Journal of Instruction Level Parallelism 3 (2001)

An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors

Peter Rundberg and Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*biff,pers*}*@ce.chalmers.se*

January 16, 2002

Abstract

We present a software approach to design a thread-level data dependence speculation system targeting multiprocessors. Highly-tuned checking codes are associated with loads and stores whose addresses cannot be disambiguated by parallel compilers and that can potentially cause dependence violations at run-time. Besides resolving many name and true data dependencies through dynamic renaming and forwarding, respectively, our method supports parallel commit operations.

Performance results collected on an architectural simulator and validated on a commercial multiprocessor show that the overhead can be reduced to less than ten instructions per speculative memory operation. Moreover, we demonstrate that a ten-fold speedup is possible on some of the difficult-to-parallelize loops in the Perfect Club benchmark suite on a 16-way multiprocessor.

Keywords: Multiprocessors, thread-level speculation, parallel compilers, performance evaluation.

1 Introduction

Thread-level data dependence speculation (TLDS) systems can potentially widen the scope of single-threaded applications that can be automatically parallelized by resolving data dependences at run-time that otherwise would prevent parallelism from being extracted.

Many hardware-centric TLDS systems have been proposed recently targeting small-scale chip multiprocessors [9, 10, 14, 18] as well as large-scale systems [4, 15, 19, 20, 22]. They typically use the cache coherence infrastructure as a mechanism to detect and resolve data dependences taking the form either as name (or anti and output) or true (or flow) data dependences through renaming and forwarding, respectively. While noticeable speedups have been reported for some applications, the complexity added to the detection mechanisms is far from negligible.

Software-based approaches, on the other hand, perform dependence resolution by a combination of static and dynamic techniques and remove the complexity from the memory system design. In the LRPD test by Rauchwerger and Padua [16], name dependence resolution relies on the compiler's ability to statically privatize data. Other dependences will be detected at run-time and will cause the speculation to fail which unavoidably limit the amount of parallelism that can be extracted. Another limitation is that dependence violations are detected after the speculative threads have terminated making misspeculations very costly. In another scheme proposed by Kazi and Lilja [8], name dependences and some true data dependences are resolved at run-time. There are however two kinds of overhead associated with their scheme. First, name dependences are resolved by a dynamic renaming scheme implying the need for restoring system state at the end of the speculative phase. In their scheme, restoration must be done in a serial fashion which may cause significant overheads. Second, true data dependences are resolved by letting each thread that defines data broadcast the addresses to other threads using the data. Forwarding is then carried out by enforcing synchronization between threads that define and use the data. This dependence resolution scheme may also impose severe overheads that again limit the gains of TLDS.

This paper contributes with a design of a software-based speculation system that reduces the overhead of the aforementioned schemes. First, name dependences are resolved by dynamically renaming data at run-time. Second, the overhead of restoring system state is greatly reduced by (i) reducing the amount of state to commit and by (ii) supporting parallel implementations of the commit phase. Third, some true data dependence violations are avoided by supporting lazy forwarding without the

need for enforcing synchronizations between a pair of conflicting threads. Finally, true data dependence violations are detected when they happen which can cut the cost of mis-speculations.

The true challenge of our proposal is to arrive at an implementation with low run-time overhead. In our approach, loads and stores whose addresses cannot be disambiguated statically are associated with highly tuned code sequences. Through a number of code optimization tricks, we show that it is possible to cut the instruction overhead of the common-case critical code path to less than ten instructions. We then analyze the performance impact of our speculation system on a set of three difficult-to-parallelize codes from the Perfect Club [2] benchmark suite using architectural simulation which is validated by runs on a commercial multiprocessor. While we find that all of the codes enjoy at least a speedup of three on eight processors, one of them achieved more than a ten-fold speedup on a 16-way multiprocessor.

As for the rest of the paper, we present the overall design of our approach in the next section. Then in Section 3, a detailed analysis of its implementation is provided. Section 4 then applies our scheme to a set of parallel benchmarks and analyzes its impact on performance using simulation as well as validation runs on a Sun Enterprise 4000 multiprocessor. Finally, we put our work in context to work by others in Section 5 before we conclude.

2 The Approach

We first provide an overview of the approach followed by the detailed design of our scheme. Design tradeoffs are finally discussed.

2.1 Overview

The overall framework in which our speculation system fits in consists of a state of the art parallelizing compiler and a general multiprocessor platform. The task of the compiler is to identify threads that are likely to be data independent and thus might successfully execute speculatively in parallel. For each load and store whose addresses cannot be disambiguated statically, the compiler does three tasks: (1) All data structures that can be touched by such *speculative* loads and stores are associated with a *support data structure* to aid in dependence resolution and violation detection. (2) Each speculative load and store instruction is augmented with checking code that aided by the support data structures resolves or detects data dependence violations dynamically, finally, (3) it assigns a number to each speculatively executed thread that reflects the execution order according to sequential semantics.

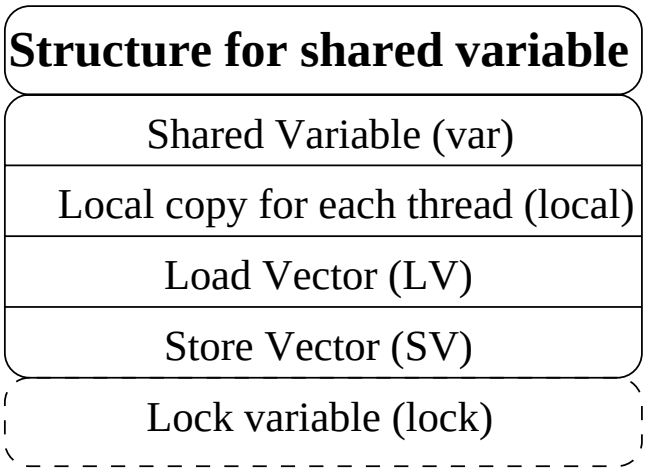

Figure 1: The support data structure associated with a shared variable that may cause data dependence violations.

The overall dependence resolution strategy is to resolve as many name dependences as possible by letting each thread write into a local copy of the shared data structure. If a thread reads from a variable that was most recently modified by a thread with a lower thread number, the data is forwarded from this thread's local copy. On the other hand, if a thread writes to a variable that has been previously read by a thread with a higher thread number with no redefinition in between, a data dependence violation is detected. Finally, if a thread writes to a variable that has been read or written by a thread with a lower thread number such anti and output dependences are resolved through the dynamic renaming scheme.

Let's now consider in detail how the speculation system resolves and detects violations to data dependences given this overall strategy and the above support mechanisms.

2.2 Speculation System Design

While our scheme is general, we will focus on its application to loops as loop-level parallelism is what we target in our evaluation.

Support data structure

With each shared variable (V) that potentially can be touched by a speculative load or store, the compiler associates a data structure containing a *Load Vector (V.LV)*, a *Store Vector (V.SV)*, henceforth called *operation vectors*, and a shadow location (V.local) for each speculatively executed thread to store local

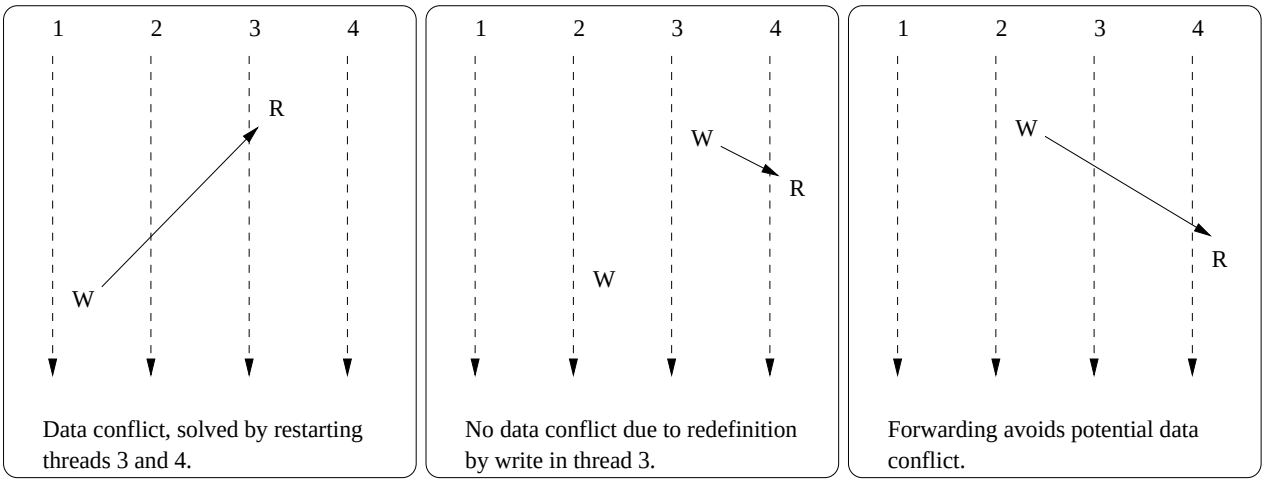

Figure 2: Situations that can occur for loads and stores to shared data across several threads.

modifications, as shown in Figure 1. These load and store vectors keep track of which speculative thread has read from or written to the variable. In the most memory efficient scheme we pursue, we also associate a lock (V.lock) with each variable to avoid race conditions as a result of concurrent read-modify-write operations applied to the operations vectors. The bit vectors are initialized to zero before speculation starts.

Checking Code for Speculative Loads

Figure 2 shows the scenarios that can occur and what happens in the different cases. It is the responsibility of the checking code to discover these cases and take the right action in order to resolve dependences. Pseudo-code for the actions on a speculative load operation is detailed in Table 1(A). In the pseudo-code, `Tnum` designates the thread number.

Besides marking that the thread has read from the variable (line 2) by setting the corresponding bit in LV, a check is done to see whether a thread with the same or a lower thread number has modified the contents of the variable (line 3). Then, the read is satisfied through *forwarding* from the local copy belonging to the thread that performed the most recent write according to sequential semantics, i.e., the thread with the *highest* number less or equal to the current thread number. Note that if only a thread of a higher number had written, the load is satisfied from the memory copy and no special action need to be taken because this anti-data dependency is resolved through the dynamic renaming scheme. However, if the thread had written the data itself, it reads from its own local copy, where it has stored

```
funct Speculative_Load(Spec_Var V) returns Value
{
1:  lock(V.lock);
2:  V.LV[Tnum]=1;
3:  if (V.SV & ME_OR_LOWER_MASK)
4:      tmp=find_local(V.SV);
5:  else tmp=V.var;
6:  unlock(V.lock);
7:  return tmp;
}
```

(A)

```
procedure Speculative_Store(Spec_Var V, Value Val)
{
1:  lock(V.lock);
2:  V.SV[Tnum]=1;
3:  V.local[Tnum]=Val;
4:  if (V.LV & HIGHER_MASK)
5:      if (!(V.SV & BETWEEN_MASK))
6:          rollback;
7:  unlock(V.lock);
}
```

(B)

Table 1: Pseudo-code for speculative loads and stores

the modification it made earlier.

Checking Code for Speculative Stores

The pseudo-code for a speculative store operation is displayed in Table 1(B). The first actions are to update the store vector by setting the bit corresponding to the thread number (line 2) and to update the local shadow copy of the variable (line 3). The next action aims at detecting whether a flow data dependency has occurred. This test is implemented as follows. We first check LV to see if a thread (denote it $\mathtt{T}_{\mathtt{read}}$) with a higher thread number than `Tnum` (`HIGHER_MASK` on line 4) has read the variable. If this is the case, the next test ascertains whether $\mathtt{T}_{\mathtt{read}}$ should have read the value produced by the current store. This is the case if no thread with a higher number than `Tnum` and lower number than $\mathtt{T}_{\mathtt{read}}$ has written to the variable (`BETWEEN_MASK` on line 5); then there might be a flow data dependence violation. Note that we also flag a violation even if $\mathtt{T}_{\mathtt{read}}$ re-defined the variable before it was read. While it is possible to avoid this, it turns out to simplify the checking code significantly to allow for false violations in this case.

A violation can not be resolved in any other way than by stopping the threads with the same or higher thread numbers than the thread that erroneously read and re-executing all of them. The specific actions to be taken then are discussed later and are referred to as a *rollback* (line 6). Discovering the dependence conflict as soon as it happens is an important feature of our method that allows us to minimize the time wasted on a data dependence mis-speculation.

Locking

The inline code modifying the operation vectors needs to be protected by locks preventing other threads from modifying the operation vectors of a variable while it is being modified. This is essential for correct

execution, but it also imposes an overhead that we would like to be without; the lock overhead. We will see in Section 3.1 that this overhead is quite significant, and we will also show how we can essentially eliminate this overhead by implementing a lock free structure in same cases.

Rollback

When a flow data dependence violation has occurred, we need to restart the thread that caused the data dependence violation, i.e., the thread that read data too early, to ensure correct execution. Conservatively, all threads with higher thread numbers than this thread are also restarted, although only those threads that have a flow data dependency with this thread must be restarted. The particular recovery actions on a roll-back involve resetting the bits set in all LV and SV touched. This can indeed be very costly and, therefore, speculation only pays off if it frequently succeeds. However, there is no cost involved in restoring system state since all modifications have been made to local copies.

There are different options for how to restart the execution. One unique feature of our method is the fact that we can identify which threads need to be restarted from the information gathered in LV and SV. This opens up several alternatives for how to restart mis-speculated threads after a violation is detected. For example, it is possible to restart only those threads that violated a flow data dependence relation and continue the parallel execution. On the other extreme, it is possible to choose a more conservative action to restart all the threads serially as done in other proposed software schemes [16].

Commit

A speculative store writes all data to a shadow copy of each variable unique to each thread to provide renaming which can resolve name dependencies. After the speculative execution, each thread needs to write back the contents of the shadow copies to the corresponding locations in memory. This is called the *commit* phase. The threads need to commit in the thread number order to preserve sequential semantics of the original execution.

A conservative approach is to commit all elements modified by each thread one-by-one across all threads. This is called *serial commit* and is potentially costly. By committing each value produced by a thread in an order consistent with the serial execution, we get the correct final state. While threads with a lower number might have their results over-written by threads with a higher number, the final state is always consistent with the serial execution because of the serial nature of the commit operation.

A useful feature of our scheme is that the thread that wrote the last value to a variable in the serial

order can easily be derived from the store vectors. In addition, different data elements can commit in any arbitrary order enabling parallel commit implementations. In such a *parallel commit*, the store vector of each variable is inspected to see which thread was the latest to modify it. Even if several threads have modified the variable, only the latest thread in serial order needs to have its value committed since this is the only value that will be seen after the parallel execution. This will greatly reduce the state that needs to be committed. The commit of the variables can be performed in any order since at most one value is going to be written for each variable. By splitting the data that is to be committed among all processors we can have each processor committing a part of the speculative state without regard for the other processors. Ideally we can achieve near perfect speedups for the parallel commit.

We will consider the performance impact of both options and will see that parallel commit operations provide an advantage when we scale the implementation to many processors.

2.3 Design Considerations

In order to arrive at a low-overhead implementation, our baseline design has been influenced by some key observations that we discuss in this section. In addition, we also discuss a number of design tradeoffs that are important to address for alternative designs.

The baseline design potentially suffers from quite significant memory overheads. However, note that support data structures only need to be associated with data structures prone to data dependence violations. In addition, there are many options for trading efficiency for lower memory overhead. One option would be to maintain operation vectors on coarser chunks of data than single data elements. Another option would be to cut down on the speculative state by dynamically allocating copies first when a thread modifies data. Both these options would result in either more false dependence violations or more complex checking code. While they may be interesting alternatives, we will not evaluate their tradeoffs in this paper.

The fact that operation vectors and shadow copies are allocated close to each other and to the original variable makes it very efficient to find the local copies and operation vectors, as they are on a fixed offset from the original variable. As noted before, the loops we are interested in are likely to have few dependences. It is therefore likely that accesses to LV and SV – common case operations in the pseudo code – cause few cache misses, because of the high temporal re-use and since the operation vectors are most likely allocated in the same cache block as the original variable. This is a well-known

technique for increasing locality called *merging arrays* [7] where data that is to be accessed together is also allocated together.

Using *merging arrays* may however have detrimental effects on cache performance as well. First, spatial locality for an array of speculative variables will be reduced because of the added distance between consecutive array variables caused by the support data structure. Secondly, each speculative variable being extended with the support data structure extends the footprint and thus requires more space than the original variable in the unmodified program. Both these effects may cause higher miss ratios which may impose higher execution overheads. In Section 4.3, we will show how much overhead this may cause.

An alternative data layout strategy would be to associate with each array of variables separate arrays for each variable in the support data structure. We refer to this data layout strategy as *non-merged*. One notes that this technique will keep the spatial locality high. In addition, it will also reduce the footprint as only the support data structures that are needed will be brought into the cache. As a result, this layout strategy is not expected to lead to as high a miss ratio. However, the instruction overhead associated with accessing the variables in the support data structures is higher. While we will use the merging arrays strategy as default in the rest of the paper, we will present measurements done with both non-merged and merged support data structures in Section 4.3 to highlight their tradeoffs.

3 Implementation

In this section we analyze the overhead of an implementation of the checking code for the baseline implementation in the previous section. We first consider a straightforward assembly code implementation in Section 3.1 and then particularly address the bottlenecks caused by the locks protecting the operation vectors in Section 3.2.

3.1 Checking Code Overhead

The instruction overhead of the checking codes is dictated by their common case paths. In a highly optimized assembly code implementation, we have counted the number of instructions that need to be executed in each path. These statistics are shown in in Table 2. In the left column we list the paths in the pseudo-code of Table 1 that correspond to well-defined actions. For example, the (presumed) common case for a load is when the data can be read from the memory and does not involve forwarding

Case	Instr.	Lines in Pseudo Code
Common Case (load)	12	A: 1, 2, 3, 5, 6, 7
Common Case (store)	11	B: 1, 2, 3, 4, 7
Forward Local	17	A: 1, 2, 3, 4, 6, 7
Forward Remote	≈ 60	A: 1, 2, 3, 4, 6, 7
Rollback	>1000	B: 1, 2, 3, 4, 5, 6
Commit (serial)	10	Not Shown
Commit (parallel)	25	Not Shown

Table 2: Instruction count for paths in the checking code.

at all. In the third column, we list the numbers corresponding to the statements in the pseudo-code that form this path. The common case for a speculative store operation is to update the store vector and the local shadow copy. Finally, the second column lists the instruction count of each path.

A first observation is that the (presumed) common case actions for speculative load and store operations are carried out by as few as 12 and 11 instructions, respectively. This is indeed encouraging given that only loads and stores that potentially can be involved in a flow data dependence violation need to take this overhead. A second observation is that forwarding is indeed inefficient in that the overhead increases to 17 and about 60 instructions for forwarding from the local or a remote shadow copy, respectively.

Forward Local is when a thread reads a variable modified by itself. The Forward Remote part is when a thread reads an element modified by a thread with a lower thread number to eliminate some flow data dependences. The reason why it is only an approximate value is because the number of instructions depends on which thread the value is read from. We have implemented it by a loop that scans the store operation vector (SV) to find the thread with the highest number lower than the current thread that has modified the variable.

Committing the modified variables back to their original location after the speculative execution can either be done in a serial fashion where threads perform all commitment work one-by-one. Or, it can be performed in parallel, with each thread committing a fraction of all the modified variables. The instruction counts for commit correspond to committing a single variable. Thus, this count has to be multiplied with the number of variables to commit.

Finally, rollback involves zeroing all bits in the operation vectors of variables it has read or written to and restart execution speculatively or serially if it is likely that violations will be frequent. The

overhead is enormous and speculation therefore only makes sense if it frequently succeeds.

Going back to the presumed common-case critical path, an area of improvement would be to do it without the locking. In fact, we have found that locking accounts for as many as five instructions. We will next study how the overhead can be further decreased by an enhanced scheme that avoids expensive locking. This also brings down the memory overhead because a lock variable then does not have to be allocated.

3.2 Optimization of Locking Overhead

Modification of the operation vectors must be done atomically within a critical section. The locks needed result in a quite significant overhead considering the highly efficient common-case path in the checking code itself.

An alternative is to implement the operation vectors as byte vectors instead, i.e. each thread is represented by a byte instead of a bit. It is then possible to perform the updates via atomic store byte instructions. This eliminates the need for the explicit lock to preserve atomicity because of operation vector updates. At first glance it seems that a synchronization for the checking code of the speculative store is still needed because modifying the shadow copy and the operation code must be done atomically. The reason is that an obsolete value might be forwarded otherwise. We need to ascertain that the local write is performed before the forwarding is allowed. This can be achieved by letting the store first write `0x0F` to its byte in the SV, then performing the update of the local copy, and after this write `0xFF` to the SV. A load discovering that the thread it needs a forward from has `0x0F` written in its SV just needs to wait for it to change to `0xFF` before it performs the forwarded load operation. These changes remove the need for explicit locks, and cuts the number of instructions needed for the checking code significantly.

A drawback of this implementation is that it will not scale well beyond 4 processors on a 32-bit machine or 8 processors on a 64-bit machine because checking the operation vector will then be split into several phases resulting in a higher instruction overhead. It will also require a large amount of memory to be reserved for just the operation vector since each bit is really a byte. Thus, to scale this implementation to large numbers of processors may be less attractive from a memory space overhead point of view. Nevertheless, it results in that the common-case critical-path for speculative loads and stores can be cut to 7 and 9 instructions, respectively, as opposed to 12 and 11, while the remote

```
Situations:
Load:
    tmp=array[i];
Store:
    array[i]=tmp;

r1=tmp
r2=&array
r3=i

Instructions in
italic are
needed even without
speculation.
```

```
load:
1:  sll r3, STRUCT_SIZE, r4
2:  add r2, r4, r4
3:  mov 255, r5
4:  stb r5, [r4 + 4 + THREAD_NUMBER]
5:  ld [r4 + 8], r6
6:  andcc ME_OR_LOWER_MASK, r6, r0
7:  bg FORWARD_HANDLER
8:  ld [r4 + 0], r1
9:  ba END_OF_CHECKCODE
-----------------------------------


Overhead:  7 instructions.
```

(A)

```
store:
 1:  sll r3, STRUCT_SIZE, r4
 2:  add r2, r4, r4
 3:  mov 15, r5
 4:  stb r5, [r4 + 8 + THREAD_NUMBER]
 5:  st r1, [r4 + 12 + THREAD_NUMBER*4]
 6:  mov 255, r5
 7:  stb r5, [r4 + 8 + THREAD_NUMBER]
 8:  ld [r4 + 4], r6
 9:  andcc HIGHER_MASK, r6, r0
10:  bg ROLLBACK_HANDLER
11:  ba END_OF_CHECKCODE
----------------------------------------
Overhead:  9 instructions.
```

(B)

Table 3: Pseudo SPARC-assembly code for common case path of speculative loads (A) and stores (B).

forward will be slightly longer. While the checking code for stores is not reduced by much, we will see in Section 4.2 that it is in fact the loads that contribute to most of the overhead, and thus benefit most from the code reduction. The dynamic assembly code sequences of the presumed common-case code paths using this optimization are shown in Table 3.

4 Performance Evaluation

We now present preliminary results of the performance impact of our speculation system on some difficult-to-parallelize loops from the Perfect Benchmark Suite run on a simulated symmetric multiprocessor model. In Section 4.1, we first present the methodology including the benchmarks used followed by the performance results in Section 4.2. Then, in Section 4.3, we validate the results on a commercial multiprocessor – a Sun Microsystems E4000 multiprocessor.

4.1 Evaluation Methodology and Benchmarks

The evaluation is performed with a combination of two tools: an execution-driven instruction-level simulator based on SIMICS [11], and a trace-driven simulation model of a symmetric multiprocessor. SIMICS is an instruction-level simulator of a SPARC V8 ISA that is capable of generating memory traces. In order to generate the trace for the trace-driven simulator, we first annotated each loop so that the memory reference trace produced by SIMICS also marks all speculative loads and stores in addition to the start of each loop iteration. The annotated loop is then run on a single-processor SIMICS SPARC simulator and produces a trace in which the virtual time is associated with each memory

reference.

The trace-driven simulator models a symmetric multiprocessor with private caches. The cache and timing parameters used during the experiments are shown in Table 4. Besides modeling cache performance, the simulator is also responsible of implementing the timing of the speculation system by advancing the virtual time an amount that depends on the instruction overhead of the speculation action according to Table 2.

The annotated memory trace is converted into a parallel trace by simply distributing the total number of iterations across the number of processors under simulation. By having access to the virtual time between two consecutive memory operations, it is possible for the trace-driven simulator to factor in the time between two consecutive memory operations by the same thread.

There are two simplifications in our methodology that needs attention. First, we assume single-issue processors. Under the assumption that the application as well as the checking code will be equally compressed by a multiple-issue processor, our numbers should be indicative for the latter. However, in Section 4.3, we will validate this assumption by measurements on a multiple-issue processor and will show that there is a fair amount of instruction-level parallelism in the checking code itself.

Second, the trace-driven methodology potentially has the shortcoming that race-conditions that appear dynamically cannot affect the trace. However, since speculative execution relies on relaxed memory consistency models, such race-conditions are only expected to affect the trace when there are data dependence violations. For the applications we experiment with, such violations are in fact rare which means that our trace-driven approach is sound.

Parameter	Setting
Cache Size	32 kbyte
Cache Organization	Direct-Mapped
Line Size	64 byte
Cache Hit Time	1 cycle
Cache Miss Penalty	100 cycles

Table 4: Machine parameters used in simulation.

We have evaluated the performance impact of our speculation technique on three hard-to-parallelize loops from the Perfect Club benchmarks [2]. The loops, listed in Table 5, are all chosen because they represent a large fraction of the serial execution time, and they are not statically analyzable [22] by

state-of-the-art parallel compiler frameworks such as Polaris [3]. All benchmarks have been translated from Fortran to C with f2c [5], and been compiled with gcc 2.95.2 with optimization level -O2.

Benchmark	Subroutine: Loop Name	% of Serial
Ocean	FTRVMT: 109	45%
Adm	RUN: 20, 30, 40, 50, 60, 100	35%
Track	NLFILT: 300	52%

Table 5: Loops chosen for evaluation.

The serial execution times of the loops are measured on a Sun Ultra Enterprise 4000, equipped with UltraSPARCII processors at 250 MHz, with the Sun Workshop Compiler tool "looptool". These benchmarks have also been used in other papers [16, 22] where the sequential part of the programs were reported to be slightly different. These differences most certainly come from compiler and system differences between the measurements. All loops contain reductions that were removed by hand prior to the speculative execution. Removing reductions by hand makes sense since uncovering reduction parallelism is orthogonal to data dependence speculation which we target in this paper.

All code is parallelized by hand by manually following the steps outlined in Section 2.1. The validation of the resulting parallel program is done by verifying that the output is the same as a serial version of the program.

The loops we have run do not cause any rollbacks. While this makes them well suited for a speculation system like the one we propose, they nevertheless also stress the issues of resolving name dependences and checking for dependence violations which is in focus of our analysis. If the loops had contained data dependence violations, the result would still be correct but the speedup would be severely limited. To study this is outside the scope of the paper as it is a fundamental issue of all speculation systems regardless whether they are software or hardware-centric.

The loop in *Ocean* consists of 32 iterations most of the 4129 times it is executed. The amount of computation versus the number of instrumented loads and stores is quite small. Because of this, the overhead of the speculation system is expected to be large. The small number of iterations for each invocation of the loop, and the small amount of computation in each iteration, makes it unsuitable for a large number of processors, thus we do not run this benchmark on more than 8 processors. Each iteration performs about an equal amount of work, thus this loop is quite well load balanced, and scales well with the number of processors.

In *Adm,* we have used several loops that all have the same dependence pattern. These loops are executed 900 times with about 32 iterations each time. The amount of computation is is not very large compared to the amount of instrumented loads and stores. The overhead of the speculation system is thus expected to be quite significant for this benchmark. The load balance is good and the performance scales very well with the number of processors.

The loop in *Track* consists of about 480 iterations and is executed 56 times. This loop is not free of dependences, as 5 of the 56 invocations are not fully parallel. Despite this, all 56 invocations of the loop are able to run in parallel when the iterations are split evenly across threads. Since there is much computation compared to the number of instrumented loads and stores, we expect the impact of the speculation system on the execution time to be small. The load in the loop is quite imbalanced, but it still scales quite well with the number of processors.

4.2 Performance Results

The baseline implementation of the speculation system uses locks to preserve atomic updates of operation vectors and implements a serial commit operation. We will first study the impact of this speculation system on the speedup of the three loops, which is done in Section 4.2.1, before we study performance optimizations of parallel commit and removal of locks, which is the theme of Section 4.2.2.

4.2.1 Performance impact of baseline system

The execution time of the loops on the baseline speculation system which uses locks and a serial commit operation are shown in Figure 3 in the left hand diagrams. The execution times are normalized against the serial execution of the loop without any checking code, and are shown for up to 16 processors for *Adm* and *Track* and for up to 8 processors for *Ocean*. The execution times are split into different parts showing the overhead of instrumented loads, stores and commits as well as overhead for cache misses.

Starting with *Ocean* and *Adm*, the overhead of the speculation system is roughly as high as the useful execution time. An explanation for this can be found in Table 6 where we show the amount of checking code that have to be applied to the three loops. In the first row we show the average distance between instrumented memory operations, counted in instructions. *Ocean* encounters an instrumented memory operation every 9 instructions, and *Adm* encounters one every 17 instructions. However, as the second and third rows in Table 6 show, only about 27% of the loads and between 13.7% and 16.8% of the stores are indeed speculative. We have also measured the frequency of different speculation actions

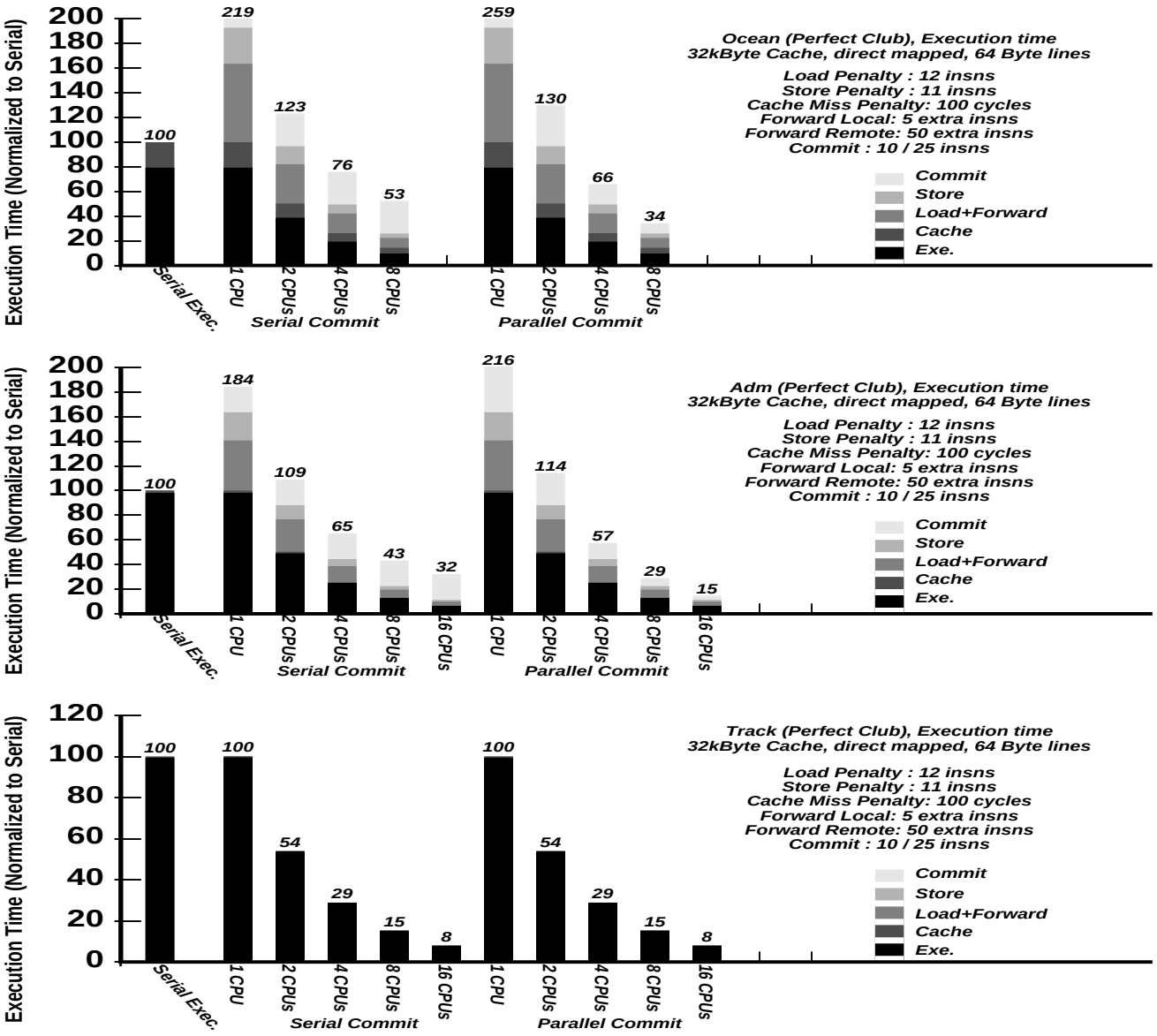

Figure 3: Execution time results for Ocean (top), Adm (middle), and Track (bottom) using locks and with serial commit to the left and parallel commit to the right.

which are shown in Table 6. Another interesting observation is that the common case in the execution path is followed by *Ocean* and *Track*, but not by *Adm*. There are actually quite a large number of local forwards in *Adm*. The fact that no remote forwards appear is probably because all forwards occur within an iteration of the loop. None of these loops requires any rollbacks, and thus executes speculatively with no flow data dependence violations.

Overall, these numbers indicate that the speculation overhead is about 100% for *Ocean* and *Adm*.

Despite these overheads, we start to achieve a speedup at four processors. Indeed, at eight processors we get a speedup of about a factor two.

For *Track*, the results are much more encouraging. From the bottom-most diagram to the left of Figure 3, we see that the overhead is not as large for this loop, in fact it is barely visible. From Table 6, we get the explanation. *Track* only encounters on average a speculative load or store every 5003 instructions. Like *Ocean*, virtually no forwarding is done which means that the common-case code path is almost always invoked upon a speculative memory operation leading to an overhead of 12 and 11 instructions for a speculative load and store, respectively.

Overall, with the baseline speculation system we get speedups ranging from 2.3 (Ocean), 3.1 (Adm) to 12.5 (Track). We next study the improvements gained from some optimizations.

Situation	Ocean	Adm	Track
Instruct. between Instr. Ops	9	17	5003
Instr. Loads / Mem. Op.	27.5 %	27.4 %	0.14 %
Instr. Stores / Mem. Op.	13.7 %	16.8 %	0.03 %
Common-case / Instr. Op.	100 %	55.5 %	99.9 %
Forward Local / Instr. Op.	0 %	44.5 %	0.1 %
Forward Remote / Instr. Op.	0 %	0 %	0 %
Rollbacks / Instr. Op.	0 %	0 %	0 %
Sum	100 %	100 %	100 %

Table 6: Distribution of events in the checking code.

4.2.2 Effects of optimizations

In studying the speculation system overheads of the leftmost diagrams in Figure 3 we see that the overhead of the commit operation start to become dominant as we increase the number of processors. In the baseline implementation, we implemented a serial commit in which all modified words are written back to system memory one-by-one. In the rightmost diagrams, we have evaluated the effect of performing the commit operation in parallel by simply assuming that we can evenly divide the commit task evenly across all available processors. As expected, for *Ocean* and *Adm*, the commit overhead is cut significantly. In fact, the speedups now obtained are 2.9 for *Ocean* on 8 processors and 6.7 for *Adm* on 16 processors. Obviously, *Track* does not benefit much since the overhead is so tiny anyway.

The next optimization we study is to replace the locks with the scheme where operation vectors

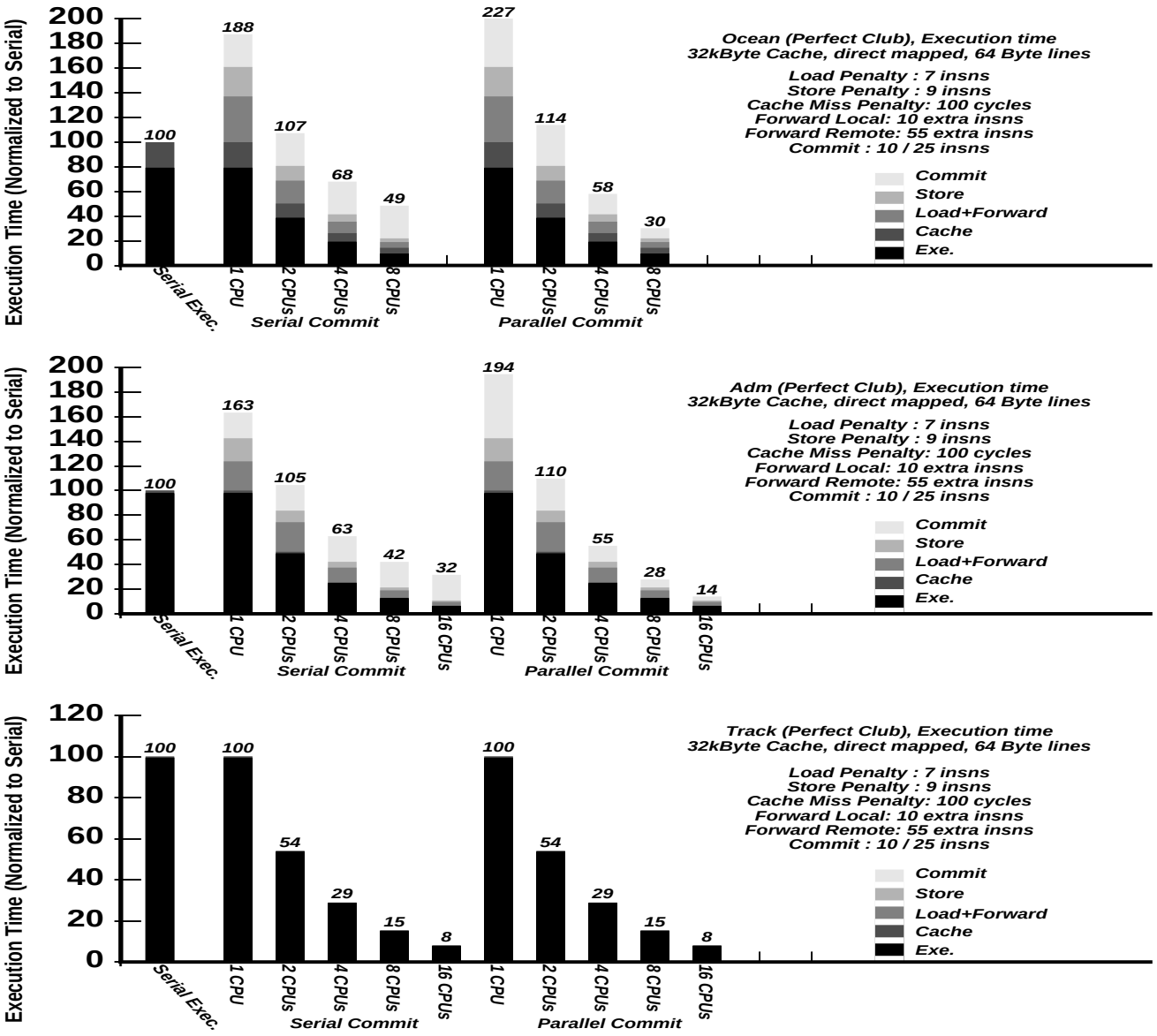

Figure 4: Execution times for Ocean (top), Adm (middle), and Track (bottom) assuming the enhanced locking scheme.

are implemented as byte-vectors instead of bit-vectors as explained in Section 3.2. The corresponding execution time diagrams are shown in Figure 4. We can see that the overhead for loads is reduced by a good amount and that we are close to getting speedup at 2 processors for *Ocean* and *Adm*. With these two applications, we now get a speedup of 3.3 for *Ocean* and 7.1 for *Adm* assuming a parallel commit implementation.

In summary, we have seen that it is possible to get quite impressive speedups of difficult-to-parallelize

codes using a purely software-based thread-level data dependence speculation system. Critical to the efficiency is to optimize the common-case operations. Two optimization tricks we have studied are the removal of expensive locks and the implementation of parallel commit operations. Both these optimizations lead to speedups of 3.3 for *Ocean* on 8 processors and 7.1 and 12.5 for *Adm* and *Track* on 16 processors, respectively.

4.3 Validation

As already pointed out, the simulation model has some limitations. First, it does not factor in the effects the support data structures have on cache performance as discussed in Section 2.3. Second, by modeling a single-issue processor, it does not properly model the overhead on a contemporary multiple-issue processor. In this section, we will validate the results in the previous section with respect to both these limitations.

As a base for our first validation, we have used a micro benchmark which consists of a loop that accesses three array elements in each iteration according to the code below.

```
for (i=1; i<N; i++)
 a[i] = b[i] + c[i];
```

From this loop we can study the importance of proper data layout of the control data structures. Let us assume, for simplicity, that the iterations are spread evenly across threads and that cross-iteration dependences cannot be statically resolved by a compiler. As a result, each iteration is associated with a speculative store and two speculative loads. Assuming no dependences, each speculative load will access an array element and the corresponding load and store vector. Each speculative store will modify the local copy for the thread and the load and the store vector. By investigating the checking code, we note that five memory reads and five memory writes will be performed in each loop iteration. It will thus be highly unlikely that a speedup can be achieved on a loop like this because of excessive instruction overhead. It nevertheless illustrates the need for good locality. With poor locality the extra memory references will impose a dramatic increase in memory stall times, and performance will suffer greatly. Let us study this effect in more detail.

Let's consider three cases. In the first case, we assume that the working set of the application was so small that all data, both the shared variable and the support structures, fit in the L1 cache. Then

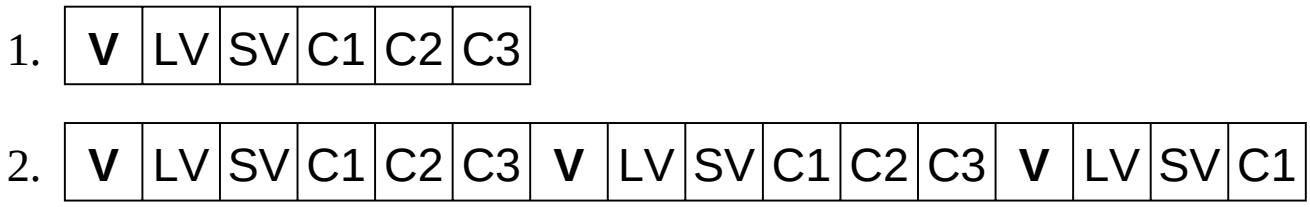

Figure 5: (1) Data layout of a four processor merged variable. (2) a 64 byte cache line filled with the variables.

no cache misses will occur and the overhead of speculation is limited to the instruction overhead. Let's refer to this case as *Small working set.*

In the second case, we assume that data does not fit in the L1 cache, but it fits in the L2 cache. It is then clear that the merged data layout strategy will cause an excessive amount of L1 cache misses. In Figure 5, we show how the data layout of the array elements and the control data structure associated with them assuming the merged data layout. Let us assume a cache block size of 64 bytes, 16 variables of 4 bytes each fit in a cache block. Then there will be 15 hits followed by a miss (hit rate = 94%) when accessing consecutive array elements. In the speculative version of the code, however, we note that only one out of six array element accesses will result in a cache hit (hit rate = 17%). The number of cache misses will increase with increasing numbers of speculative threads because of more local copies in the control data structure. We refer to this case as *Medium working set.*

In the final case, we assume that the working set does not fit into the L2 cache either. Then we will have to pay a huge penalty since each cache miss will need to be satisfied from memory. We refer to this case as *Large working set.*

Working set	Merged	Non-Merged
Small	14	20
Medium	30	30
Large	250	140

Table 7: Running times in clock cycles for different data layouts.

We measured the average number of cycles for each iteration on an AMD Athlon microprocessor, with 64 byte cache lines in both the L1 and the L2 caches. We have run these measurements on an Athlon because we wanted to study the amount of ILP extracted in our checking code by a more aggressive processor than the UltraSPARCII which is the processor used in the E4000 multiprocessor which we will

later use to study parallel speedup. We made sure the caches were warm before measurements started to properly illustrate the behavior with the three working set sizes. The cycle counts for the three cases assuming the merged array are shown in the leftmost column in Table 7.

With the Small working set, we can see in Table 7 that an iteration of the loop takes 14 clock cycles with the merged data layout. These 14 cycles almost completely consist of instruction overhead because of the checking code. Running the same loop without checking code requires 2 cycles per iteration. Thus there seems to be quite good instruction level parallelism in the checking code since a loop iteration consists of about 25 instructions and this imposes only 12 extra cycles.

Moving on to the Medium working set case, the runtime increases to 30 clock cycles. The loop is still only 25 instructions long, but the L1 cache misses cause a degradation in performance. Since the L2 cache is very fast the degradation is not that severe. The L2 hit time is 11 cycles according to the specification from AMD. Since 5 out of 6 array element accesses (83%) cause cache misses the three variable accesses in an iteration should take around 3*11*5/6=28 cycles. In Table 7 we can see that instruction overhead turns out to be 30 cycles which is close to what we expect. The Large working set causes L2 misses instead of L1 misses. The same calculation with an L2 miss penalty of 100 cycles gives us an iteration time of 250 cycles which corresponds well to our measured value (3*100*5/6=250).

Recalling the data layout in the non-merged layout in Section 2.3, shared data and its associated support data structures are not allocated together. This causes extra instruction overheads because we can not simply assume that the support data structures are on a fixed offset from the shared variable. It also causes more intricate access patterns, but the increased spatial locality improves cache performance. Because of this, each loop iteration with the large working set is now run at only 140 cycles per iteration, a huge improvement. As can be seen in Table 7, for the small working set the iterations are slower than with the merged layout. With the medium working set both layouts are tied. This illustrates how important the layout of the support data structures is.

Our second validation concerns the speedup on a real machine. We have used a Sun Microsystems E4000 multiprocessor and instrumented the Ocean application as follows. Ocean was parallelized using optimized locks, parallel commit, and a non-merged data layout, i.e. the same assumptions as used in the simulations except that the cache performance now is factored in. The loop in question was split into threads by hand, and the result of a parallel run was validated against a sequential run of the loop. Our method is of course aimed at having automatic tools performing the parallelization and inserting

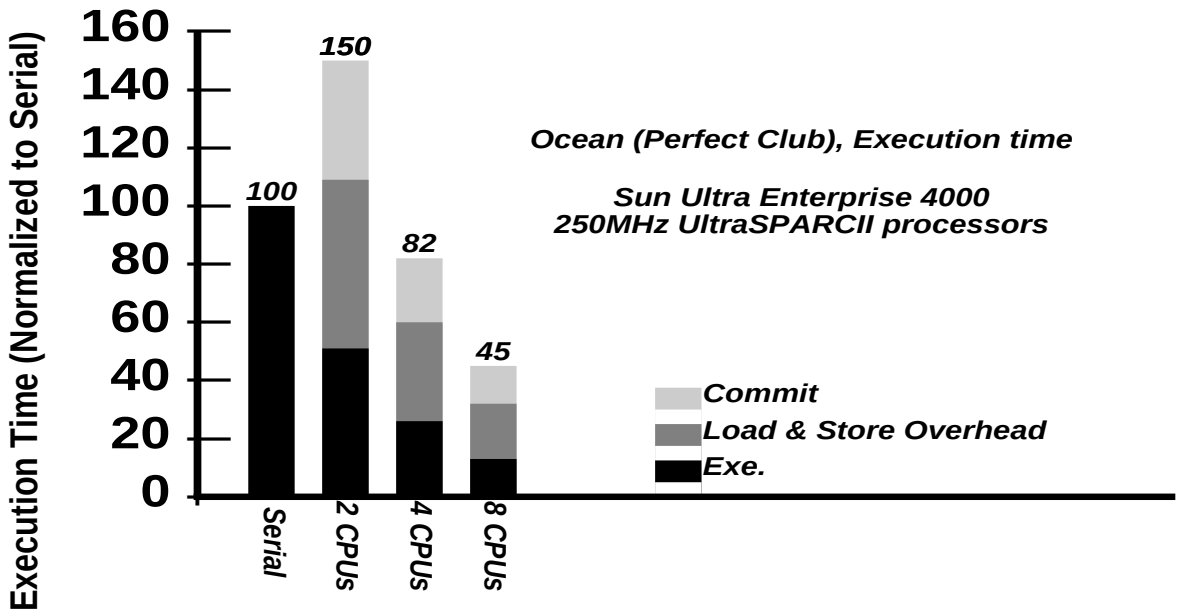

Figure 6: Ocean parallelized using parallel commit and optimized locks and run on a Sun Enterprise 4000.

the necessary checking code. We have not implemented an automatic tool and are thus forced to do the validation through hand parallelization. The corresponding execution times for parallel runs with up to eight processors are shown in Figure 6. The results show that our simulations, with their simplifications, are quite representative of real world performance. A speedup of 2.2 times was achieved on 8 processors, and considering that Ocean is the least favorable of the three benchmarks to our speculation system, this is an encouraging result. We would have expected even closer correspondence for the other benchmarks.

5 Related Work

Previous approaches to design thread-level data dependence speculation system have taken either a predominantly hardware-centric or software-centric approach.

Hardware-centric approaches include multithreaded architectures [1, 12, 13], tightly-coupled chip multiprocessors [9, 10, 14, 18], and distributed shared-memory multiprocessor systems [4, 15, 19, 20, 22]. Our method is targeting a multiprocessor infrastructure and will be compared to other such systems. Proposed hardware-centric speculation systems for multiprocessors have leveraged the same basic infrastructure including private caches kept consistent by a cache coherence protocol [4, 9, 10, 14, 15, 18, 19, 20, 22]. In all these schemes, validation of speculative loads and stores is conceptually simple

because all conflicting accesses (two accesses from different threads to the same location of which at least one is a store) result in cache coherence transactions (invalidations or coherence misses). Like in our approach these systems (1) resolve all name dependencies (2) resolve some flow data dependencies through forwarding, and (3) detect flow dependence violations. Data dependence checking calls for extra complexity in terms of more state information per block, a more complex protocol, and using part of the cache space to keep shadow copies. Exhausting this space causes a performance impact either in terms of stall time for the processor that caused it [14] or by provoking a violation [18].

In comparison with our software-centric approach, hardware-centric schemes potentially allow more concurrency in the actions needed to do validation of memory operations which gives them a potential performance advantage. However, since the overhead in our implementation could be kept at a fairly decent level, it is not obvious that this performance advantage is justified in light of the non-trivial changes to the basic cache-coherence infrastructure.

Software-centric implementations provide flexibility because the speculation strategy can be customized by the compiler using high-level program information. Examples of obvious customizations include providing privatizing as well as non-privatizing algorithms. Software-centric algorithms are also not restricted by system parameters such as the cache size and the block size. In the hardware-centric approaches they may cause performance problems owing to exhausting the space for renaming and the signaling of false dependencies. In summary, there is certainly not a clear-cut case for choosing a purely hardware-centric approach to support thread-level data dependence speculation.

As with our software scheme, most hardware-centric approaches rely on the compiler to split the program into threads for each execution unit. There have been some work done on having the hardware extract the parallelism for multithreaded architectures instead of letting a compiler do this [1, 12, 13].

While software-centric speculation systems have been tried in the past, they suffer from limitations not apparent in our proposal. In the LRPD test proposed by Rauchwerger and Padua [16] the speculative execution of threads consists of two passes. In the first pass, the threads are executed and all speculative loads and stores are marked. In the subsequent validation pass, violations to flow data dependencies are tested and the result of the test is simply pass or fail.

A key difference between their and our approach is that we do the marking and the validation on the fly. In addition, since the marking information they provide cannot be used to identify which load/store pair causes a dependence violation prevent them from implementing dynamic renaming and providing

forwarding at run-time. This same information also enabled our scheme to implement an efficient commit algorithm. Another advantage of our method is that it will recover faster from misspeculations and can allow the threads to restart the parallel execution.

However, while it is clear that our approach detects violations to flow data dependences much earlier, it could be argued that this is to the cost of more overhead in the validation of each individual load and store access. While Rauchwerger and Padua do not provide any detailed analysis of the code overhead, it is far from convincing that it is much smaller.

Kazi and Lilja [8] present a quite different speculation system which is inspired by the superthreaded pipelined execution model [21]. The execution of a thread is divided up into a number of phases: target-store-address-generation (TSAG), computation, and write-back (WB). Unique to their approach is that consecutive speculative threads execute these stages in a pipelined fashion. For instance the TSAG stage must be completed by the previous thread before the current one can enter it. As a consequence, the WB stage, which commits speculative data, is carried out serially across all the speculative threads. In our approach, this serialization is effectively prevented. In addition, all modifications to the same variable by different threads must in their scheme be committed one-by-one. In our scheme, committing a single value will do. We have seen that parallel commit is indeed important when we use many processors.

Their implementation can also eliminate some true data dependences through forwarding. However, while it is difficult to fairly compare the overheads of their and our scheme, it appears as the cost of providing forwarding in their scheme is very high. First, the write addresses need to be broadcast to all threads and next an event synchronization is needed to signal the availability of the forwarded data. In our scheme, we do forwarding if it happens to occur on time; otherwise we simply signal a flow dependence violation. This approach effectively eliminates the overhead of event synchronization and the broadcast of addresses involved in Kazi and Lilja's scheme. In our analysis, we showed the importance of avoiding synchronizations.

The idea to associate with speculative memory operations highly-tuned checking codes is related to the approach taken in Shasta [17] to implement a fine-grain coherence protocol entirely in software. While the protocol is quite different from a speculation system it is interesting to see that they also managed to keep the overhead low. Some of the code optimization tricks they tried were to amortize the checking code overhead across consecutive loads and stores to the same cache block. We tried to also pursue the idea of amortizing the overhead over a number of speculative operations, but have not

yet found it useful in the context of speculation systems.

6 Conclusions

In this paper, we have proposed a new software-based approach to design a thread-level data dependence speculation system in which the key objective has been to achieve low software overheads. Like aggressive hardware-based speculation systems, our approach is to resolve all name dependences through renaming and also some flow data dependences through forwarding by keeping track of dependence violations on the granularity of single variables. We have demonstrated that the code attached to each load and store that potentially can cause a dependence violation can be implemented very efficiently; indeed, the critical code path only involves between 7 and 9 instructions. Among the many considerations behind coming up with this low-overhead solutions are locality observations, avoiding expensive synchronizations, as well as the use of parallel commit operations.

We have presented a performance evaluation based on architectural simulation of how much difficult-to-parallelize loops from the Perfect Club benchmarks can be sped up using our method. Our results are encouraging and shows quite significant speedups: 3.3 for *Ocean* on 8 processors and 7.1 and 12.5 for *Adm* and *Track* on 16 processors, respectively. In validating our results, we found that the layout of speculative variables and their associated control data structures may have a significant effect on cache performance. We particularly studied the tradeoffs between instruction overhead and cache miss penalties for two data layout schemes. We also validated the speedup of speculated code on a multiprocessor and showed that the measured speedup corresponded well with the simulated.

This study suggests that the low overhead obtained by purely software-based thread-level data dependence speculation casts a doubt on using purely hardware-based approaches. An interesting avenue for future research is to study what primitives should be supported in hardware by still offering the flexibility of software-based approaches.

Acknowledgments

This research has been supported by a grant from Swedish Research Council on Engineering Sciences (TFR) under contract 221-98-443. Sun Microsystems Inc. has contributed with a donation of a multiprocessor server which has been instrumental to carry out this research. We would like to thank Erik Stage and Reine Stenberg for their invaluable input on implementation details.

References

[1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor", *Proc. 31st. Int. Symposium on Microarchitecture*, 1998.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrun, and J. Martin, "*The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers*", Technical Report CSRD-827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, May 1989.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel Programming with Polaris", *IEEE Computer*, Vol. 29, No. 12, pp. 78-83, Dec 1996.

[4] M. Cintra, J. Martinez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors", in *Proc. of 27th Ann. Int. Symposium on Computer Architecture*, pp. 13-24, June, 2000.

[5] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "*A Fortran-to-C Converter*", Technical Report No. 149, AT&T Bell Laboratories, Murray Hill NJ 07974, May 1990.

[6] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion and M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", *IEEE Computer*, Vol. 29, No. 12, pp. 84-89, Dec 1996.

[7] J. Hennessy, and D. Patterson. "*Computer Architecture: A Quantitative Approach*", Morgan Kaufmann Publishers, 1990, 1996.

[8] I. Kazi and D. Lilja, "Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors", in *Proc. of 1998 Int. Conf. on Supercomputing*, pp. 93-100, July 1998.

[9] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Binaries on a Chip-Multiprocessor", in *Proc. of 1998 Int. Conf. on Supercomputing*, July 1998.

[10] V. Krishnan and J. Torrellas. "A Chip-Multiprocessor Architecture with Speculative Multithreading", in *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pp. 866-880, Sep. 1999.

[11] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, B. Werner, "SimICS/sun4m: A Virtual Workstation", in *Proc. of Usenix Annual Technical Conference*, June 1998.

[12] P. Marcuello, A. González, and J. Tubella, "Speculative Multithreaded Processors", in *Proc. of the 12th Int. Conf. on Supercomputing*, pp. 77-84, 1998.

[13] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors", in *Proc. of the 13th Int. Conf. on Supercomputing*, pp. 365-372, 1999.

[14] J. Oplinger, D. Heine, S. Liao, B. Nayfeh, M. Lam, and K. Olukotun. "*Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor,*" Tech. Report Computer Systems Laboratory, Stanford University, CSL-TR-97-715, Feb. 1997.

[15] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization." in *Intl. Symp. on Computer Architecture*, pp. 204-215, June 2001.

[16] L. Rauchwerger and D. Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," in *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, No 2., February 1999, Vol. 10, No 2., pp. 160-199, Feb. 1999.

[17] D. Scales, K. Gharachorloo, and C. Thekkath. "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proc. of Seventh Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 174-185, Oct. 1996.

[18] G. Steffan and T. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," in *Proc. of Fourth Int. Symp. on High-Performance Computer Architecture*, pp. 2-13, Feb. 1998.

[19] G. Steffan, C. Colohan, A. Zhai, and T. Mowry. "A Scalable Approach to Thread-Level Speculation," in *Proc. of 27th Ann. Int. Symposium on Computer Architecture*, pp. 1-12, June, 2000.

[20] G. Steffan, C. Colohan, A. Zhai, and T. Mowry. "Improving Value Communication for Thread-Level Speculation", in *Proc. of Intl. Symp. on High-Preformance Computer Architecture*, Feb. 2002.

[21] J. Tsai and P. Yew. "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," in *Proc. of 1996 Parallel Architectures and Compilation Techniques (PACT)*, Oct. 1996.

[22] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors," in *Proc. of Fourth Int. Symp. on High-Performance Computer Architecture*, pp. 162-173, Feb. 1998.