

(a) Speedup limit for fine-grain synchronization (**f**)



(b) Speedup limit for fine-grain synchronization with aggressive instruction scheduling (**s**)

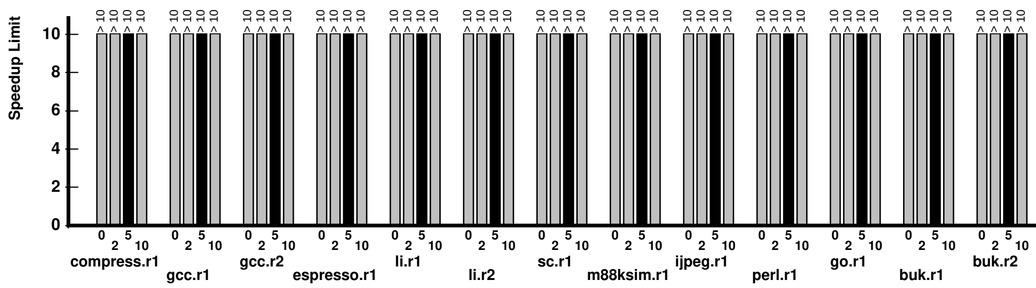


Figure 8: Speedup limits due to forwarding for varying communication latency.

3.3.1 Region Speedups

Figure 9 shows the potential region speedups with two different forwarding latencies: the "*L2*" case corresponds to forwarding data through the normal memory hierarchy (i.e. the shared L2 cache) in ten cycles, and the "*fast*" case uses special forwarding hardware support to communicate data in just two cycles.[7] For each forwarding latency, we show the following five cases. Case "**B**" is a base case with TLDS but no optimizations to eliminate or forward register or memory dependences. In case "**Br**" (and all remaining cases), register dependences are optimized and forwarded, as described earlier in Section 3.2. Case "**R**" also optimizes away some memory dependences, as described in Table 3. Case "**Sf**" forwards memory scalars using fine-grained synchronization, and case "**Ss**" also reschedules instructions to maximize parallel overlap.

As we see in Figure 9, using TLDS without any compiler support to eliminate or forward data dependences (i.e. case "**B**") results in no speedup for these regions. Only two regions (`sc.r1` and `ijpeg.r1`) show improvement when register dependences are optimized and forwarded but memory dependences are not. Eight of the thirteen regions enjoy significant speedups (50% or more) when memory dependences are eliminated under case "**R**". Forwarding memory scalars without rescheduling offers no significant additional improvement, but by rescheduling the code, all but one region potentially achieves an overall speedup of roughly twofold or more on four processors.

Comparing the performance with and without special hardware support for fast forwarding of data, we see that it does make a noticeable difference in performance in five of the thirteen regions, mainly for the "**R**" and "**Sf**" cases. However, when we reschedule the code to maximize parallel overlap (case "**Ss**"), we see that the performance is less sensitive to the forwarding latency. Therefore aggressive compiler scheduling can potentially eliminate the need for expensive forwarding hardware, thus allowing us to forward data through the normal cache hierarchy.

[7] An example of fast forwarding support is the register forwarding mechanism proposed by the Multiscalar architecture [7, 27].

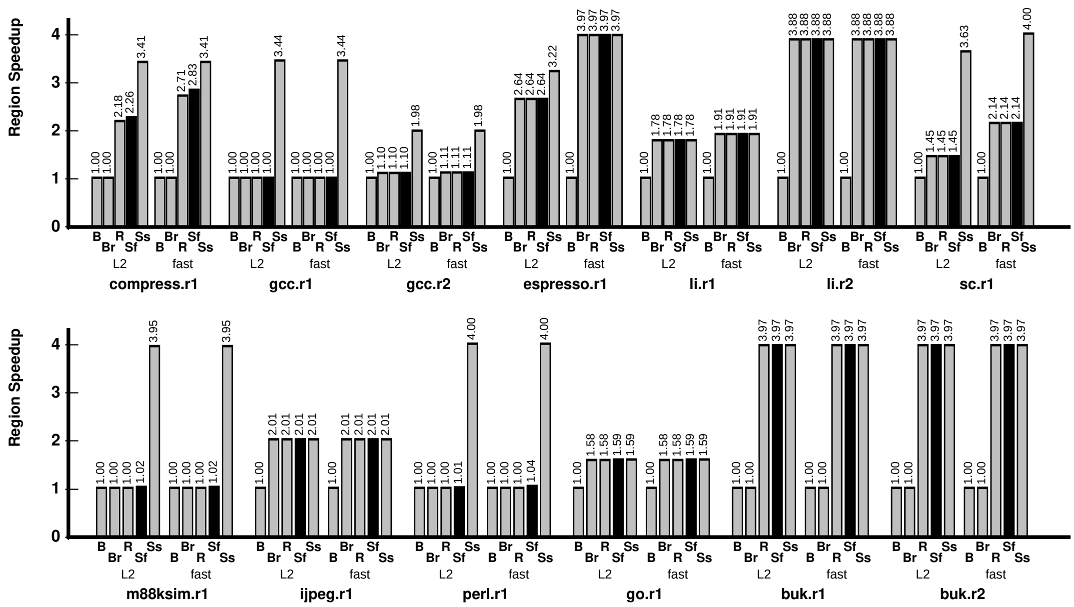


Figure 9: Possible region speedup on four processors (**L2** = forwarding through the shared L2 cache in 10 cycles, **fast** = special hardware support to forward in 2 cycles, **B** = base TLDS hardware with no compiler support, **Br** = register dependences eliminated or forwarded, **R** = case **R** dependences eliminated, **Sf** = case **S** dependences eliminated and fine-grain synchronization, **Ss** = case **S** dependences eliminated and aggressively scheduled code with fine-grain synchronization, see Table 3).

3.3.2 Program Speedups

Given the fraction of total execution time spent in each region (shown earlier in Table 2), we can estimate the potential overall speedup for each application. Figure 10 shows these potential speedups for the three more aggressive optimization levels, and with both types of hardware forwarding support. 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. 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 twofold or more on four processors. Three other applications (`espresso`, `perl`, and `buk`) achieve more modest speedups of 17-37%, and the remaining three applications improve by less than 10%. We believe that our region coverages (and hence program speedups) are pessimistic for many of these applications because finding regions by hand was a very time-consuming process, and we could not begin to do justice to large applications such as `gcc`. We are planning to automate this process in the future, which should help us find more regions.

3.4 Summary

The results in this section have demonstrated that TLDS can potentially provide significant improvements in thread-level parallelism, thus accelerating the performance of non-numeric applications. To achieve the full benefit of TLDS, the compiler should eliminate data dependences whenever possible, explicitly forward cross-epoch dependences that cannot be eliminated, and reschedule the code to minimize any non-overlapped sequences. We have also observed that aggressive instruction scheduling might eliminate the need for performing fast data forwarding between processors, thus allowing us to communicate through the shared L2 cache instead.

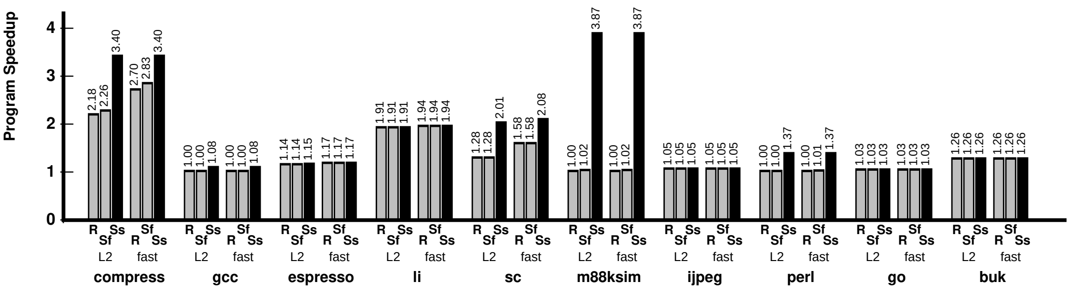


Figure 10: Possible program speedup on four processors (**L2** = forwarding through the shared L2 cache in 10 cycles, **fast** = special hardware support to forward in 2 cycles, **R** = case **R** dependences eliminated, **Sf** = case **S** dependences eliminated and fine-grain synchronization, **Ss** = case **S** dependences eliminated and aggressively scheduled code with fine-grain synchronization, see Table 3).

4 Architectural Support for Thread-Level Data Speculation

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 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 times. The starting point for our design is a standard single-chip multiprocessor where the secondary cache is physically shared and the individual primary data caches are kept coherent to provide a shared memory abstraction.

We begin by discussing 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 Thread Management

In this section, we briefly discuss the architectural mechanisms which are required by TLDS for managing and coordinating the parallel threads. Since there is often flexibility in how the mechanisms might be implemented, our goal is simply to raise the important issues and discuss tradeoffs. Although we do describe a potential implementation of TLDS, the purpose of this is to illustrate how TLDS might be implemented, rather than claiming that this is necessarily the optimal approach.

The first mechanism that is needed is a way to create parallel threads and schedule the epochs onto them. One option is to dynamically create a new thread per epoch, and another is to statically create one thread per processor and have them execute multiple epochs. The disadvantage of the dynamic approach is the runtime overhead of frequent thread creation, which may be reduced to some extent through a lightweight `fork` instruction [19]. A potential advantage of associating one dynamic thread per epoch is that it may simplify the case of having multiple outstanding (uncommitted) epochs per processor, rather than having a single thread maintain the state of several outstanding epochs. (There are a number of subtle issues involved with allowing a processor to have multiple outstanding uncommitted epochs, but such an investigation is beyond the scope of this paper.)

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. There are two important things to note. First, hardware's representation of epoch numbers does not necessarily need to coincide with epoch numbers in software, provided that hardware can still make the appropriate relative comparisons to detect dependence violations. Second, in some cases software might not even need to be aware of epoch numbers—they could instead be maintained implicitly by hardware.

A third mechanism is needed to distinguish speculative versus non-speculative memory accesses. Only speculative loads must be checked for RAW dependence violations, and only speculative store results must be buffered until an epoch successfully completes. For applications which do not exploit TLDS, all memory accesses will be non-speculative, and hence the TLDS hardware support will not be invoked. One possibility is to explicitly mark individual memory instructions as being speculative or non-speculative—while this approach allows us to interleave both types of accesses, it unfortunately requires a new flavour of memory instructions. Another approach is 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 all of its speculative loads were safe. In addition, the thread executing the oldest epoch must be non-speculative.

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 at the end of an epoch to determine safety, and given the fact that the exact interleaving of accesses between threads is not known *a priori* since they run asynchronously, a purely software-based approach of explicitly comparing memory addresses [18] would appear to be impractical. Instead, we propose extending cache coherence schemes to allow hardware to detect potential dependence violations with little overhead, and letting software control the recovery process. We will discuss this mechanism in greater detail later in Section 4.2.

Finally, one aspect of recovering from unsafe speculation which software cannot perform efficiently on its own is rolling back any side effects of speculative stores on memory. To do this effectively, we propose extending the cache functionality so that hardware can buffer speculative store results until they can be safely committed to memory. We will describe this mechanism in more detail later in Section 4.3.

4.1.1 Example

To illustrate how software might interface with these architectural mechanisms, Figure 11 shows how the `compress` benchmark (described earlier in Section 1.5) might be modified to exploit TLDS. In this example, the loop has been modified to execute as a chain of threads, where each thread performs the work of one epoch. Since the hardware will only support a finite number of outstanding threads, we assume that `create_new_thread()` returns a boolean value indicating whether the fork was successful. If so, the current thread simply terminates once it completes its epoch—otherwise, the current thread will execute the next epoch itself. In this example, we assume that hardware implicitly maintains epoch numbers as part of thread creation.

Within the epoch, the thread switches its state to speculative just before its first speculative load (the load of `hash(index1)`). At the end of the epoch, the thread waits until it is the oldest thread to ensure that any writes from earlier threads have been committed—hence any RAW dependence violations would have been detected by this point. While the thread is waiting, it could potentially suspend itself, thus freeing the processor to do other work. Finally, the thread checks whether it is safe to commit its speculative results—if not, it recovers by re-executing the epoch.

4.2 Extending Cache Coherence to Detect Data Dependence Violations

A key component of TLDS support is a mechanism which detects unsafe data speculation—i.e. whenever a cross-epoch read-after-write (RAW) data dependence violation has occurred. To provide this support with minimal hardware cost, we propose a straightforward extension of an invalidation-based cache coherence protocol. Here is the basic intuition behind our scheme. When an earlier epoch performs a load and a subsequent epoch stores to the same address, the coherence protocol would accept the modified cache line once the subsequent epoch commits. Similarly, when two epochs both store to the same address, the coherence scheme will have to combine the two cache lines when the later epoch commits.[8] The case we need to worry about is when a store from an earlier epoch occurs *after* a subsequent epoch has speculatively loaded the same address. When this

[8] This may require a state bit per word in each cache line to mark words which have been modified.

```
while (continue_cond) {
   ...
   x = hash(index1);
   ...
   hash(index2) = y;
   ...
}
```

(a) Pseudo-code for compress

```
EPOCH_START:
  if (continue_cond) {
    successful_fork =
      create_new_thread(EPOCH_START,params);
    do {
      ...
      become_speculative();
      x = hash(index1);
      ...
      hash(index2) = y;
      ...
      wait_to_become_oldest_thread();
    } while (!attempt_commit());
    if (successful_fork)
      terminate_self();
    else goto EPOCH_START;
  }
  else wait_to_become_oldest_thread();
```

(b) TLDS version of compress

Figure 11: Example of a potential software interface for TLDS execution.

store occurs (as with any store), we must *invalidate* any copies of the cache line to maintain cache coherence. We augment the functionality of the invalidation such that if it notices that the line has been speculatively loaded into another cache, it compares the epoch numbers of the store and the speculative load to determine whether a RAW dependence violation has occurred.

To illustrate this mechanism, Figure 12 shows how it would work for the sequence of epochs shown earlier for `compress` in Figure 3. We augment each cache line with two bits indicating whether the line has been speculatively loaded or modified, and we associate with each processor an epoch number and a boolean value indicating whether a dependence violation might have occurred. During step "**4**" in this example, *processor 4* speculatively loads the value of `hash(10)`, thus bringing it into its cache and setting the speculative load bit for that line. Later, during step "**5**", *processor 1* stores to `hash(10)`—during the subsequent invalidation of this line from *processor 4's* cache (step "**6**"), we notice that since the store has an earlier epoch number (one versus four) and the line was speculatively loaded, a dependence violation may have occurred. Hence the violation bit is set for *processor 4* (step "**7**"). When *processor 4* subsequently attempts to commit its speculative results, it will notice that this is unsafe and can recover by re-execute its epoch.

4.2.1 Impact of Cache Line Size On TLDS

A potential drawback of tracking data dependences at a cache line rather than a word granularity is the possibility of "false" dependence violations—i.e. when separate parts of a line were read and written, and hence no true dependence occurred. While these false dependences do not affect program correctness, they can reduce our ability to exploit parallelism by invoking the recovery mechanism when it is unnecessary. To quantify how false dependences might affect TLDS parallelism, we measured how the average run lengths due to memory dependences (discussed earlier in Section 3.1) changed at 32 and 128 byte granularities. As we see in Figure 13, some applications are insensitive to changes in the dependence granularity, while others typically experience decreased run lengths with larger line sizes.[9]

The compiler could potentially avoid false dependences by changing the data layout such that these important objects do not fall within the same cache lines. Also, a more sophisticated hardware

[9]In some cases, run lengths increase somewhat with larger line sizes due to fortuitous circumstances where false violations alter run length boundaries relative to the ten epoch window size.

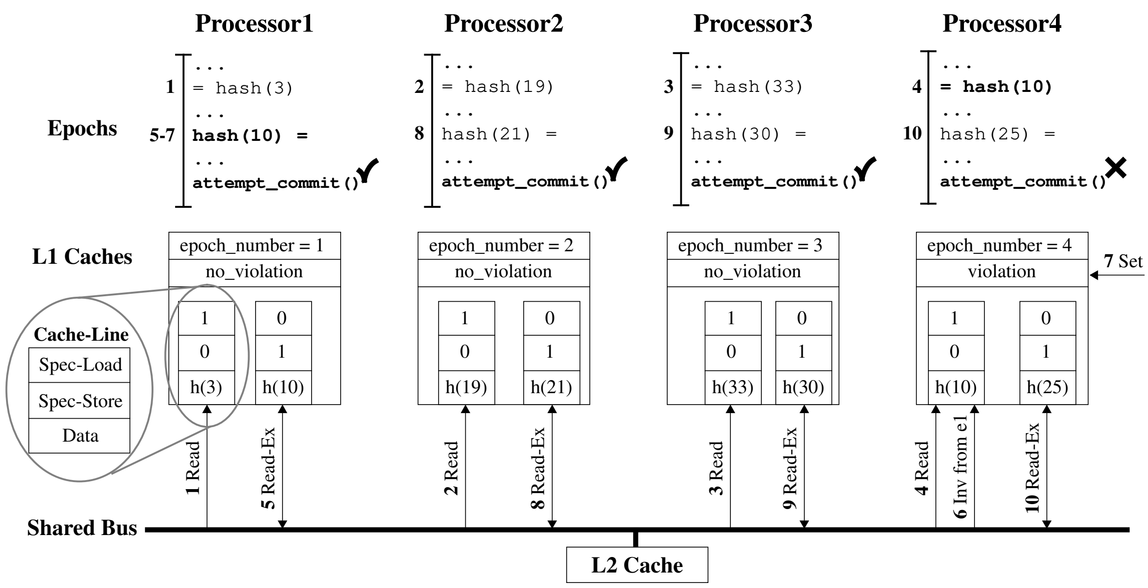


Figure 12: Example of an augmented invalidation-based cache coherence scheme which supports TLDS. (Note: the numbers next to the events indicate the order in which they occur.)

scheme could maintain state information on a per-word basis to further avoid false dependences. In general, a number of refinements on this basic scheme are possible, but the bottom line is that run lengths typically remain long enough that this is a viable approach to detecting unsafe data speculation.

4.3 Using the Cache to Buffer Speculative State

When unsafe data speculation is detected, a thread must recover its original program state. To simplify this process, we would like to buffer any speculative store results until we are certain that they can be safely committed to memory. Rather than building a separate buffer which is devoted entirely to data speculation, it would be attractive if we could simply use the *cache* as our speculative buffer.

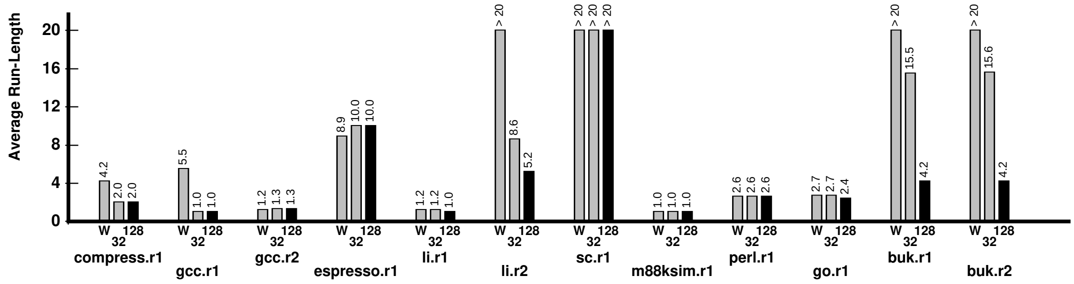
Intuitively, the way this would work is that speculative stores would be free to modify the cache, but the resulting cache lines would conceptually be "locked down" such that their side effects cannot propagate to the rest of the memory system.[10] Speculatively-loaded cache lines must also remain in the cache for the duration of the epoch to track possible RAW dependence violations. Special bits associated with each cache line would indicate this state (e.g., the "speculative store" and "speculative load" bits shown earlier in Figure 12). If data speculation fails, the hardware will squash these speculative lines by marking them *invalid*; if the data speculation succeeds, these stored values can be committed to memory and the lines will return to a non-speculative state. If a line that has been speculatively modified or loaded is forced out of the cache for any reason, its side effects (if any) will be discarded and the *violation* bit will be set, indicating that the data speculation has failed.[11]

In this section, the capacity required to buffer all speculative accesses will be measured, as well as the associativity required to avoid replacements. We will also investigate the impact of adding a victim cache.

[10] There are many ways to implement this "locked down" type of behaviour.

[11] Note that this will not result in deadlock, because the loads and stores of the oldest active epoch will always be interpreted as non-speculative.

(a) Run lengths for optimization level "**R**"



(b) Run lengths for optimization level "**S**"

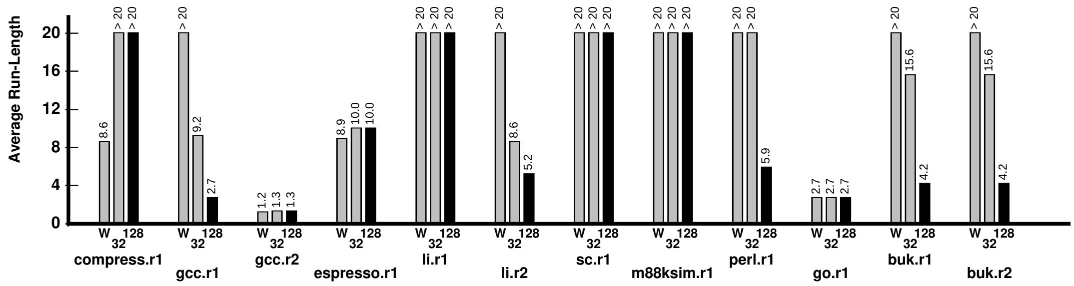


Figure 13: Impact of cache line size on run lengths due to memory dependences.

Table 5: Average amount of storage required per epoch for buffering speculative accesses.

		32B Cache Lines		128B Cache Lines	
Application	Region	Unique Lines Accessed	Total Storage (kB)	Unique Lines Accessed	Total Storage (kB)
compress	r1	10.6	0.33	10.0	1.25
gcc	r1	43.7	1.37	27.0	3.38
	r2	34.2	1.07	20.3	2.54
espresso	r1	4.1	0.13	3.9	0.49
li	r1	1.6	0.05	1.5	0.19
	r2	7.1	0.22	5.3	0.66
sc	r1	4.4	0.14	3.6	0.45
m88ksim	r1	46.9	1.47	30.3	3.79
ijpeg	r1	139.6	4.36	45.3	5.66
perl	r1	11.0	0.34	7.7	0.96
go	r1	8.6	0.27	7.5	0.94
buk	r1	4.0	0.13	4.0	0.50
	r2	4.0	0.13	3.0	0.38

4.3.1 Storage Required

A key question that is addressed in this section is whether the cache has sufficient capacity to hold all of the cache lines accessed by a typical epoch. As we see in Table 5, all of the regions in our experiments require less than 6KB of buffering on average. The worst case is `ijpeg.r1`, which also has by far the largest epoch size (9406 instructions). With 32B cache lines, all twelve of the other regions require less than 1.5KB of buffering. Clearly a relatively small, fully-associative cache would suffice.

4.3.2 Associativity Required to Avoid Replacement

An important question is whether mapping conflicts within a realistic primary data cache with limited associativity would pose a serious problem. To answer this question, each region was simulated

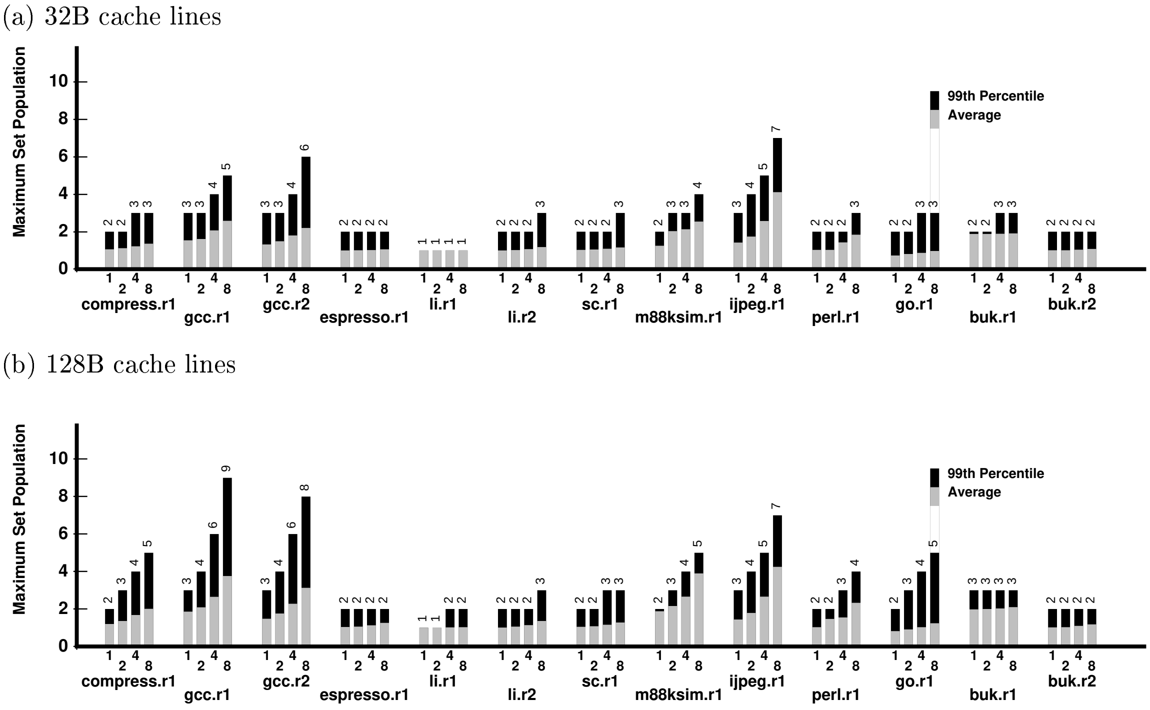


Figure 14: Maximum set population per epoch in a 16KB cache (numbers below the bars indicate the associativity).

using a 16KB primary data cache with 32B and 128B cache lines. We measured the maximum number of lines accessed which map to each *set* of the cache with varying associativities. If the maximum number of lines per set exceeds the number of *ways* in that set, at least one of these lines will be forced out of the cache. Figure 14 shows the maximum set population of any set within the cache, both for the average epoch and for the 99th percentile case. As we increase the associativity, there are more ways within each set, but there are also fewer sets overall, so the maximum set population often increases. Our goal is to find the smallest associativity with enough ways per set to capture these set populations.

As we see in Figure 14, a direct-mapped cache does not appear to be sufficient to capture the set populations. A two-way set associative cache is much more successful—with 32B lines, the 99th percentile case is two or less for nine of thirteen regions, and the average is almost always less than two. With four-way and eight-way associativities, the average set populations are captured for all regions. For eight-way associativity, even the 99th percentile case is always less than eight for all regions. However, associativities of four and eight may increase cache access time significantly, hindering the performance of applications which do not exploit TLDS. Therefore, two-way associativity is perhaps the most attractive option.

4.3.3 Adding A Victim Cache

Rather than giving up as soon as a speculatively accessed line is forced out of the cache, another possibility is to capture these spilled lines within a small *victim cache* [13]. Figure 15 shows the maximum number of victim entries necessary to capture all speculatively loaded or modified lines that would be ejected from a 16KB cache of various associativities. For the direct-mapped strategy, a large victim cache would be needed to capture the 99th percentile case, which may increase overall cache access time. A four-entry victim cache combined with a two-way set-associative cache would

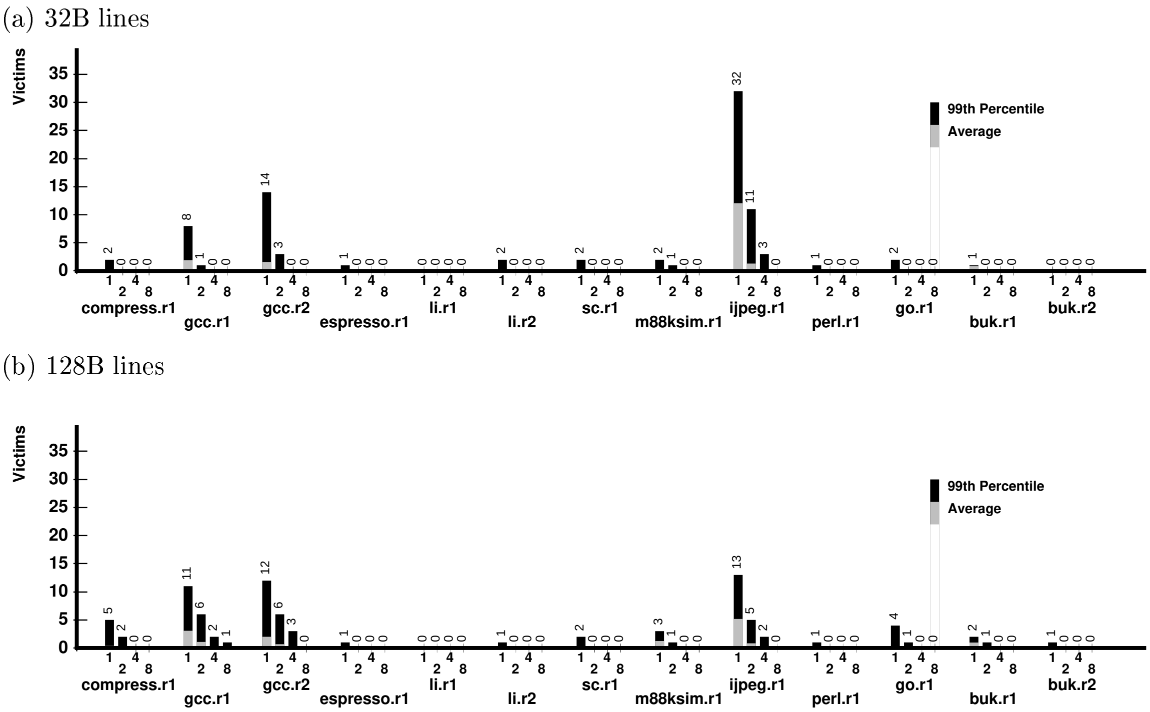


Figure 15: Average victims per epoch in a 16KB cache (numbers below the bars indicate the associativity).

capture the 99th percentile case for all regions but `ijpeg.r1` for 32-byte cache lines, and all but three regions for 128-byte cache lines.

4.4 Summary

We have seen that it is feasible to support TLDS through modest hardware support 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. A 16KB, two-way set associative cache used in conjunction with a four-entry victim cache would be sufficient for buffering speculative state to simplify recovery.

5 Compiler Support for Thread-Level Data Speculation

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. Some of these compiler issues are briefly discussed in this section.

5.1 Finding Speculative Regions

The first step in compiling for TLDS is choosing the appropriate speculative regions to parallelize. We performed this step by hand in our experiments as follows: we used profiling feedback to identify where the program was spending its time, and we then attempted to find the largest surrounding regions which did not have obvious data dependences that would prevent TLDS from working. The compiler could also use feedback information to focus on the important regions of the code, and could statically analyze data dependences where possible. By performing a cost-benefit analysis, it could choose the most promising regions to speculatively parallelize. Once the program is actually running, the rate at which unsafe speculation occurs could be measured and fed back into the compiler again to further refine its choice of speculative regions.

5.2 TLDS Transformations and Optimizations

Once speculative regions have been chosen, the compiler should perform the optimizations described earlier in Section 3.1 (and Table 3) to eliminate memory dependences. We will now briefly describe how these optimizations would work.

Dependences due to induction variables [29] may be eliminated, given that there is a mapping between each *epoch number* (described in Section 4.1) and the value of the induction variable for that epoch. This is the case if epoch numbers are consecutive integers, and they are somehow visible to software. The hardware epoch numbers may be made visible to software, or software may maintain separate epoch numbers.

Certain library routines may cause RAW dependences between epochs, such as input and output (I/O) routines like `getchar()` and `putchar()`. However, it may be possible for the compiler to replace calls to these routines with calls to new parallel versions, thus eliminating the data dependences between the calls. I/O routines may be parallelized by using epoch numbers to impose an ordering on input and output characters. Character input would use the epoch number to index the input buffer, and return the appropriate character. The output buffer for character output can be extended to store an epoch number with each output character. Calls to the character output routine may then be made out-of-order as long as the characters are sorted by the epoch numbers when the buffer is flushed.

Memory allocation library routines may also cause RAW dependences between epochs. These routines would be trivial to parallelize, since each processor could maintain a free list of its own portion of available shared memory.

Reductions [29] are another source of frequent RAW data dependences between epochs, and must be optimized to fully exploit TLDS. Since a reduction applies an associative operation to a variable, the order in which the operation is applied by different epochs does not matter. RAW dependences due to the reduction may then be eliminated by giving each thread a local copy of the variable and combining the local copies at the end of the speculative region. Another possibility would be to maintain one central copy of the variable in memory, and for each epoch to operate on it atomically or through some form of *lock* mechanism.

Finally, the compiler may optimize scalar variables. An important characteristic of a scalar is that accesses to it are not ambiguous, since it is always referred to by its exact location in memory.[12] The value of a given scalar may therefore be forwarded between epochs, if the scalar is the cause of frequent RAW dependence violations. It is important to note that scalar values do not have to be forwarded to preserve program correctness, but the performance of TLDS may be improved by doing so. Forwarding a scalar value involves the insertion of synchronization whenever the scalar is accessed, and arranging the communication of the value through shared memory or some faster means such as a shared register file. The compiler should also aggressively schedule the code to minimize any non-overlapped portions within epochs.

Once data dependences have been optimized, the compiler must then insert calls to the TLDS primitives and create recovery code, such as the example in Figure 11(b) illustrates. The compiler can potentially reduce overheads by minimizing the amount of recovery code—e.g., rather than re-executing the entire epoch, only re-execute the portion that depends on speculative loads. The compiler may also optimize regions which have a small number of dynamic instructions per epoch by combining consecutive epochs. This would decrease the relative costs of TLDS overheads and would give the compiler more instructions to work with when scheduling the code.

[12] This may be a static memory address or a location relative to the current position of the stack pointer.

5.3 Summary

TLDS allows the compiler to focus on parallelism as a performance tradeoff rather than something which is likely to break program correctness. Although compiling and optimizing for TLDS is still a non-trivial task, we believe that for non-numeric codes it is much more feasible than attempting to statically prove 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 17% 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 trade-offs 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 inter-processor communication offers 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. Based on the encouraging results in this study, we advocate that future single-chip multiprocessors provide the modest support necessary for TLDS.

6.1 Future Work

We have described possible architectural and compiler support for TLDS. Future research efforts should be made to fully support TLDS in a compiler, and to devise a working architectural implementation. For the compiler, this will involve automating the process of finding speculative regions and implementing the optimizations described in Section 5.2. In addition, the possibility of using TLDS to parallelize new types of speculative regions such as recursive calls should be investigated. The architectural implementation outlined in Section 4 should be refined. This will involve defining an exact set of TLDS primitives, and developing a precise cache coherence scheme for detecting RAW dependence violations.

We have explored the potential for TLDS in non-numeric codes, since they have previously been difficult to parallelize. Knowing that TLDS will be an effective way to parallelize non-numeric codes, it would also be interesting to measure the impact of TLDS on the simpler case of *numeric* codes. The complexity of the array subscript expressions in some loops of numeric codes prevents parallelization using known methods, while TLDS should prove to be effective.

A Appendix: Experimental Methodology In Depth

This appendix describes our experimental methodology in greater detail. The methods used to track data dependences and run lengths are given, and the removal of data dependences is described. We then present the equations for calculating speedup limits due to forwarding, communication latency, recovery and run lengths. Finally, we describe how these are used to compute region and program speedups.

A.1 Run Lengths

Our simulator uses a hash table (indexed by data address) to record the number of the epoch which performed each load and store for all memory locations accessed within a speculative region, allowing us to track all RAW dependences. Using this information about RAW dependences and the rules given in Section 2.2.1, we may compute the run lengths for a given speculative region.[13]

As discussed in Section 3.1 and Section 5.2, the compiler may remove data dependences due to induction variables, certain library routines, and reductions. Scalar variables may also be optimized such that they do not limit the run lengths. We removed these data dependences in our simulations as follows. First, we used the simulator to profile each speculative region, which gave us the *program counter* (PC) pairs for all cross-epoch RAW dependences. We then used the output of the disassembler (which includes source line numbers) to find the corresponding computation in the source code. For each PC pair which suffered frequent RAW dependence violations, we examined the corresponding source code to decide whether this dependence could be eliminated by the compiler. If so, then this PC pair was included in a list of PC pairs which was explicitly passed to the simulator, such that it would ignore any data dependences at runtime caused by a PC pair in the list.

A.2 Critical Path Lengths

Scalar values and registers may be forwarded between epochs as described in Section 2.3. To compute the critical path lengths, we specify the PCs of all loads and stores of each forwarded variable to the simulator, and then measure the critical paths for different synchronization schemes:

coarse-grain synchronization: The critical path length for each epoch e in a speculative region r for this method of synchronization ($CP_{r,e}^{coarse\_grain}$) is the number of cycles between the first load of any of the forwarded values in the epoch and the last store to any of the forwarded values in the epoch (see Figure 5(a)).

fine-grain synchronization: For this synchronization scheme, we first compute the critical path length for each forwarded variable v, which is the number of cycles between the first load of the variable and the last store of the variable. The critical path length for the epoch e in a speculative region r is then given by

$$CP_{r,e}^{fine\_grain} = \max_{\forall v}(CP_{r,e,v}^{fine\_grain}), \tag{1}$$

where we find the maximum of the critical path lengths for all forwarded variables in that epoch (as illustrated by Figure 5(b)).

fine-grain synchronization with aggressive instruction scheduling: To estimate an aggressive instruction schedule, we first find the number of dynamic instructions in the dependence chain for each forwarded variable.[14] For each forwarded variable v in an epoch e and speculative region r, the critical path length is the length of the dependence chain between the first load and the last store to that variable in the epoch ($CP_{r,e,v}^{fine\_grain\_scheduled}$). The critical path for each epoch e and speculative region r with fine-grain synchronization and aggressive instruction scheduling is therefore

$$CP_{r,e}^{fine\_grain\_scheduled} = \max_{\forall v}(CP_{r,e,v}^{fine\_grain\_scheduled}), \tag{2}$$

the maximum of the critical path lengths for all forwarded variables in that epoch.

The critical path length for a speculative region r and synchronization/scheduling scheme S may then be computed from the sum of the critical path lengths for all epochs in that region:

$$CP_r^S = \sum_{\forall e} CP_{r,e}^S. \tag{3}$$

[13] Note that the threshold value T used for all experiments was 10.

[14] A dynamic instruction is part of the dependence chain for a given value if the instruction uses that value, or if it uses the result of a dynamic instruction already in the dependence chain for that value.

A.3 Speedup Limits

Having demonstrated how the critical path lengths are computed, we now discuss how the speedup limits due to forwarding, communication latency, recovery and run lengths are computed. The number of cycles spent communicating for a region r is given by

$$Comm_r = N_r^{epochs} \times T^{comm}, \tag{4}$$

where the number of epochs in the region (N_r^{epochs}) is multiplied by the number of cycles to communicate between two epochs (T^{comm}). This assumes that there is communication between all consecutive epochs.

We may also estimate the number of cycles to recover from failed speculation for a speculative region r:

$$Recover_r = \frac{T_r - CP_r^S}{N_r^{epochs} \times 2} \times N_r^{run\_lengths}. \tag{5}$$

First, we find the average amount of non-overlapping computation per epoch. For a given epoch which causes a RAW dependence violation, the initial segment of non-overlapping computation will limit the speedup of the speculative region–we estimate this initial segment to be half of the average amount of non-overlapping computation per epoch. We multiply this amount by the number of run lengths for the speculative region ($N_r^{run\_lengths}$, the number of times recovery must occur) to compute the number of cycles spent recovering.

Now we may calculate the speedup limit due to forwarding, communication latency, and recovery by adding the critical path, the communication cycles and the recovery cycles for the speculative region, and dividing the sum into the total number of cycles for the speculative region:

$$Limit_r^{forward\_comm\_recover} = \frac{T_r}{CP_r^S + Comm_r + Recover_r}. \tag{6}$$

An orthogonal speedup limit may be computed from the run lengths, by mapping the run lengths onto a fixed number of processors P (in our experiments, P was set to four). First, we estimate the average number of cycles per epoch in a given region r by dividing the number of cycles in the speculative region (T_r) by the number of epochs in the speculative region (N_r^{epochs}):

$$T_r^{avg\_epoch} = \frac{T_r}{N_r^{epochs}}. \tag{7}$$

We may estimate the number of cycles required to execute a given run length rl in parallel on P processors given the number of epochs in the run length (N_{rl}^{epochs}) as follows:

$$T_{r,rl} = T_r^{avg\_epoch} \times \texttt{ceiling}(\frac{N_{rl}^{epochs}}{P}). \tag{8}$$

Here, we assume that P epochs will always be executed concurrently until the run length end is reached, or until there are less than P epochs left in the run length. We also assume that each new run length begins on P free processors.

Using the number of cycles to execute each run length in a speculative region as computed by equation (8), we may estimate the limit to speedup due to the run lengths for that speculative region:

$$Limit_r^{run\_length} = \frac{T_r}{\sum_{\forall rl} T_{r,rl}}. \tag{9}$$

A.4 Speedups

Speedup limits due to forwarding, communication latency, recovery and run lengths may be combined to calculate a realistic speedup for a given speculative region. This is computed by finding the minimum of the limit to speedup due to forwarding, communication latency and recovery, and the limit to speedup due to the run lengths:

$$Speedup_r = \min(Limit_r^{forward\_comm\_recovery}, Limit_r^{run\_length}). \tag{10}$$

Using equation (10), and given the percentage of total execution time of each speculative region, we may estimate the number of cycles required to execute all speculative regions in parallel with

$$T_{parallel} = T_{original\_program} \times \sum_{\forall r} \frac{\%Exec_r}{Speedup_r \times 100}. \tag{11}$$

Given that

$$T_{with\_TLDS} = T_{parallel} + T_{sequential}, \tag{12}$$

where $T_{sequential}$ is the number of cycles to execute the sequential portion of the application. We may then use Amdahl's law [11] to compute program speedup:

$$Seedup_{program\_with\_TLDS} = \frac{T_{original\_program}}{T_{parallel} + T_{sequential}}. \tag{13}$$

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[2] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, October 1996.

[3] Standard Performance Evaluation Corporation. The spec95int benchmark suite. Technical report. http://www.specbench.org.

[4] K. M. Dixit. New cpu benchmark suites from spec. In *COMPCON*, Spring 1992.

[5] M. Farrens, G. Tyson, and A. R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 338–347, April 1994.

[6] 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 the 28th Annual International Symposium on Microrchitecture*, December 1995.

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

[8] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5), May 1996.

[9] 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 the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 183–195, October 1994.

[10] G. Goff, K. Kennedy, and C. W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[11] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[12] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 200–210, April 1994.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[14] S. W. Keckler and W. J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[15] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[16] D. E. Maydan. *Accurate Analysis of Array References.* PhD thesis, Stanford University, September 1992.

[17] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[18] A. Nicolau. Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38:663–678, May 1989.

[19] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, May 1989.

[20] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th Annual International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[21] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, August 1992.

[22] L. Rauchwerger and D. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[23] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[24] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

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

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

[27] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[28] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[29] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT Press, Cambridge, Massachusetts, 1989.