Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors*

Marcelo Cintra
School of Informatics
University of Edinburgh
Edinburgh, UK
mc@inf.ed.ac.uk

Diego R. Llanos
Departamento de Informática
Universidad de Valladolid
Valladolid, Spain
diego@infor.uva.es

ABSTRACT

With speculative parallelization, code sections that cannot be fully analyzed by the compiler are aggressively executed in parallel. Hardware schemes are fast but expensive and require modifications to the processors and memory system. Software schemes require no extra hardware but can be inefficient.

This paper proposes a new software-only speculative parallelization scheme. The scheme is developed after a systematic evaluation of the design options available and is shown to be efficient and robust and to outperform previously proposed schemes. The novelty and performance advantage of the scheme stem from the use of carefully tuned data structures, synchronization policies, and scheduling mechanisms. Experimental results show that our scheme has small overheads and, for applications with few or no data dependence violations, realizes on average 71% of the speedup of a manually parallelized version of the code, outperforming two recently proposed software-only speculative parallelization schemes. For applications with many data dependence violations, our performance monitors and switches can effectively curb the performance degradation.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming

General Terms

Performance

Keywords

Speculative parallelization, Thread-Level Speculation

1. INTRODUCTION

Although parallelizing compilers have proven successful for a large set of codes, they fail to parallelize codes when data dependence information is incomplete. Such is the case of accesses through pointers or subscripted subscripts, complex interprocedural data flow, or input-dependent data and control flow. In these cases, run-time parallelization in software has been explored under two major approaches: *inspector-executor* [15, 25] and *speculative parallelization* [7, 8, 21, 23, 24]. With the inspector-executor scheme, an inspector loop is extracted from the original loop with the purpose of computing the cross-iteration data dependences to guide the execution of the executor loop. This approach is effective when computing the address reference stream is cheap compared to the actual computation. In many cases, however, the overhead of the inspector loop limits the performance benefits of this approach. Under the speculative parallelization (also called thread-level speculation) approach, the code is speculatively executed in parallel while the reference stream is monitored for data dependence violations. If a dependence violation is found, the system reverts the state back to some safe condition and threads are re-executed.

While various degrees of hardware support for speculative parallelization on multiprocessors have been proposed in the literature (e.g., [5, 9, 16, 20, 26, 29, 31]), these are costly and require modifications to the processors and caches. In this paper, we focus on software-only implementations of speculative parallelization. In this case, the user application itself is augmented with code to perform all the speculative operations.

The contributions of this paper are twofold. First, we systematically explore the design options for software speculative parallelization and analyze the tradeoffs involved. In this process we place previously proposed schemes into a single framework of high-level design choices and we quantitatively evaluate part of this design space. The second contribution of this paper is the design of a highly cost-effective software speculative parallelization scheme. The novelty and performance advantage of the scheme stem from:

- Aggressive scheduling mechanism based on a sliding window, which reduces the impact of load imbalance across threads as well as the memory overheads associated with the speculative access and version data structures.
- Synchronization policies that relax the critical section requirements of previous work, while still allowing checks for cross-thread data dependence violations upon speculative memory operations.
- Speculative access data structures that allow for efficient implementation of the speculative operations regardless of the size of the user speculative data structures.

Experimental results show that our scheme has small overheads and is able to reach a large fraction of the potential parallel execution performance. In particular, for applications with few or no data dependence violations, the scheme realizes on average 71% of the speedup of a manually parallelized version of the code. The results also show that the scheme outperforms two recently proposed

*This work was supported in part by the European Commission under grant HPRI-CT-1999-00026 and by EPSRC under grant GR/R65169/01.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

software-only speculative parallelization schemes: one by 25% on average and with similar memory overhead, and the other by 7% on average but with significantly less memory overhead. For applications with many data dependence violations, our performance monitors and switches can effectively curb the performance degradation.

The rest of the paper is organized as follows: Section 2 describes speculative parallelization and highlights the operations involved; Section 3 presents our proposed speculative parallelization scheme and its rationale; Section 4 describes our evaluation methodology; Section 5 presents the experimental results; Section 6 discusses related work; and Section 7 concludes the paper.

2. SOFTWARE-ONLY SPECULATIVE PARALLELIZATION

2.1 Basic Concepts

Under speculative parallelization threads are extracted from sequential code and run in parallel, hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables, while loads from speculative threads are provided with potentially incorrect versions. At special points in time data versions that have become safe must be *committed* to safe storage.

As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. *Read-after-write* (RAW) dependence violations occur whenever a speculative thread consumes some version of the data other than the one produced by the proper store by a predecessor thread. When the memory accesses of such dependences occur *in-order* at run time, a violation can be prevented by *forwarding* the value produced by the predecessor thread. *Write-after-write* (WAW) dependences can cause violations when the speculative parallelization scheme maintains access information at a coarser granularity than that of the user data structure. In this case it is not possible to disambiguate the two modifications and a violation by the successor thread must be assumed. *Write-after-read* (WAR) dependences usually do not cause violations as modifications by successor threads are contained in their respective versions and cannot be consumed by predecessor threads.

When data dependence violations are detected, the offending thread must be *squashed*, along with any other threads that may have an inconsistent state. These usually include all successors of the offending thread. When a thread is squashed, all the data that it speculatively modified must be purged from the memory hierarchy and the thread then restarts execution from its beginning. During re-execution the thread can be then provided with the updated value.

From the above discussion, we can summarize the main operations required by speculative parallelization as follows: (1) identify and mark data that can be accessed in an unsafe manner (the *speculative data*) and the loads and stores that can potentially access these data (the *speculative loads and stores*), (2) maintain access information to speculative data (the *speculative access state*), (3) schedule speculative threads, (4) buffer speculative data and commit it to safe storage when appropriate, (5) detect data dependence violations, squash, and restart threads as necessary. In the following sections we give an overview of these operations and present some of the design options available.

2.2 Identifying Speculative Data and Memory References

The first step in preparing the application code for speculative parallelization is to identify and mark the speculative data. The compiler analysis required can be easily built on top of the data dependence and data flow analyses of existing automatic parallelizing compilers. All data that can potentially be used by a thread before being modified by it have to be marked as speculative. In this case we must both maintain multiple versions of the data and look out for cross-thread RAW dependence violations. Data that is identified by the compiler as read-only does not need to be marked as speculative. Finally, for data that is guaranteed to be defined before being used in the same thread, the system must maintain multiple versions of the data, but there is no need to look out for data dependence violations.

Once the speculative data are identified, the compiler must identify all their uses and definitions (loads and stores) and replace these with special operations to update the speculative access information appropriately (Section 2.3). When the compiler can identify that a load to a certain speculative datum is dominated by some other load or store to the same datum, then the dominated load does not have to be augmented with speculative operations. This is not always true for stores. Idempotent loads and stores also do not need to be augmented with speculative operations [13].

2.3 Maintaining Access Information

To maintain multiple speculative versions and track data dependences, special data structures are generated for every user speculative data. To maintain multiple versions, the user data structure must be replicated for each thread that can be active at any given time (Section 2.4). These are called the *version copies* of the user data. To track accesses to different parts of the user speculative data structure, we must create a *speculative access structure* that keeps per-thread access information for each of such parts. How closely the speculative access structure identifies individual data in the user data structure is referred to as the *granularity* of the access information.

Typically, each entry in the speculative access structure should record whether the corresponding part of the user data structure has been: not accessed by the thread, modified by the thread, exposed loaded by the thread, or exposed loaded and later modified by the thread. An *exposed load* occurs when a thread issues a load without having previously issued a store to the same data. Such loads can potentially cause RAW dependence violations.

Upon a speculative store, the corresponding entry in the access structure must be updated: from not accessed to modified and from exposed loaded to exposed loaded and modified. The store is then performed to the corresponding version copy of the data. Upon a speculative load, if the corresponding entry in the access structure is not accessed then it must be changed to exposed loaded, otherwise it remains in the same state. If forwarding is supported then the most up-to-date version is located by searching the access structure backward for the closest predecessor entry in state other than not accessed. Otherwise, the current (or *reference*) value is returned.

2.4 Scheduling Speculative Threads

In the simplest approach to scheduling threads under speculative parallelization, we can divide the iteration space in as many chunks as there are processors and *statically* assign chunks to processors. Alternatively, we can create more chunks than there are processors and *dynamically* schedule these chunks on the processors.

Recall that under speculative parallelization, each thread must be assigned a version copy and a speculative access structure. Thus, the number of such structures must be equal to the maximum num-

ber of chunks that can be active at any time. With the static and dynamic scheduling policies, this corresponds to the total number of chunks created. An alternative to these policies is to decouple the number of possibly active threads from the total number of chunks by using a *sliding window* over the iteration space [5, 7]. In this case a window of W active threads slides over the series of chunks created, which can be much larger than W.

2.5 Committing Safe Data

Speculative modifications to the user data are temporarily stored in the version copies. Physically, these version copies can be implemented as private or as shared data. Supporting forwarding requires that the version copies be shared. At some point after becoming safe, version copies must be committed to safe storage, which is usually the user data structure itself. The commit at the end of the speculative execution we call a *final commit*. After the final commit the user data structure is in the state it would have been if the speculative section had been executed sequentially. In addition to the final commit, we can perform intermediate *partial commits*. Performing intermediate partial commits simplifies the roll-back in case of squash, frees up version storage, and reduces the amount of data that has to be committed at the end of the speculative execution. The main disadvantage of partial commits is the execution time overhead.

2.6 Squashing and Restarting Threads

Data dependence violations are detected by looking at the speculative access structures: a RAW dependence violation has occurred whenever there is an entry in state exposed loaded or exposed loaded and modified, with some predecessor entry in state modified or exposed loaded and modified. Checking for violations, and squashing, regularly can incur execution overheads but prevents processors from performing much useless work.

We can check for data dependence violations upon the speculative loads and stores themselves. When scheduling policies based on windows are used, violation checks must be performed at least when the window must be advanced, since at this time the speculative access information is lost. With static or dynamic scheduling policies we can postpone the checks to the end of the execution of the speculative section.

When data dependence violations are detected we must squash and re-execute threads that may have consumed incorrect values. These are usually the offending thread and all its successors.

When a squash occurs the speculative data generated by squashed threads must be purged and the speculative access information cleared. Additionally, when the threads are restarted some mechanism must be provided to prevent the same violation from occurring again. One way is to perform forwarding. Another way is to remember the violation and avoid it with synchronization [6, 19, 30]. Alternatively, we can wait to restart threads until all the predecessors have committed, in which case the squashed threads can use the safe version of the data.

3. COST-EFFECTIVE SOFTWARE SPECULATIVE PARALLELIZATION

In this section we present the design of a new scheme for software speculative parallelization. We highlight the major features and design decisions along with the rationale for the choices.

3.1 Sliding Window

The main design decision in our scheme was to implement in software a sliding window mechanism similar to the hardware mechanism of [5]. Traditionally, scheduling of parallel loops is either static or dynamic. With static scheduling, the iteration space is partitioned into P chunks of iterations, where P is the number of processors. With dynamic scheduling individual iterations are dynamically assigned to processors. Both policies are undesirable under speculative parallelization. Static scheduling will perform poorly when there is load imbalance or when there are data dependence violations. Dynamic scheduling is not practical when the number of iterations, T, is very large, because the memory overhead of the speculative structures is proportional to the number of iterations. Alternatively, we can partition the iteration space in C chunks of iterations, with $P < C < T$, and dynamically schedule these chunks on processors. This alleviates, but does not solve, the problems of the static and dynamic policies. In fact, tolerance to load imbalance and data dependence violations is now proportional to C, but memory overhead is also proportional to C. To keep memory overhead tolerable, C must be a small multiple of P, which may not be enough to limit the impact of load imbalance and data dependence violations.

A *sliding window* mechanism [5, 7] can better decouple the memory overhead from the number of iterations. In these schemes, threads consist of chunks of a small number of iterations, but scheduling is limited to a window of W chunks at a time. If $W > P$ then threads within a window can be statically or dynamically scheduled. At any time there are only W active threads and the memory overhead is proportional to W, regardless of the total number of chunks, C. Nevertheless, tolerance to load imbalance and data dependence violations is significantly increased because C can still be very large and the sliding of the window approximates the behavior of a dynamic schedule across C.

There are two possible schemes for sliding the window: once all threads in the window are completed [7], and every time non-speculative threads commit [5]. Despite their larger complexity and management overhead, we expect schemes based on windows to perform better across a broad range of situations.

In our implementation, the sliding window mechanism consists of an array of characters of length W containing a status descriptor for each uncommitted (active) thread. Additionally, two integers mark the boundaries of the window at any time, pointing to the non-speculative and the most-speculative threads (top part of Figure 1).

In the window array, thread slots can be in the following states: `FREE`, meaning that there is no thread associated with this slot at the moment; `RUN`, meaning that the thread is still being executed; `DONE`, meaning that the thread has been executed to completion; and `SQUASHED`, meaning that the thread has been involved in a violation, directly or indirectly, and must be squashed.

During speculative execution, the non-speculative and most-speculative pointers in the window structure are used by speculative loads and stores to determine the section of the speculative access array that must be checked. At the end of a thread's execution, the thread states in the window structure are used to determine what threads must be committed, if any, and what thread to run next.

3.2 Speculative Access Structure

The implementation of the speculative access structure has a direct impact on the execution overheads of checking for data dependence violations and committing. The simplest data structure that we can choose is an array of access state of size M for each active thread (M is then the number of parts of the user data structure that can be unambiguously tracked). This approach is very efficient for checking for data dependence violations on speculative memory operations, since then the system knows exactly where to obtain the relevant access information for the part of the user data structure being accessed. However, this approach is very inefficient when committing and when checking for data dependence violations on

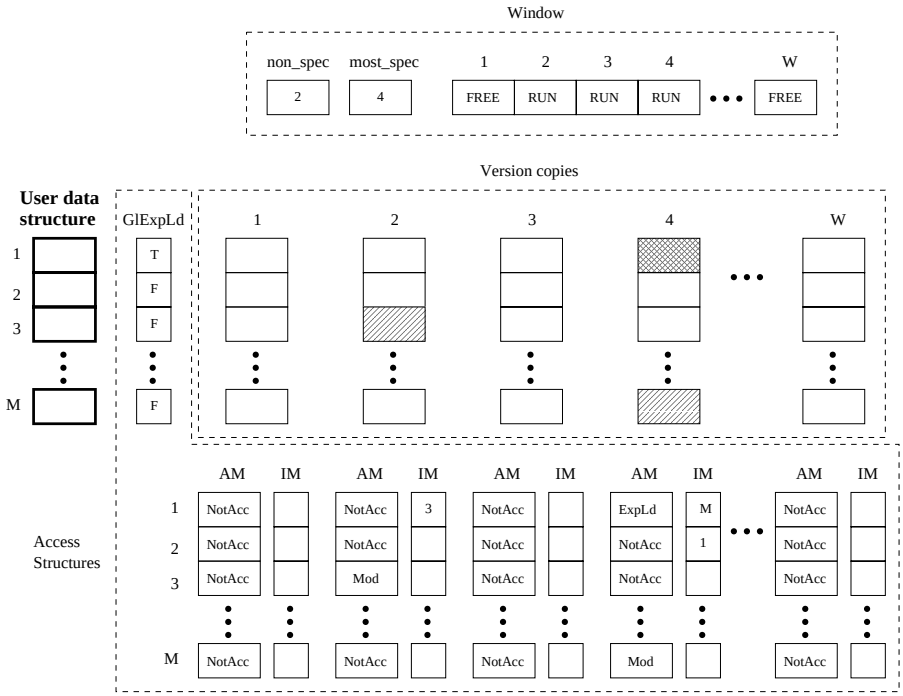

Figure 1: Data structures used in our software speculative parallelization scheme. In the version copies, the single-striped boxes correspond to `Mod` data and the double-striped boxes correspond to `ExpLd` data.

multiple elements of the user data structure, since in these occasions all M access information would have to be checked.

When the user speculative data is very large compared to the amount of data actually used by each thread, we can reduce these overheads by implementing the speculative access structure as a list of the indices of the user data elements actually touched. With such a structure, the search for violations and the commit of modified data stop when the end of the list is reached. This approach, however, is not well suited for checking data dependence violations on every speculative memory operation because of the extra overhead of walking the list.

To support both fast commits and fast checks for data dependence violations upon memory accesses, we use a set of three speculative access structures. The first structure is an MxW array of characters, where W is the maximum number of active threads (the window size). In our system, M is equal to the total number of elements of the user structure that can be independently accessed. We call it *AM*, for *Access Matrix* (bottom part of Figure 1). Each element in this speculative access structure encodes the following four states: not accessed data (`NotAcc`), exposed loaded data (`ExpLd`), modified data (`Mod`), and exposed loaded and modified data (`ExpLdMod`). This access structure allows for quick lookups upon speculative loads and stores for any particular element of the user structure.

The second speculative access structure is an MxW array of integers where the first elements in each column point to elements of AM in states other than `NotAcc`. We call this structure *IM*, for *Indirection Matrix* (bottom part of Figure 1). The last element in each column of IM that corresponds to an accessed element of AM is identified by a *tail pointer* that is part of an array of W integers (not shown in Figure 1). The IM access structure is traversed on commits to quickly identify the user data actually used by a thread.

To further speed up the search for data dependence violations, we use a third access structure: a single array of M logical values. Each element can be in state either `ExpLd` (`TRUE`) or `Safe` (`FALSE`). The `ExpLd` state indicates that at least some thread, since the start of the speculative execution, has performed an exposed load to this particular element of the user data, while the `Safe` state indicates that no thread has ever performed an exposed load to this element. This access structure is useful in applications where the memory accesses of threads do not overlap at all, or overlap but are write-first. We call this structure *GlExpLd*, for *Global Exposed Load* (left part of Figure 1).

In addition to the type of data structure used, the actual layout in memory of the access structure has a second-order effect due to locality of references and cache misses. For instance, the structure can be laid out in memory such that elements along W or M are consecutive. When checking for data dependence violations on a speculative load or store, multiple access information data corresponding to the same user element are checked in sequence along W. On a commit, multiple access information data corresponding to elements of the user data belonging to the committing thread are accessed in sequence along M. Thus, these operations call for conflicting layouts in memory. Since in practice we expect W to be much smaller than M, we lay out our speculative access structure such that elements along M are consecutive in memory.

3.3 Speculative Loads and Stores

Fast response to data dependence violations requires frequent checks for violations. In general, detecting data dependence violations can be done on every speculative load and store, when threads commit, or simply at the end of the speculative section. Although it may seem that looking for data dependence violations on every speculative load and store is too expensive, this is not necessarily the case. This is because in this case we must only check for violations on a particular element of the user speculative data, while in the other cases we must check for violations on at least all the elements exposed loaded by the threads involved, and potentially on

all elements of the user speculative data. In our system we implement the checks for data dependence violations in the speculative loads and stores.

Figures 2a and 2b show abridged implementations of our speculative load and speculative store operations, respectively, in a C-like syntax. From these figures we can highlight the following features of our scheme. Only the first load to a datum requires special handling by the protocol (line 1 in Figure 2a). The search for predecessor versions of the datum on loads only requires looking up one element of AM per thread (lines 6 to 11 in Figure 2a). Similarly, the search for data dependence violations on stores only requires looking up one element of AM per thread (lines 11 to 15 in Figure 2b). The use of the GlExpLd structure avoids searching for data dependence violations when no thread has performed an exposed load to the datum (line 10 in Figure 2b). Also, the search for data dependence violations can stop early if a successor thread is found to have modified the datum without an exposed load (lines 12 and 13 in Figure 2b). Note that squashes can only be triggered by stores since forwarding is supported. The `squash()` operation simply involves setting the window state of the successor threads to SQUASHED and moving the non-speculative pointer backward. Later when a thread commits (Section 3.4) this will trigger the re-execution of the thread.

3.4 Commits

Figure 3 shows an abridged implementation of the code executed at the end of each thread, in a C-like syntax. This code is divided in two main sections: the commit proper (lines 3 to 16) and the assignment of a new thread (lines 22 to 29). From this figure we can highlight the following features of our scheme. Only the non-speculative thread performs commits (line 3), and it is responsible for committing itself and all successor threads that have already finished (lines 5 to 14). Committing the modified data is limited to checking the elements accessed by the threads, as identified by the IM structure (lines 10 and 11). When the window is full, processors spin-wait without contention until a thread slot is freed (line 22). Finally, before starting a new thread the AM structure is efficiently cleared for reuse by using IM (lines 24 and 25). After incrementing the most-speculative pointer and securing a slot in the window, the processor is ready to grab another thread to execute (code not shown, for simplicity). The `do_squash()` operation in practice simply requires setting the window slot to FREE.

3.5 Protocol Races

As described so far, our protocol for detecting data dependence violations and for partial commits suffers from race conditions. These races are caused by uses of and updates to the shared window structure and the shared speculative access structures. More specifically, races appear between the speculative loads and stores upon accesses to GlExpLd, AM, and version; between speculative loads and commit upon accesses to ref and non_spec; between speculative stores and commit upon accesses to AM and most_spec; and between commit and thread start upon accesses to non_spec, most_spec, and window. We have marked these conflicting memory operations in Figures 2 and 3. For instance, Ld_2 in Figure 2a refers to the load of non_spec embedded in line 6 and Ld_d in Figure 2b refers to the load embedded in lines 12 and 14 of the particular element of AM that can cause a conflict with an ongoing speculative load.

We can divide the races in two major cases: those that appear due to the protocol itself when executed in strict program order and when the memory operations follow a sequential consistency memory model [1]; and those that appear when the compiler may re-order the operations in the protocol and/or when the system only enforces some relaxed memory consistency model that allows both loads and stores to bypass each other, as is the case in Sun's SPARC [27] and IBM's PowerPC systems [18].

- Race 1: speculative load vs. speculative store

  According to the protocol in Figure 2a, upon an exposed speculative load the consumer must perform the following three actions: set the corresponding state in the speculative access structures (St_1), scan the access array backward for the most up-to-date version of the data (Ld_3), and copy the forwarded value (Ld_4). In contrast, as shown in Figure 2b, upon a speculative store the producer must perform the following three actions: modify its version copy (St_a), set the corresponding element in the access array (St_b), and scan the access array forward for possible data dependences (Ld_d). It can be shown that, as long as the operations occur in program order and follow a sequential consistency model, all possible interleavings of these operations maintain the semantics of the original sequential program, with either the new value being forwarded or a data dependence violation being detected. Note, however, that some interleavings can lead to the detection of a data dependence violation even when a successful forwarding is performed. For instance, consider the following interleaving of the operations: $St_a \rightarrow St_b \rightarrow St_1 \rightarrow Ld_3 \rightarrow Ld_4 \rightarrow Ld_d$. In this case the consumer performs a forwarding with the correct up-to-date value, but the producer still detects a data dependence violation. The problem is that we do not guarantee atomicity of the speculative store operation. We note, however, that this does not affect the correctness of the execution and will lead to some performance degradation only in the presence of data dependences and races.

  Consider again the operations described above. It can be shown that incorrect behavior occurs if those loads and stores are not globally performed in program order. For instance, if loads and stores are allowed to bypass preceding stores, then the order $St_b \rightarrow Ld_3 \rightarrow Ld_4 \rightarrow Ld_d \rightarrow St_1 \rightarrow St_a$ can lead to a consumer performing a forwarding with a stale value while the producer fails to detect a data dependence violation. The problem in this case is that the store to the AM element by the producer thread (St_b) performs with respect to the consumer thread before the store to the version copy (St_a).

- Race 2: speculative load vs. commit

  According to the protocol in Figure 3, when a thread commits, it must perform the following two actions: update the reference values (St_A) and advance the non-speculative pointer (St_B). Upon a speculative load, a successor consumer thread scans the access array backward for the most up-to-date version of the data, using the non-speculative pointer to delimit the end of the search (Ld_2 in Figure 2a), and may read the reference copy (Ld_5) if no such version is found. As long as the operations occur in program order and follow a sequential consistency model, all possible interleavings of these operations maintain the semantics of the original sequential program, with the correct value being loaded by the consumer thread either from the already updated reference copy or from the still up-to-date version copy.

  If, however, the update to the non-speculative pointer is allowed to bypass the updates to the reference array, then the consumer thread could obtain a stale version from the not yet updated reference copy, leading to incorrect execution. For

```
      1.  if (AM[I][tid] == NotAcc) {
      2.    GlExpLd[I]=TRUE;
St1 [ 3.    AM[I][tid]=ExpLd;
      4.    IM[++tail[tid]][tid]=I;
      5.  #pragma memory fence
Ld2 [ 6.    for (j=tid-1; j>=non_spec; j--)
Ld3 [ 7.      if (AM[I][j] != NotAcc) {
Ld4 [ 8.        version[I][tid]=version[I][j];
      9.        forwarded=TRUE;
     10.        break;
     11.      }
     12.    if (!forwarded)
Ld5 [ 13.     version[I][tid]=ref[I];
     14.  }
     15.  lvalue=version[I][tid];
```

(a)

```
Sta [  1.  version[I][tid]=rvalue;
       2.  #pragma memory fence
       3.  if (AM[I][tid] == NotAcc) {
Stb [  4.    AM[I][tid]=Mod;
       5.    IM[++tail[tid]][tid]=I;
       6.  }
       7.  if (AM[I][tid] == ExpLd)
Stb [  8.    AM[I][tid]=ExpLdMod;
       9.  #pragma memory fence;
      10.  if (GlExpLd[I])
Ldc [ 11.    for (j=tid+1; j<=most_spec; j++)
Ldd [ 12.      if (AM[I][j] == Mod)
      13.        break;
Ldd [ 14.      else if (AM[I][j] != NotAcc)
      15.        {squash(j); break;}
```

(b)

Figure 2: Abridged C-like code for speculative loads (a) and speculative stores (b). The `memory fence` directives are discussed in Section 3.5. In this figure `I` is the index corresponding to the element of the user structure being operated on, `tid` identifies the thread performing the operation, `ref` corresponds to the original user data structure, and `lvalue` and `rvalue` correspond to the variable or expressions used in the original operations.

```
      1.  #pragma critical
      2.  if (window[tid] != SQUASHED) {
      3.    if (tid == non_spec) {
      4.      window[tid]=DONE;
      5.      for (i=non_spec; i<=most_spec; i++) {
      6.        if (window[i] == DONE &&
                    window[i+1] != DONE)
      7.          {last=i; break;}
      8.      }
      9.      for (j=non_spec; j<=last; j++) {
     10.        for (k=1; k<=tail[j]; k++)
StA [ 11.          ref[IM[k][j]]=
                    version[IM[k][j]][j];
     12.  #pragma memory fence
     13.        window[j]=FREE
     14.      }
StB [ 15.      non_spec=last+1;
     16.    }
     17.    else
     18.      window[tid]=DONE;
     19.  }
     20.  else do_squash();
     21.  #pragma end critical
     22.  while(window[most_spec+1] != FREE) {}
     23.  #pragma critical
     24.  for (j=1; j<=tail[most_spec+1]; j++)
StC [ 25.    AM[IM[j][most_spec+1]][most_spec+1]=NotAcc;
     26.  #pragma memory fence
     27.  tail[most_spec+1]=0;
     28.  window[most_spec+1]=RUN;
StD [ 29.  most_spec++;
     30.  #pragma end critical
```

Figure 3: Abridged C-like code executed at the end of each thread's execution. The `memory fence` and `critical` directives are discussed in Section 3.5. In this figure `ref` corresponds to the original user data structure.

instance, in the order $St_B \rightarrow Ld_2 \rightarrow Ld_5 \rightarrow St_A$, the consumer will see the new value of the non-speculative pointer and will stop the backward search before reaching the committing producer and will read the reference value before it is actually updated.

- Race 3: speculative store vs. thread start

  According to the protocol in Figure 3, when a new speculative thread starts it must reset the old elements of the access structure (St_C) and increment the most-speculative pointer (St_D). Upon a speculative store, a predecessor thread scans the access array forward for any data dependence violation, using the most-speculative pointer to delimit the end of the search (Ld_c) and checks the access structure for an exposed load (Ld_d). As long as the operations occur in program order and follow a sequential consistency model, all possible interleavings of these operations lead to correct behavior and no unnecessary squashes.

  If, however, the update to the most-speculative pointer is allowed to bypass the updates to the access structure, then the producer thread could incorrectly detect a violation based on an old value of the access array, leading to an unnecessary squash. For instance, in the order $St_D \rightarrow Ld_c \rightarrow Ld_d \rightarrow St_C$, the producer will search for violations based on the new value of the most-speculative pointer and may detect a violation based on the value of the access structure before it is cleared. This does not affect program correctness.

- Race 4: commit vs. thread start and thread start vs. thread start

  If multiple threads are allowed to start simultaneously or while a thread is committing, it can be shown that some improper interleavings of the commit and thread start operations shown in Figure 3 can lead to inconsistent states of the window and speculative access structures. For instance, a deadlock could occur when the earliest running speculative thread finishes while the non-speculative thread is committing. In this case, the non-speculative thread might not realize that one extra thread must be committed (lines 5 and 6), while the speculative thread will not commit itself as it is not yet non-speculative (line 3). Thus this speculative thread will never be committed and once the window becomes full the program will stop making progress. As another instance, if two threads are allowed to start simultaneously they could use the same value of the most-speculative pointer to select the elements of the access structure for reuse (lines 24 and 25), before either of them updates the most-speculative pointer (line

29). In this case, part of the access structure will contain incorrect stale values and will lead to incorrect execution.

- Race 5: commit vs. squash and thread start vs. squash

  The squash operations encapsulated in line 15 of Figure 2b and line 20 of Figure 3, which are not shown in detail for simplicity, require updating the shared window structure. If threads are allowed to commit or start while a squash is being performed, it can lead to inconsistent states of the window and speculative access structures. For instance, if the state of the window is changed to `SQUASHED` (in the `squash()` procedure) after the squashed thread executes line 2 in Figure 3 but before it executes line 18 in this figure, then the final state of the window will be `DONE`, instead of `SQUASHED` and this thread will not be squashed after all.

The simplest way to avoid all the race conditions described above is to envelop all the speculative operations in critical sections, which guarantees mutual exclusivity and atomicity. This solution was used in the approach with simple locks in [24]. This approach, while correct, can be very inefficient and can be shown to be too conservative. In practice, we can relax the constraints on the ordering of the speculative operations and reduce the use of critical sections.

We note that races 1 through 3 only lead to incorrect behavior when program order and sequential consistency are not guaranteed. If the processor and memory system as well as the compiler enforce sequential consistency [1], then correct behavior is guaranteed. However, if the system uses some relaxed memory consistency model then *memory fences* must be inserted to guarantee correctness. A memory fence tells the processor that all pending memory operations must be globally performed before any following memory operation can be issued. It also tells the compiler not to reorder any memory access instructions across the fence.

The fence directives shown in Figures 2 and 3 show the set of fences that must be used when the memory consistency model allows for all types of reordering of memory references. For instance, the fence in line 5 in Figure 2a enforces $St_1 \rightarrow Ld_2, Ld_3, Ld_4, Ld_5$ and the fence in line 2 in Figure 2b enforces $St_a \rightarrow St_b, Ld_c, Ld_d$. Note that the orders $Ld_2 \rightarrow Ld_3 \rightarrow Ld_4, Ld_5$ and $Ld_c \rightarrow Ld_d$ are guaranteed by the control flow constraints and require no extra memory fences. Some subset of these fences may be sufficient in systems that support more strict memory consistency models. To see that these fences guarantee correct behavior one can easily check all the remaining possible interleavings of the operations that respect the memory fences, critical sections and control flow.

In contrast, avoiding races 4 and 5 without a critical section is not as straightforward. In this case, program order of the operations is not enough to guarantee correct behavior. We do not attempt to eliminate this critical section. Note that, despite the critical section around the commit operation, the memory fences to avoid races 2 and 3 are still necessary, since there may be no guarantee on the order of memory operations generated inside a critical section.

3.6 Overhead Monitors

Automatic speculative parallelizing compiler technology is still in its infancy. In many cases we expect the compiler to incorrectly suggest speculative parallelization when in fact there are too many cross-thread data dependences. Moreover, the number of data dependences may vary with the input data. A robust speculative parallelization scheme must provide mechanisms to dynamically limit the slowdowns in such cases. We implement in our scheme a performance monitor that tracks the amount of data dependence violations. It is implemented as two shared counters, *commit_count* and *squash_count*; a flag, *speculate*; and a threshold, *squash_threshold*. When a thread commits, commit_count is incremented by the number of threads being committed. The squash_count counter is incremented whenever a squash is detected. Note that since both commit and squash operations are protected by critical sections there is no race condition to update these shared variables. At the end of the speculative section if the ratio of the squash and commit counters is greater than the threshold, then the speculate flag is set to `FALSE` and no speculation is attempted on future invocations of the speculative section. To switch off speculative execution we simply implement two versions of the loop, one speculative and one sequential, guarded by a test of the speculate flag.

4. EVALUATION METHODOLOGY

4.1 Applications

To evaluate our scheme, we choose the following applications: *TREE* from [3], *WUPWISE* and *LUCAS* from SPECfp2000 [28], *MDG* from the PERFECT Club suite [4], and *AP3M* from [12]. These applications are representative of legacy as well as recent sequential scientific Fortran programs. The input sets are the standard ones provided with the applications. All applications spend a large fraction of their sequential execution time on loops that cannot be automatically parallelized by state-of-the-art compilers.

Table 1 shows, for each application, the loops that we attempt to parallelize speculatively, the fraction of the sequential execution time taken on our Sun server (Section 4.2), the average number of iterations executed per loop invocation, the size of the data accessed through speculative references, and whether the loops present cross-iteration data dependences. In the case of *WUPWISE*, we obtain loops muldeo_200' and muldoe_200' by merging the three outer loops in loop nests muldeo_200 and muldoe_200, respectively. For that, it is necessary to hoist some induction variables and compute the loop indices appropriately, which is within the capabilities of recent compilers [1].

Application	*Loops to Parallelize*	*% of Seq. Time*	*Iterations per Invocation*	*Spec data size (KB)*	*RAW Dependences*
TREE	accel_10	94	4,096	< 1	No
WUPWISE	muldeo_200' muldoe_200'	41	8,000	12,000	No
MDG	interf_1000	86	343	< 1	No
LUCAS	mers_mod_square (line 444)	20	4,194,304	4,000	Yes
AP3M	shgravll_700	78	343 to 2,197	3,000	Yes

Table 1: Characteristics of the applications studied.

4.2 Parallel Execution Environment

We ran the applications described in Section 4.1 on a 24-processor Sun Sunfire 6800 symmetric multiprocessor (SMP). The machine is equipped with 750MHz UltraSPARC-III processors, each with a private 64KByte 4-way set-associative L1 cache, a private 8MByte direct-mapped L2 cache, and 48GBytes of shared main memory. The system runs SunOS 5.8. The system interconnect has a sustained bandwidth of 9.6GBytes/s. The SPARC V9 architecture supports any of three different memory consistency models: *relaxed memory order* (RMO), *partial store order* (PSO), and *total store order* (TSO) [27]. The model enforced depends on the actual configuration of the system. We developed our code assuming RMO, as it is the most relaxed of the three and a program that

[1]Recently, as part of the SPEC OMP parallelization effort [2], loops similar to muldeo_200 and muldoe_200 have been parallelized with help from hand analysis. Such analysis is still beyond the capabilities of automatic parallelization alone.

correctly executes in this model is guaranteed to correctly execute in the other two.

The applications were compiled with Sun Workshop 6 update 2 using the highest optimization settings for our execution environment: `-fast -xchip=ultra3 -xarch=v8plusb -cache=64/32/4:8192/64/1`. They had exclusive use of the processors during the entire execution and we use wall-clock time in our time measurements. For the execution time breakdowns we use the performance collector tool which is part of Sun Workshop. In our experiments the performance collector introduced negligible execution overheads.

We used OpenMP 2.0 to parallelize the loops because of its wide acceptance and portability. The memory fences described in Section 3.5 were then implemented using the OpenMP `flush` directive. The semantics of this directive is different from that of the memory fences: it simply enforces that the processor and memory have a consistent view of some shared object specified in the directive. With proper declaration and placement of these directives we can guarantee that all processors and memory have a consistent view of the shared objects and then mimic the behavior of the memory fences of Section 3.5. A more aggressive implementation of our scheme could use the more selective `MEMBAR` fence provided in the SPARC processor [27].

4.3 Systems Evaluated

The goal of our experiments is to quantitatively evaluate some of the design tradeoffs available to software speculative parallelization schemes as well as to compare our proposed scheme against other possible designs, some of which are similar to schemes previously proposed in the literature. The schemes that we evaluate are the following.

Baseline: uses a window scheme with partial commits when the non-speculative thread finishes, checks for data dependence violations on every speculative store operation, and supports forwarding. This is our baseline system described in Section 3.

sys2: a variation of our baseline scheme that only checks for data dependence violations when threads commit, and does not support forwarding. We evaluate this system to assess the cost of our dependence checking mechanism.

sys3: uses a window scheme with partial commits only when all threads in the window complete, checks for data dependence violations on every speculative store operation, and supports forwarding. This system is similar in concept to the SW-R-LRPD scheme of [7], except that there violations are only checked when the window is moved and the data structures used are different.

sys4: has no window and no partial commits, uses dynamic scheduling of iterations to processors, checks for data dependence violations on every speculative store, and supports forwarding. This system is similar in concept to the scheme of [24], except that we use our own speculative access structures and do not use locks [2].

In addition to the speculative parallelization schemes described above, we also quantitatively evaluate three other scenarios: *Amdahl*: the maximum speedup obtained according to Amdahl's law and the coverage of the speculative loops given in Table 1; *Auto par*: applications automatically parallelized with Sun's compiler using the `-autopar` flag; *DOALL*: speculative loops hand-parallelized with OpenMP directives without any speculative scheme (obviously, this is only possible when the loops have no RAW dependences). In the latter case, `SCHEDULE(DYNAMIC,1)` was used.

[2] In [24], a parallel implementation of the final commit is proposed to reduce its cost when only an access structure equivalent to our AM is used. This optimization is complicated with the use of the IM structure, but it then becomes somewhat redundant as the IM structure already cuts to a minimum the amount of searching for data to commit.

5. EXPERIMENTAL RESULTS

Sections 5.1 to 5.5 evaluate our system with speculative sections that do not suffer from cross-thread data dependence violations (*TREE*, *WUPWISE*, and *MDG*). Section 5.6 evaluates our system with speculative sections that suffer from frequent data dependence violations (*LUCAS* and *AP3M*).

5.1 Overall Speedups

We start by presenting the overall speedup results for the whole applications in Figure 4 (top charts). The systems shown are described in Section 4.3. In this plot, *Baseline* uses the best window sizes we found (Section 5.2). The speedup is given for 2, 4, 8 and, for *TREE* and *MDG*, 16 processors. It is computed with respect to the sequential execution of the unmodified application on a single processor. Because the coverage of the speculative section in *WUPWISE* is below 60% of the sequential execution time, scalability is poor and, thus, we do not present results for 16 processors.

From the figure we observe that, in contrast to automatic parallelization, our scheme delivers speedups for all applications. This demonstrates the importance of speculative parallelization as a complementary technique to traditional parallelization techniques. In all cases, the performance of our scheme is very close (within 82% on average) to the *DOALL* parallel execution speedup and also close (within 79% on average) to the ideal Amdahl's law speedups.

To better visualize how our speculative parallelization scheme compares to manual parallelization, Figure 4 also presents the speedups for the speculative sections only (bottom charts). From this figure we observe that the overheads introduced by our scheme do not prevent it from delivering speedups close to the *DOALL* parallel execution. In fact, our scheme realizes an average of 71% of the parallelization speedup of *DOALL*. This demonstrates that, despite its overheads compared to manual parallelization, speculative parallelization can deliver a large fraction of the total potential performance improvements.

5.2 Effects of Commit Policy

To evaluate the performance impact caused by commits we compare our scheme, *Baseline*, with systems that vary from ours in their commit policies: *sys3* and *sys4*. For the window-based systems we vary the window size to evaluate its impact on the commit overheads and execution times. Figure 5 shows the execution time breakdowns of the speculative sections only, on 4 and 8 processors. On top of each bar we show the speedups relative to sequential execution time. The bars are normalized to the sequential execution time and are broken down into the following components:

Busy: execution time of the original loop body plus OpenMP overhead.

Init: initialization time of the speculative access structures at the beginning of the speculative sections and, for the window based systems, when these structures are re-assigned to new threads (lines 24 to 27 in Figure 3).

Spin: idle time due to load imbalance when waiting for other threads to complete in order to advance the window (line 22 in Figure 3), plus idle time due to load imbalance at the end of the speculative section.

Ld+St: overhead time spent on speculative loads and stores, excluding the original memory operation (all of Figure 2a except line 8 or line 13, and all of Figure 2b except line 1).

Commit: overhead time of the commit operations and setting up of a new thread (lines 2 to 19 and 28 to 29 in Figure 3).

Contention: idle time waiting at the locks and barriers required by the different schemes.

From the figure we see that *Baseline* performs consistently better than *sys3* and, with a large enough window, better than *sys4*. For

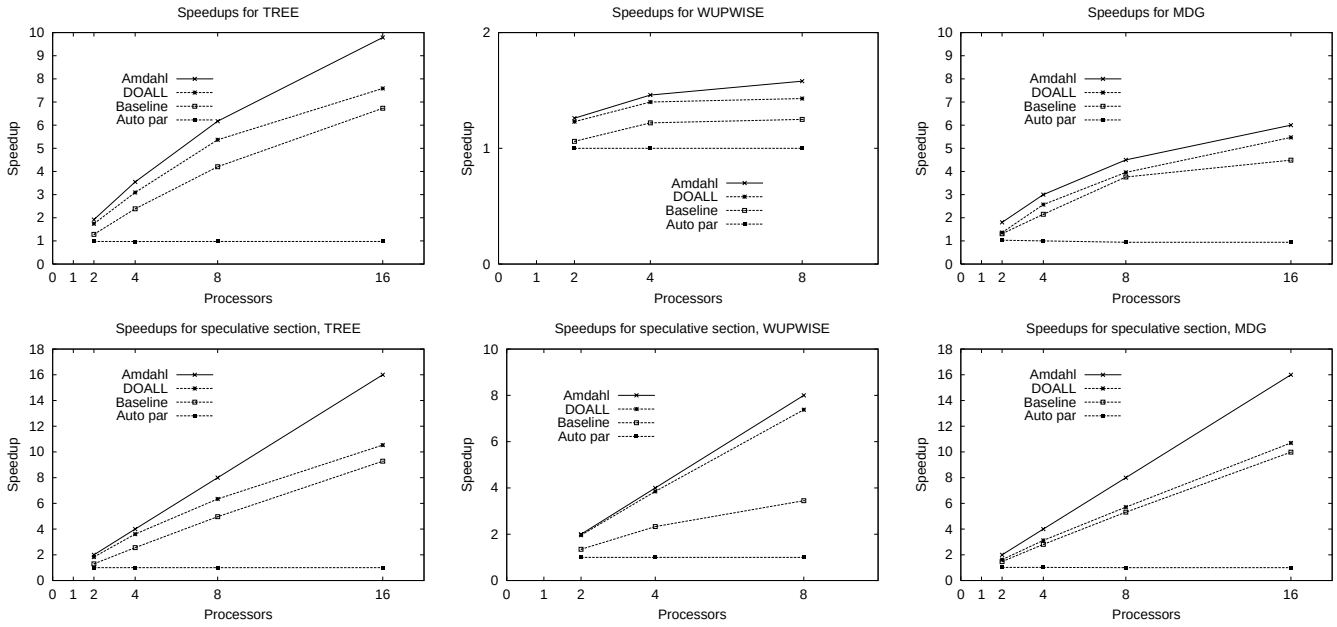

Figure 4: Overall speedups (top charts) and speedups for the speculative sections only (bottom charts).

these 4 and 8 processor configurations, *Baseline* is on average 26% faster than *sys3* and 13% faster than *sys4* for *TREE*; 43% faster than *sys3* for *WUPWISE*; and 5% faster than *sys3* and 1% faster than *sys4* for *MDG*. As expected, *Baseline* suffers less from load imbalance than *sys3* for the same window size, and is able to reduce *Spin* to levels comparable to *sys4* even with small window sizes. The main reason for the performance advantage of *Baseline* over *sys4* is the large commit time required by the final commit in *sys4*.

We also note that with *sys4*, tolerance to load imbalance comes at a high cost of memory overhead and in the case of *WUPWISE* this overhead prevented us from running this scheme in our multiprocessor system.

Finally, we note that as we increase the window size, there is an increase in *Commit* and *Init* for *Baseline* and *sys3*, as expected. This effect reduces the benefits of larger window sizes with *Baseline* and *sys3*, especially for *WUPWISE*. In general we find that a window size of 2 or 4 times the number of processors performs consistently well across all speculative sections.

5.3 Constrained Memory Overhead

In Section 5.2, *Baseline* and *sys3* had the same memory overheads with a given window size, while *sys4* had significantly higher overheads, which even prevented us from running *WUPWISE* with this system. One way to bring the memory overhead of *sys4* to the same levels of *Baseline* and *sys3* is to block the iterations in *sys4* so that the resulting number of chunks is equal to the size of the window in *Baseline* and *sys3*. Figure 6 shows the normalized execution times of the speculative sections only for *Baseline* and a blocked version of *sys4*, on 4 and 8 processors with minimum memory overhead, i.e., window size or number of chunks equal to the number of processors. Again, the numbers on top of the bars show the speedups relative to sequential execution time and the bars are broken down as before.

From this figure we see that blocking *sys4* leads to mixed results. When iterations are not well balanced locally but blocking iterations leads to well balanced chunks, as in *TREE*, *sys4* outperforms *Baseline* (in this case by 31% on average). When iterations are locally well balanced but blocking iterations leads to unbalanced chunks, as in *MDG*, *Baseline* outperforms *sys4* (in this case by 27% on average). Finally, when both iterations and chunks are well balanced, as in *WUPWISE*, other factors, such as the commit overhead, determine the better performing system (in this case *Baseline* by 12% on average).

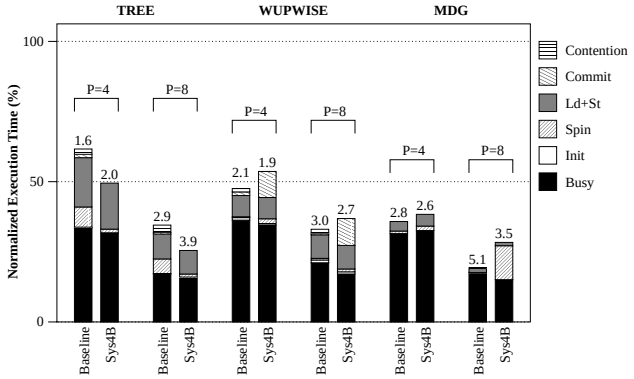

Figure 6: Normalized execution time breakdown for *Baseline* and a blocked version of *sys4* with minimum memory overhead. Results are shown for 4 and 8 processors (P). The numbers on top of the bars are the speedups relative to sequential execution.

Blocking iterations with *sys4* seems a viable option when load imbalance is not significant. However, this technique still has serious limitations when the speculative sections suffer occasional data dependence violations.

5.4 Effects of Dependence Violation Checks

To verify the cost of our policy of checking for data dependence violations on every store operation, we compare our baseline system, *Baseline*, with a similar system that only checks for dependence violations when threads commit, *sys2*. Figure 7 shows the

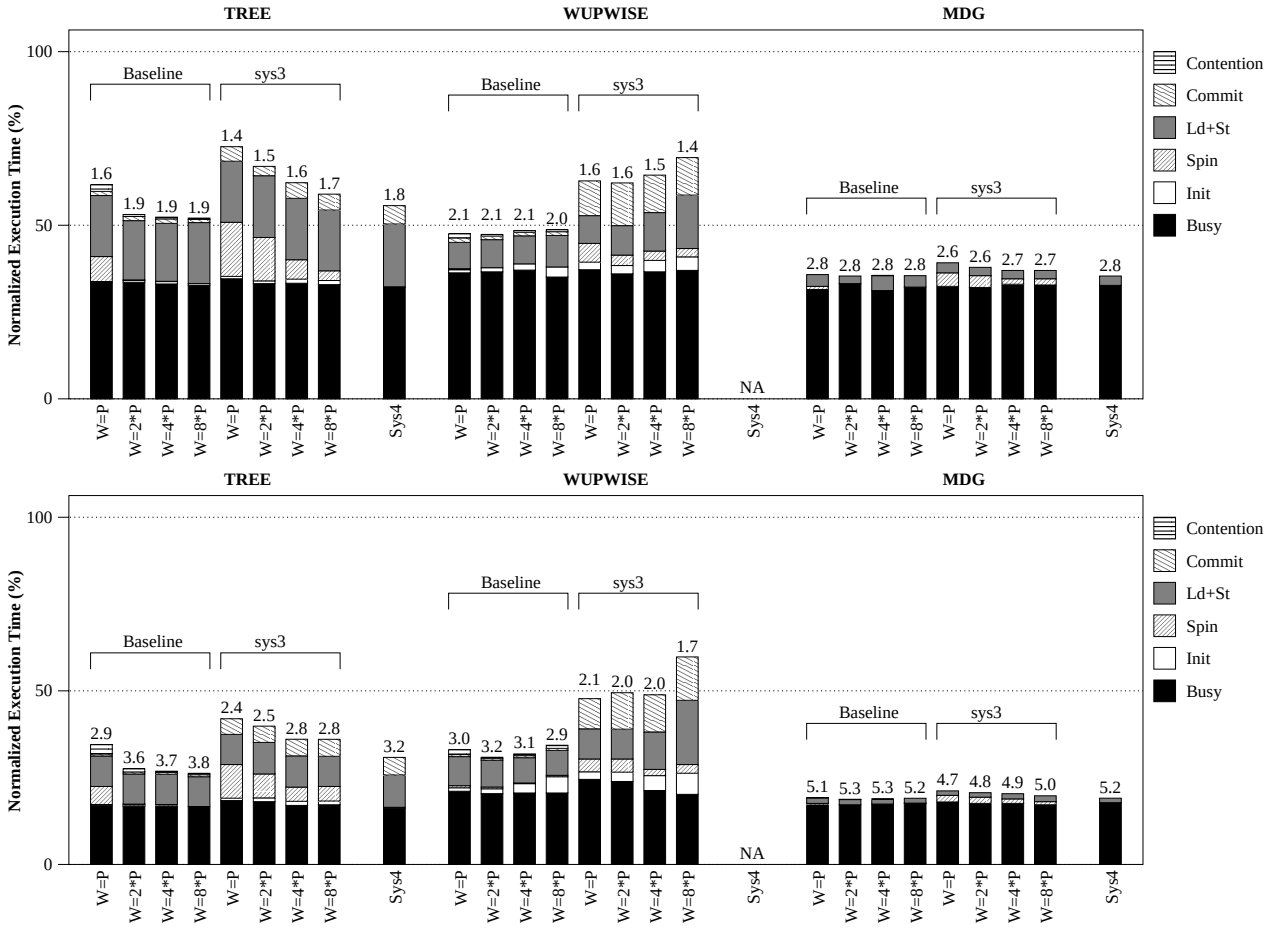

Figure 5: Normalized execution time breakdown for *Baseline*, *sys3*, and *sys4*. For *Baseline* and *sys3* the window size (W) is varied from 1 to 8 times the number of processors (P). Results are shown for 4 processors (top chart) and 8 processors (bottom chart). The numbers on top of the bars are the speedups relative to sequential execution.

normalized execution time of the speculative sections only, on 4 and 8 processors with the best window sizes from Section 5.2. Again, the numbers on top of the bars show the speedups relative to sequential execution time and the bars are broken down as before.

From this figure we see that in all cases the cost of checking for data dependence violations in *Baseline* does not lead to noticeable performance degradation with respect to *sys2*. As expected, *Ld+St* is greater with *Baseline* while *Commit* is greater with *sys2*. This indicates that moving the checks for dependence violations from the memory operations to the end of threads simply moves the overheads from the loads and stores to the commit. However, in the case of *WUPWISE* with 8 processors, the additional time spent on violation checks at commit time with *sys2* significantly increase the contention overhead. This is because the violation checks are then inside the critical section. In this case *sys2* performs significantly worse than *Baseline*. We expect this behavior to occur when the speculative data is large and a large number of processors is used. Overall, *Baseline* offers good and consistent performance along with the opportunity for quicker response to dependence violations.

5.5 Effects of Speculative Access Structure

In this section we evaluate the performance impact of our optimized speculative access structures. Figure 8 shows the normalized execution time breakdown of the speculative sections only for our base system, *Baseline*, and a variation of it without the IM optimization, *noIM*. The bars are broken down as before. Note that to keep the plot readable we truncate several sections of the bar for *WUPWISE noIM* and the height of the bars are much bigger than shown.

Comparing *Baseline* to *noIM* in this figure we see that the IM optimization is crucial for applications with very large speculative data (*WUPWISE*). The cost of scanning all elements of the AM access structure at commit and initialization increases these overheads significantly. The added time at commit also leads to significant increase in both load imbalance and contention overheads.

5.6 Effects of Performance Feedback

In this section we study the performance of speculative parallelization in the presence of cross-thread data dependence violations and we evaluate the importance of having performance monitors and feedback. The loops we use, from *LUCAS* and *AP3M*, are examples of cases where an automatic speculative parallelizing compiler would likely suggest, incorrectly, that speculative parallelization should be used. Figure 9 shows the normalized execution time breakdown of the speculative sections only for our *Baseline* configuration with some values of squash threshold and for a similar system without our squash overhead monitor (*noMonitor*).

We can see from this figure that without squash monitors significant slowdowns are possible. Contention for the squash and com-

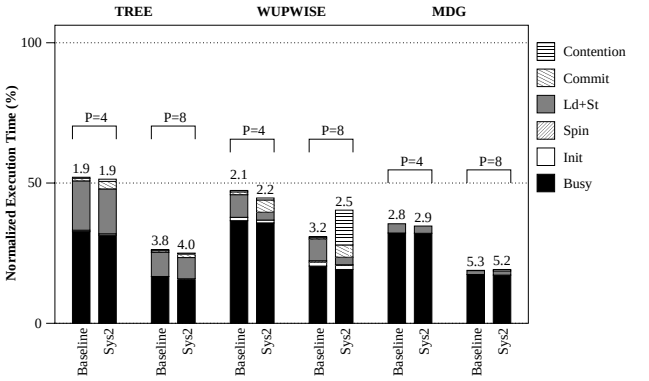

Figure 7: Normalized execution time breakdown for *Baseline* and *sys2* with the best window size from Figure 5. Results are shown for 4 and 8 processors (P). The numbers on top of the bars are the speedups relative to sequential execution.

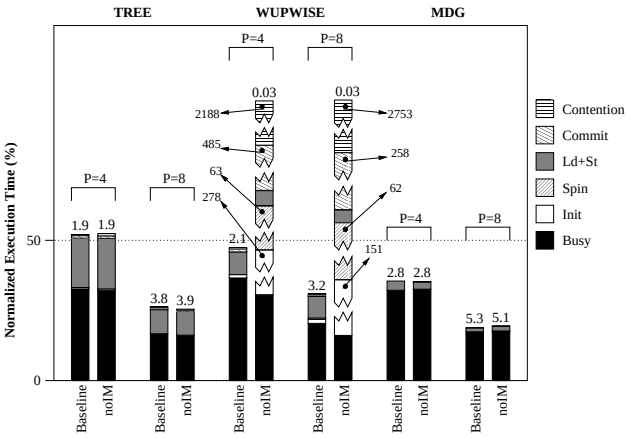

Figure 8: Normalized execution time breakdown for *Baseline* with and without the IM data structure optimization. In all cases we use the best window size from Figure 5. Results are shown for 4 and 8 processors (P). Note that the bars for *WUPWISE noIM* are truncated and the height of the bars are bigger than shown. The values of these truncated sections are shown in the figure. The numbers on top of the bars are the speedups relative to sequential execution.

mit operations account for most of the execution time increase, but re-execution of the original instructions and the speculative operations also contribute to the overall slowdown. With the addition of a squash monitor, our scheme is able to keep the slowdowns to tolerable levels.

A complete study of the sensitivity of the slowdowns to the squash_threshold value is beyond the scope of this paper and we present here only some preliminary observations. In the case of *LUCAS* the number of data dependence violations increases for each invocation of the speculative section and we observe a gradual increase in execution time up to threshold values around 0.15. Between 0.15 and 0.2 there is a sharp rise in execution time and with a threshold of 0.2 all the invocations of the speculative section are run speculatively. In the case of *AP3M*, the knee is even sharper since the number of data dependence violations is effectively the same in all invocations. Thus, as we increase the threshold value, we observe little change in execution time as only the first few invocations run speculatively; after a certain point, all invocations run speculatively. Overall, our experiments seem to indicate that small thresholds (0.01 to 0.1) are required to keep the slowdowns tolerable in most cases, but further study is necessary.

6. RELATED WORK

Run-time speculative parallelization in software was introduced in the LRPD test [23]. Data dependence violations are checked at the end of the tentative parallel execution, and the loop is re-executed sequentially if a violation is detected. Thus, this scheme can only handle fully parallel loops. The scheme in [8] proposed a series of run-time tests, also at the end of the tentative parallel execution. They are tailored for different access patterns and rely on the compiler to identify the most likely behavior. More recently, [7] extended the LRPD work with two new mechanisms. The most aggressive, SW-R-LRPD test, uses a sliding window mechanism somewhat similar to ours. This system differs from ours in three ways: the window only moves when all threads in the window complete; checking for data dependences only occurs after all the threads within a window are finished; and the threads in a window are statically partitioned and assigned to processors. The scheme in [24] applied in software many of the ideas of hardware-based speculative parallelization, such as checking for data dependence violations on memory operations and forwarding. It differs from ours in two ways: no window is used and either locks or a non-scalable byte-vector implementation of the access structures are used to avoid races in the protocol. The work in [21] takes a different approach to software speculative parallelization by placing most of the operations in the software distributed coherence engine.

Several hardware approaches for speculative parallelization have been proposed (e.g., [5, 9, 16, 20, 26, 29, 31]). While these alleviate many of the overheads of speculative parallelization by moving some of the operations to hardware, they require significant changes to the hardware structures, such as caches, protocol controllers, and even the processors.

Alternatively to speculative parallelization, inspector-executor schemes [15, 25] pre-compute the reference stream and use the dependence information to execute the loop in parallel with explicit synchronization where necessary. This approach works well only when computation of the reference stream is cheap compared to the actual loop computation.

Finally, speculative parallelization is also related to optimistic concurrency control and synchronization [10, 14], including hardware-assisted schemes [11, 17, 22]. Under these schemes, which target explicitly parallel code, threads are allowed to speculatively enter critical sections simultaneously or speculatively proceed past a barrier before all threads have reached it. In these, there is no need to enforce a total order on the memory accesses to shared objects, but only that such accesses satisfy some valid partial order (mutual exclusion in the case of critical sections or pre-barrier before post-barrier accesses in the case of barriers). Speculative parallelization schemes, on the other hand, tackle a more general problem that requires enforcing a total order of accesses that satisfies the execution semantics of the original sequential code.

7. CONCLUSIONS

In this paper, we proposed and evaluated a new scheme for software speculative parallelization. After systematically considering the design space we implemented our scheme with carefully tuned data structures, synchronization policies, and scheduling mechanisms. The access information structures are shown to work well on speculative load and store operations as well as commit operations, regardless of the size of the user data structure. The synchronization policies relax the constraints of previous work with software

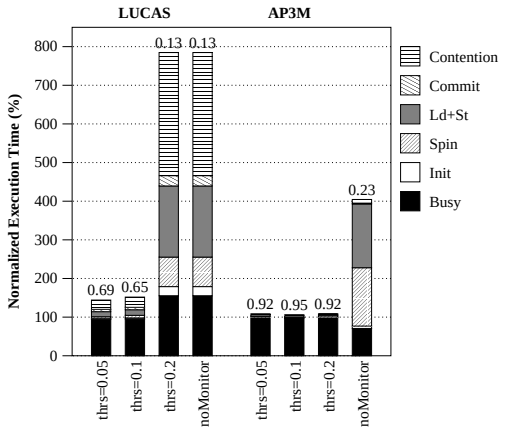

Figure 9: Normalized execution time breakdown for *Baseline* with and without a squash monitor. With a squash monitor thresholds of 0.05 (*thrs=0.05*), 0.1 (*thrs=0.1*), and 0.2 (*thrs=0.2*) were used. Results are shown for 4 processors. The numbers on top of the bars are the speedups relative to sequential execution.

speculative parallelization and allow for faster dependence violation checks on every speculative store operation. We also show how to guarantee proper synchronization in the common case that the hardware and the compiler only enforce relaxed memory consistency models. Finally, the scheduling mechanism, based on an aggressive sliding window, offers increased tolerance to load imbalance and data dependence violations while keeping the memory overhead associated with the access structures tolerable. Experimental results show that our scheme has small overheads and is able to reach a large fraction of the potential parallel execution performance. In particular, for applications with few or no dependence violations, the scheme realizes on average 71% of the speedup of a manually parallelized version of the code. The results also show that the scheme outperforms two recently proposed software-only speculative parallelization schemes: one by 25% on average and with similar memory overheads, and the other by 7% on average but with significantly less memory overhead. For applications with many data dependence violations, our performance monitors and switches can effectively curb the performance degradation.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their valuable suggestions. We also thank Michael O'Boyle, José F. Martínez, and Pedro Trancoso for their helpful comments on earlier drafts of this paper. Finally, we thank the Edinburgh Parallel Computing Center (EPCC) for the main computing resources used in this work and its support staff, in particular Mark Bull and Catherine Inglis.

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial." *IEEE Computer*, Vol. 29, No. 12, pages 66-76, December 1996.

[2] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance." *Wksp. on OpenMP Applications and Tools*, pages 1-10, July 2001.

[3] J. E. Barnes. Institute for Astronomy, University of Hawaii. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, L. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers." *Intl. Journal of Supercomputer Applications*, Vol. 3, No. 3, pages 5-40, Fall 1989.

[5] M. Cintra, J. F. Martínez, and J. Torrellas. "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors." *Intl. Symp. on Computer Architecture*, pages 13-24, June 2000.

[6] M. Cintra and J. Torrellas. "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors." *Intl. Symp. on High Performance Computer Architecture*, pages 43-54, February 2002.

[7] F. Dang, H. Yu, and L. Rauchwerger. "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops." *Intl. Parallel and Distributed Processing Symp.*, pages 20-29, April 2002.

[8] M. Gupta and R. Nim. "Techniques for Run-Time Parallelization of Loops." *Supercomputing*, November 1998.

[9] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 58-69, October 1998.

[10] M. Herlihy. "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types." *ACM Trans. on Database Systems*, Vol. 15, No. 1, pages 96-124, March 1990.

[11] M. Herlihy and J. E. B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures." *Intl. Symp. on Computer Architecture*, pages 289-300, May 1993.

[12] The Hydra Consortium. Department of Physics and Astronomy, McMaster University. http://hydra.mcmaster.ca/hydra/.

[13] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. "Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution." *Symp. on Principles and Practice of Parallel Programming*, pages 2-11, June 2001.

[14] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control." *ACM Trans. on Database Systems*, Vol. 6, No. 2, pages 213-226, June 1981.

[15] S.-T. Leung and J. Zahorjan. "Improving the Performance of Runtime Parallelization." *Symp. on Principles and Practice of Parallel Programming*, pages 83-91, May 1993.

[16] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 365-372, June 1999.

[17] J. F. Martínez and J. Torrellas. "Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 18-29, October 2002.

[18] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann Publishers Inc., San Francisco, second edition, 1994.

[19] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. "Dynamic Speculation and Synchronization of Data Dependences." *Intl. Symp. on Computer Architecture*, pages 181-193, June 1997.

[20] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. "Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor." *Intl. Conf. on Supercomputing*, pages 368-380, June 2001.

[21] S. Papadimitriou and T. Mowry. "Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity." School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-01-145, July 2001.

[22] R. Rajwar and J. R. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution." *Intl. Symp. on Microarchitecture*, pages 294-305, December 2001.

[23] L. Rauchwerger and D. Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization." *Conf. on Programming Language Design and Implementation*, pages 218-232, June 1995.

[24] P. Rundberg and P. Stenström. "Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors." *Wksp. on Scalable Shared Memory Multiprocessors*, June 2000.

[25] J. Saltz, R. Mirchandaney, and K. Crowley. "Run-time Parallelization and Scheduling of Loops." *IEEE Trans. on Computers*, Vol. 40, No. 5, pages 603-611, May 1991.

[26] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture*, pages 414-425, June 1995.

[27] SPARC International Inc. *The SPARC Architecture Manual Version 9.* Prentice Hall PTR, Englewood Cliffs, 2000.

[28] Standard Performance Evaluation Corporation. http://www.spec.org/.

[29] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation." *Intl. Symp. on Computer Architecture*, pages 1-12, June 2000.

[30] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "Improving Value Communication for Thread-Level Speculation." *Intl. Symp. on High-Performance Computer Architecture*, pages 55-66, February 2002.

[31] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Runtime Parallelization in Distributed Shared-Memory Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 161-173, February 1998.