Design Space Exploration of a Software Speculative Parallelization Scheme

Marcelo Cintra, *Member*, *IEEE*, and Diego R. Llanos, *Member*, *IEEE*

Abstract—With speculative parallelization, code sections that cannot be fully analyzed by the compiler are optimistically executed in parallel. Hardware schemes are fast but expensive and require modifications to the processors and/or memory system. Software schemes require no changes to the hardware of existing shared-memory systems, but can suffer from significant overheads involved with the speculative execution. In fact, the performance of software schemes is highly dependent on application characteristics, the design and implementation of the scheme, and the system configuration and size. This paper explores the design space of a recently proposed software speculative parallelization scheme. In the process, we gain insight into the most beneficial features of software schemes for speculative parallelization, as well as the most influential application characteristics. For instance, experimental results show that, contrary to intuition, checking for data dependence violations on every speculative store, as opposed to at commit time, leads to little performance degradation in the worst case and to significantly better performance with large configurations. Also, scheduling policies based on windows can perform very close to fully dynamic policies with a fraction of the memory overhead. Finally, experimental results show consistent speedups in the execution of loops that cannot be parallelized at compile time, both with and without RAW data dependences, for 4 to 32 processors.

Index Terms—Speculative parallelization, thread-level speculation, parallel architectures.

✦

---

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, runtime parallelization in software has been explored under two major approaches: *inspector-executor* ([14], [23]) and *speculative parallelization* ([6], [8], [9], [20], [22]). 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 reexecuted.

While various degrees of hardware support for speculative parallelization on multiprocessors have been proposed in the literature (e.g., [10], [15], [25], [28], [30], [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 is augmented with code to perform all the speculative operations. Software schemes, however, can suffer from significant overheads involved with the speculative operation. In fact, the performance of software schemes is highly dependent on application characteristics, the design and implementation of the scheme, and the system configuration and size.

The contribution of this paper is to present a comprehensive quantitative exploration of the design space of a recently proposed software speculative parallelization scheme [6]. In this way, we gain insight into the design features more likely to impact performance as well as into the application characteristics more likely to affect the performance of software speculative parallelization. A secondary contribution is to provide a proof of the correct placement of memory fences in the scheme of [6] in order to guarantee sequentially consistent execution of the protocol in systems that support some relaxed memory consistency model.

Experimental results show that, contrary to intuition, checking for data dependence violations on every speculative store, as opposed to at commit time, leads to better performance on average, with little performance degradation in the worst case. Also, scheduling policies based on windows can perform very close to fully dynamic policies with a fraction of the memory overhead. With fully dynamic policies, the serialization of the commit can become a serious bottleneck and mitigating techniques, such as partial commits, should be used. Finally, all the configurations evaluated with the speculative parallelization scheme provide good speedups for configurations of 4 to 32 processors, even in the presence of RAW data dependences.

The rest of the paper is organized as follows: Section 2 describes speculative parallelization, highlights the operations involved, and discusses in detail the design options

---

- *M. Cintra is with the School of Informatics, The University of Edinburgh, 2505 James Clerk Maxwell Building, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, United Kingdom. E-mail: mc@inf.ed.ac.uk.*
- *D.R. Llanos is with the Departamento de Informática, Universidad de Valladolid, Edificio Tecnologías de la Información, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: diego@infor.uva.es.*
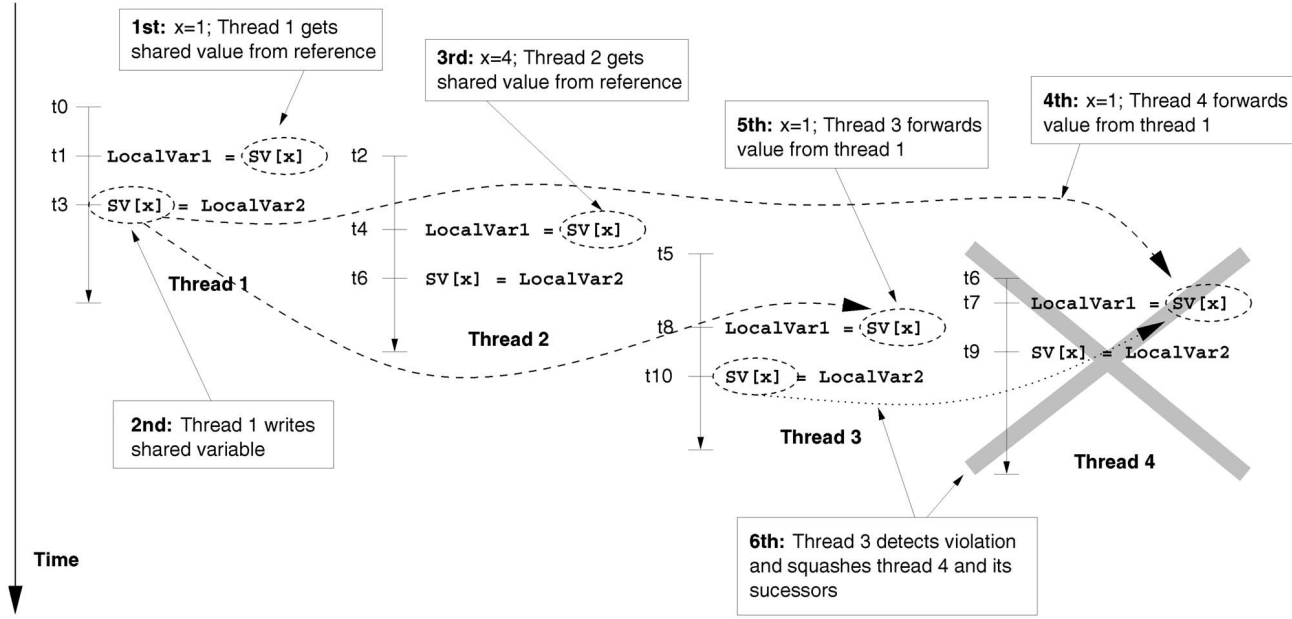
Fig. 1. Example of speculative parallel execution with forwarding and a RAW dependence violation.

available and their expected impact on performance. Section 3 summarizes the speculative parallelization scheme that we proposed. Section 4 describes our evaluation methodology. Section 5 presents the experimental results. Section 6 discusses related work and Section 7 presents some conclusions. Finally, the Appendix provides a formal analysis of the races involved in our protocol.

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 an order on the threads. At any given time during execution, the earliest thread in program order is *nonspeculative* 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 runtime, a violation can be prevented by *forwarding* the value produced by the predecessor thread. *Write-after-write* (WAW) and *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 reexecution, the thread can be then provided with the updated value. Fig. 1 shows how threads can be speculatively executed in parallel.

With software speculative parallelization, the application itself is augmented to perform the above operations. The first step is then 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. Once the speculative data are identified, the compiler must identify all their uses and definitions (the *speculative loads and stores*) and replace them with special operations to maintain access information.

From the above discussion, we can summarize the main operations required by speculative parallelization as follows:

1. schedule speculative threads,
2. maintain access information to speculative data (the *speculative access state*),
3. buffer speculative data and commit it to safe storage when appropriate, and
4. detect data dependence violations, squash, and restart threads as necessary.

In the following sections, we give an overview of these operations and discuss in more detail the main design options available.

2.2 Scheduling Speculative Threads

Traditionally, the basic scheduling options for parallel loops are 1) static or 2) dynamic. With *static scheduling*, the iteration space is partitioned into P chunks of iterations, where P is the number of processors. With dynamic scheduling iterations are dynamically assigned to processors at runtime. The simplest dynamic technique is *self-scheduling* [29], where all the chunks are unit size. While these schemes are simple and effective for parallel loops, both are undesirable under speculative parallelization. Static scheduling will perform poorly when there is load imbalance or when there are data dependence violations. Self-scheduling is not practical when the number of iterations, T, is very large because the memory overhead of the speculative structures is then proportional to the number of iterations. This is because each thread requires its own set of data structures for speculative execution, as explained in Section 2.3.

Under speculative parallelization an alternative design option is to use a 3) *sliding window* mechanism [5], [8]. In these schemes, chunks of iterations are scheduled in windows of size W at a time. 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. Such schemes can better decouple the memory overhead and tolerance to load imbalance and data dependence violations from the number of iterations. This is because C can be made very large without increasing the memory overhead while the sliding of the window approximates the behavior of a dynamic schedule across C.

There are two possible variations of sliding windows. The first variation is to have a conservative mechanism and only slide the window once all threads in the window complete execution and commit [8]. The second scheme is to slide the window every time nonspeculative threads complete execution and commit [5]. Despite their larger complexity and management overhead, we expect schemes based on windows to perform better across a broad range of situations.

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 2.2). 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.

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. Initially, all elements of the speculative data structure are in state *not accessed*. An *exposed load* occurs when a thread issues a load without having previously issued a store to the same data. 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*.
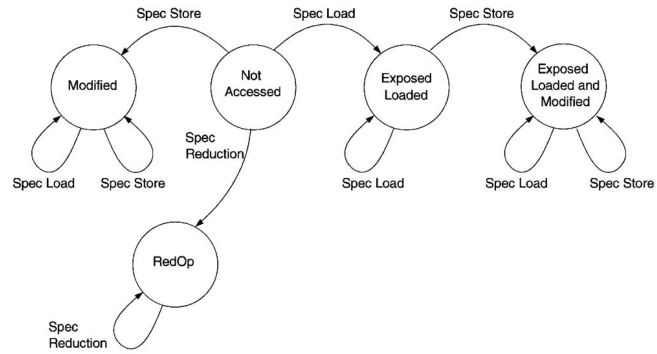

Fig. 2. Transition state diagram for a datum speculatively accessed.

If forwarding is not supported or if no up-to-date predecessor version is found, the last committed value (called the *reference* value) is returned. Such loads can potentially cause RAW data dependence violations. Upon speculative loads and stores, the corresponding entry in the access structure may be update, following the transition state diagram shown in Fig. 2.

While variation of the high-level operations just described is limited, a few design options exist with respect to the implementation of the speculative access structure. These will have a direct impact on the execution overheads of checking for data dependence violations and committing. A simple 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) 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. However, this approach is very inefficient when committing and when checking for data dependence violations on 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 complementing the speculative access structure with 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.

2.4 Committing Safe Data

Speculative modifications to the user data are temporarily stored in the version copies. At some point after becoming safe, version copies must be committed to safe storage, which is usually the user data structure itself. There are two main design options here: 1) to commit all the safe speculative modifications at once at the end of the speculative execution (the *final commit*), or 2) to perform intermediate *partial commits*, that is, committing intermediate versions of the speculative data modified during the execution of the loop. Partial commits can be done at some arbitrary points in time, but are usually done when the nonspeculative thread finishes or, in a window-based system (Section 2.2), when the window moves. Performing intermediate partial commits frees up version storage, offers some opportunity for overlapping the commit overhead with computation by other threads, 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. This includes the need for some form of synchronization to guarantee orderly commit.

In a window-based scheduling design, partial commits are required in order to advance the window and recycle the speculative data structures. Partial commits are not required, but are still possible in a design without window. However, even if partial commits are implemented in a system without windows, the size of the speculative structures must still be proportional to the size of the iteration space to accommodate worst-case scenarios.

2.5 Checking for Data Dependence Violations and Squashing Threads

Data dependence violations are detected by looking at the speculative access structures: A RAW data dependence violation has occurred whenever a thread has a datum in state *exposed loaded* or *exposed loaded and modified*, while some predecessor thread has the same datum in state *modified* or *exposed loaded and modified*. Checking for violations regularly can incur execution overheads but prevents processors from performing much useless work in the presence of violations. In general, detecting data dependence violations can be done: 1) on every speculative load and store, 2) when threads commit, or 3) simply at the end of the speculative section.

Checking for data dependence violations on every speculative memory operation requires some sort of synchronization to avoid races. However, this policy has two major advantages. First, it allows for early detection of violations, which can be exploited to eagerly squash and restart threads. Second, in this case, we must only check for violations on a particular element of the user speculative data. Moreover, checks for violations can be overlapped with computation by other threads. On the other hand, checking for data dependence violations on commits or at the end of the speculative section requires checking for violations on at least all the elements exposed loaded by the threads involved, and potentially on all elements of the user speculative data, depending on the implementation of the access structure. The cost of this operation is further increased by the fact that commits are serialized.

3 COST-EFFECTIVE SOFTWARE SPECULATIVE PARALLELIZATION

In this section, we briefly review the design of our scheme for software speculative parallelization [6]. The purpose of this description is twofold. First, it makes the discussions from the previous sections more concrete. Second, by using this implementation as the common baseline for our experiments, we isolate the high-level design alternatives discussed in Section 2 from effects of implementation details.

3.1 Sliding Window

Our scheme implements in software a sliding window mechanism similar to the hardware mechanism of [5]. 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 nonspeculative and the most-speculative threads (Fig. 3).

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 nonspeculative 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

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 $M \times W$ 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 Fig. 3). 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 $M \times W$ 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 Fig. 3). 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 (bottom part of Fig. 3). 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 datum, while the Safe state indicates that no thread has ever performed an exposed load to this datum. 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 Fig. 3).

3.3 Speculative Memory Operations

Figs. 4a and 4b show abridged implementations of our speculative load and speculative store operations, respectively, in a C-like syntax. From these figures, we can
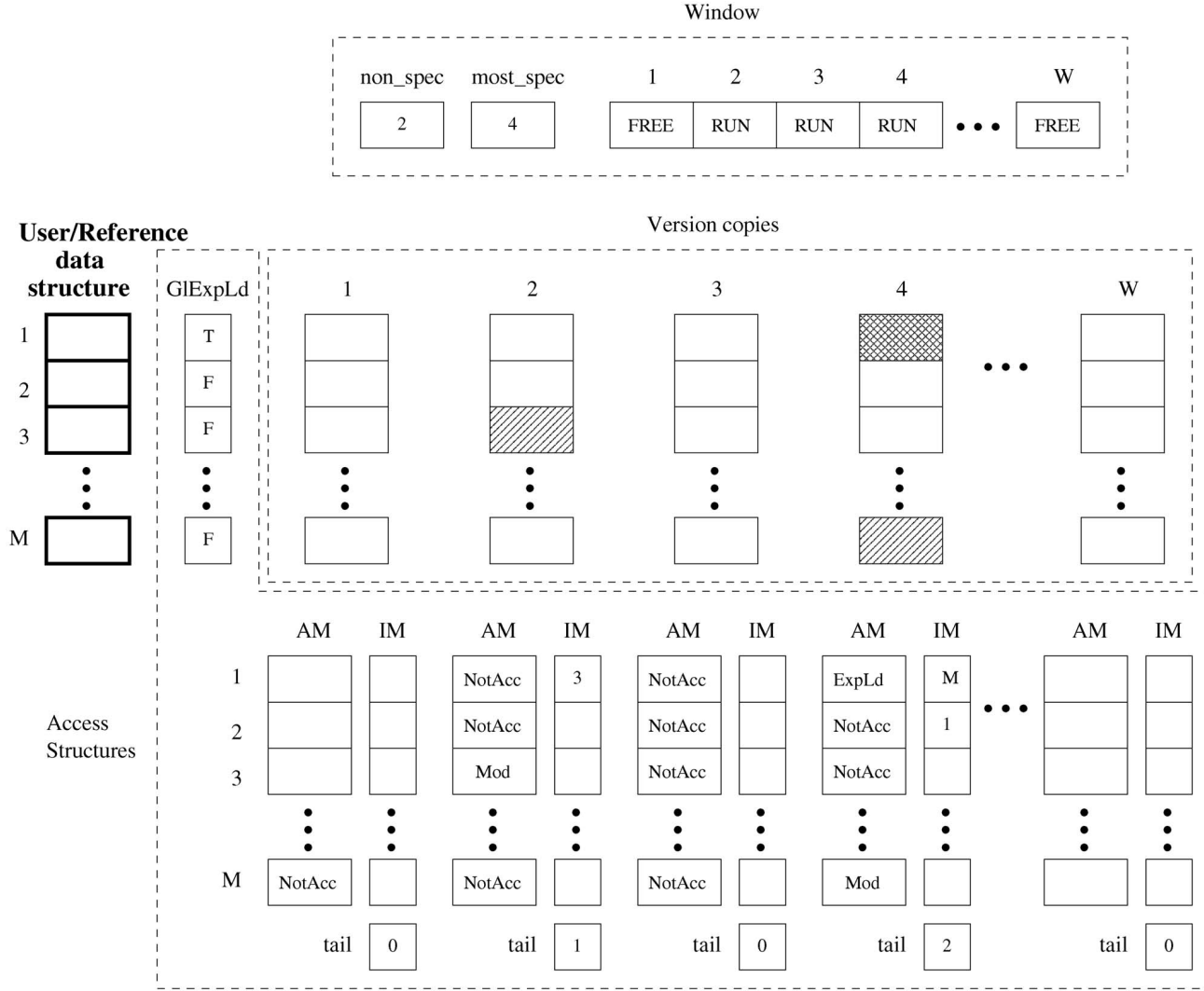
Fig. 3. Data structures used in our software speculative parallelization scheme. In the version copies, the single-striped boxes correspond to `Mod` data and the double-striped boxes correspond to `ExpLd` data.

highlight the following features of our scheme: Only the first load to a datum requires special handling by the protocol (line 1 in Fig. 4a). 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 Fig. 4a). Similarly, the search for data dependence violations on stores only requires looking up one element of AM per thread (lines 11 to 15 in Fig. 4b). 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 Fig. 4b). 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 Fig. 4b).

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 most-speculative pointer backward. Later, when the squashed thread finishes, it will trigger the reexecution of the offending threads. Additionally, to increase the chances of performing squashes as early

as possible, threads check their window state before every speculative memory operation (code not shown) and immediately terminate the execution if they find themselves to have been squashed. In this way, the global squash operation may start earlier and threads can be rescheduled earlier.

In addition to speculative loads and stores, we also implement a *speculative reduction* operation, shown in Fig. 4c. This operation simply accumulates in the version copy the contribution of the current thread to the global reduction operation, according to the semantics of each particular reduction operation. At commit time, the value is merged appropriately with the main value stored in the user data structure (code not shown). This speculative reduction operation is necessary to guarantee correct execution in the presence of squashes. Note that user data identified for speculative reductions cannot be operated on by speculative loads and stores, and vice-versa, and if this occurs the error() operation is called.

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

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

```
1.  if (AM[I][tid] == NotAcc) {
2.      version[I][tid]=rvalue;
3.      AM[I][tid]=RedOp;
4.      IM[++tail[tid]][tid]=I;
5.  }
6.  else
7.      if (AM[I][tid] == RedOp)
8.          version[I][tid] = red_op(version[I][tid],rvalue);
9.      else error();
```
(c)

```
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++)
11.                 ref[IM[k][j]]=
                        version[IM[k][j]][j];
12.     #pragma memory fence
13.             window[j]=FREE
14.         }
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++)
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;
29.  most_spec++;
30.  #pragma end critical
```
(d)

Fig. 4. Abridged C-like code for (a) speculative loads, (b) speculative stores, (c) a speculative reduction, and (d) the code executed at the end of each thread's execution. 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. `red_op()` corresponds to a reduction operation. The `memory fence` and `critical` directives are discussed in Section 3.5 and the Appendix.

3.4 Commits

Fig. 4d 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 nonspeculative 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. Note that speculative reductions operate only on exclusive data and metadata, so there are no races involving these operations.

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 reorder the operations in the protocol and/or when the hardware only enforces some relaxed memory consistency model that allows loads and stores to bypass each other, as is the case in Sun's SPARC [26] and IBM's PowerPC systems [17].

Previous work with software speculative parallelization has dealt with these races by serializing these operations through locks and/or critical sections. This leads to unnecessary additional synchronization overheads and reduces the amount of overlapped execution of these operations. In our system, we carefully order the individual actions in these operations to avoid races under a sequential consistency model and we add a small set of memory directives to guarantee this order under relaxed memory consistency models. This allows us to use a single critical section to guarantee exclusive access on commit and squash operations (Fig. 4d). In the Appendix, we show that our fences are the minimal set to guarantee that the protocol execution appears as sequentially consistent even if the system uses some relaxed consistency model.

4 EVALUATION METHODOLOGY

4.1 Applications

To evaluate our scheme, we choose four applications: *TREE* from [3], *WUPWISE* from SPECfp2000 [27], *MDG* from the PERFECT Club suite [4], and *2D-HULL* from [7]. These applications are representative of legacy as well as recent sequential scientific programs. They spend a large fraction of their sequential execution time on loops that cannot be automatically parallelized by state-of-the-art compilers because they have dependence structures that are either too complicated to be analyzed at compile time or dependent on the input data. The loop in *TREE* takes up to 94 percent of sequential execution time, the loops in *WUPWISE* take up to 41 percent, the loop in *MDG* takes up to 86 percent, and the loop in *2D-HULL* takes up to 99 percent. The first three applications do not suffer from RAW data dependences at run time. The fourth application, a two-dimensional Convex

TABLE 1
Characteristics of the Applications and Input Sizes Used

Application (loop)	Size	Description	Spec data size in KB	Iterations per invocation	% of violations	Times spec. section is executed
TREE (`accel_10`)	Small	Off-axis parab. collision, 1024 part.	< 1	1024	0	768
	Medium	Off-axis parab. collision, 2048 part.	< 1	2048	0	768
	Large	Off-axis parab. collision, 4096 part.	< 1	4096	0	768
WUPWISE (`muldeo_200'`, `muldoe_200'`)	Small	test input	12,000	8,000	0	8
	Medium	train input	12,000	8,000	0	44
	Large	reference input	12,000	8,000	0	304
MDG (`interf_1000'`)		reference input	13	343	0	101
2D-HULL (main loop)	Small	Square input set, 10M particles	13	9,999,997	2.00	1
	Medium	Square input set, 20M particles	14	19,999,997	1.12	1
	Large	Square input set, 40M particles	15	39,999,997	0.64	1
2D-HULL (main loop)	Small	Disc input set, 10M particles	86	9,999,997	15.48	1
	Medium	Disc input set, 20M particles	109	19,999,997	10.63	1
	Large	Disc input set, 40M particles	137	39,999,997	7.35	1

Hull solver, uses a randomized incremental algorithm, which generates RAW data dependences every time a new point causes the current convex hull solution to be expanded.

Table 1 shows, for each application, the loops that we attempt to parallelize speculatively, the different input sizes used, the size of the data accessed through speculative references, the average number of iterations executed per loop invocation, the fraction of threads that suffer data dependence violations, and the number of times the loops are invoked. 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]

The input sets used with *TREE*, *WUPWISE*, and *MDG* are the standard ones provided with the applications, unless stated otherwise. For *2D-HULL*, we evaluate six different input sets that distribute the points inside a square or inside a disc and contain different numbers of input points. The shape of the distribution affects the expected number of points that fall outside the incremental solution, thus affecting the number of data dependences.

4.2   Parallel Execution Environment

We have run the applications described in Section 4.1 on a 48-processor Sun Fire 15K symmetric multiprocessor (SMP). The machine is equipped with 900MHz UltraSPARC-III processors, each with a private 64KByte 4-way set-associative L1 cache, a private 8MByte direct-mapped L2 cache, and 48 GBytes of shared main memory. The system runs SunOS 5.8. The system interconnect has a sustained bandwidth of 9.6 GBytes/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) [26]. 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 correctly executes in this model is guaranteed to correctly execute in the other two.

The applications were compiled with the Forte Developer 7 Fortran 95 compiler using the highest optimization settings for our execution environment: `-O3 -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 the Forte development environment. 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 implemented using OpenMP `flush` directives. The semantics of this directive enforce that the local 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 could use the more selective `MEMBAR` fence provided in the SPARC processor [26].

4.3   Systems Evaluated

The goal of our experiments is to quantitatively evaluate the discussed design trade offs of software speculative parallelization schemes. The schemes that we evaluate are the following.

sys1 uses a window scheme with partial commits when the nonspeculative 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 is 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,

---

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.

TABLE 2
Characteristics of the Systems Evaluated

System	Window type	Checks for data dependence violations	Partial commits
sys1	Aggressive	On spec. stores	Mandatory
sys2	Aggressive	On commits	Mandatory
sys3	Conservative	On spec. stores	Mandatory
sys4	Conservative	On commits	Mandatory
sys5	No window	On spec. stores	No
sys6	No window	On commits	No
sys7	No window	On spec. stores	Yes
sys8	No window	On commits	Yes

and supports forwarding. We evaluate this system to compare our aggressive sliding window with a more conservative one.

sys4 is a variation of *sys3* that only checks for data dependence violations when threads commit and does not support forwarding. While the implementations are different, this system is similar in concept to the SW-R-LRPD scheme of [8] with respect to the high-level design options discussed in Section 2.

sys5 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. While the implementations are different, this system is similar in concept to the scheme of [22] with respect to the high-level design options discussed in Section 2, except that we do not use locks for speculative loads and stores.[2]

To show all the possible combinations of design options, we also study three other systems: *sys6* is a variation of *sys5* that only checks for data dependence violations at the end of the speculative execution, *sys7* is a variation of *sys5* that has partial commits, and *sys8* is a variation of *sys6* that has partial commits. Table 2 summarizes the characteristics of the systems evaluated and their design decisions.

5 EXPERIMENTAL RESULTS

5.1 Baseline Speedups

We start by presenting the execution time breakdowns and speedups of the speculative sections only for all the systems described in Section 4.3 for configurations of 4 to 24 processors. For the window-based systems, the window size is set equal to the number of processors (minimal size). Fig. 5 shows the results. 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* is the execution time of the original loop body plus OpenMP overhead; *Spin* is the idle time due to load imbalance when waiting for other threads to complete in order to advance the window (e.g., line 22 in Fig. 4d) plus idle time due to load imbalance at the end of the speculative section; *Memory ops.* is the overhead time spent on

---

2. In [22], 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.

speculative loads, stores, and reductions, excluding the original memory operation (e.g., all of Fig. 4a except line 8 or line 13, all of Fig. 4b except line 1, and all of Fig. 4c except lines 2 and 8); *Commit* is the overhead time of the commit operations and setting up of a new thread (e.g., lines 2 to 19 and 28 to 29 in Fig. 4d) plus initialization time of the speculative access structures at the beginning of the speculative sections and, for the window based systems, when these structures are reassigned to new threads (e.g., lines 24 to 27 in Fig. 4d); and *Contention* is the idle time waiting at the locks and barriers required by the different schemes.

From the figure, we can draw the following general observations:

- It was not possible to run *WUPWISE* or *2D-HULL* on our Sun platform with the systems without window-based scheduling (*sys5* to *sys8*) because of the large memory overhead (Table 1). Thus, dynamic systems are not suitable when the user data structure and/or the iteration space are very large.
- Despite its overheads, fairly good speedups can be achieved with software speculative parallelization for medium configurations (8 to 16 processors). However, efficiency drops significantly with larger configurations due mainly to contention, which can hinder the scalability of these software schemes. Another reason for the scalability problems is the spin-wait in the window-based systems (*sys1* to *sys4*), an effect that can be alleviated with larger window sizes, as shown in Section 5.2. Finally, the conservative window and fully dynamic schemes have more smooth performance degradation than the aggressive window systems (e.g., *sys3* and *sys5* versus *sys1*).
- Checking for data dependence violations on every speculative store (*sys1*, *sys3*, and *sys5*) can sometimes degrade performance compared to the corresponding systems that only check for violations at commit (*sys2*, *sys4*, and *sys6*) for smaller configurations, but leads to better performance for larger configurations with *TREE*, *MDG*, and *2D-HULL*. This is due to the added contention due to the longer commit operation. With *WUPWISE*, checking for violations on speculative stores consistently leads to better performance. The reason for this is the much larger size of the user data structure in this application. Also, in the presence of data dependence violations, checking for violations on every speculative store and eagerly squashing threads can reduce the amount of useless work performed by squashed threads, as evidenced by the difference in *Busy* time between *sys1*, *sys2*, and *sys3* and *sys4* for *2D-HULL* with 7.35 percent and 10.63 percent violations.
- The more aggressive window-based schemes (*sys1* and *sys2*) consistently outperform the less aggressive ones (*sys3* and *sys4*) due to a large reduction in load imbalance overhead. The schemes with fully dynamic scheduling (*sys5* and *sys6*) perform as well as, or better than, the window-based schemes with such a minimal window size for *TREE*, but not for *MDG*. This is because of the slightly larger speculative data set size of *MDG*, which increases the commit time and its associated contention overhead.
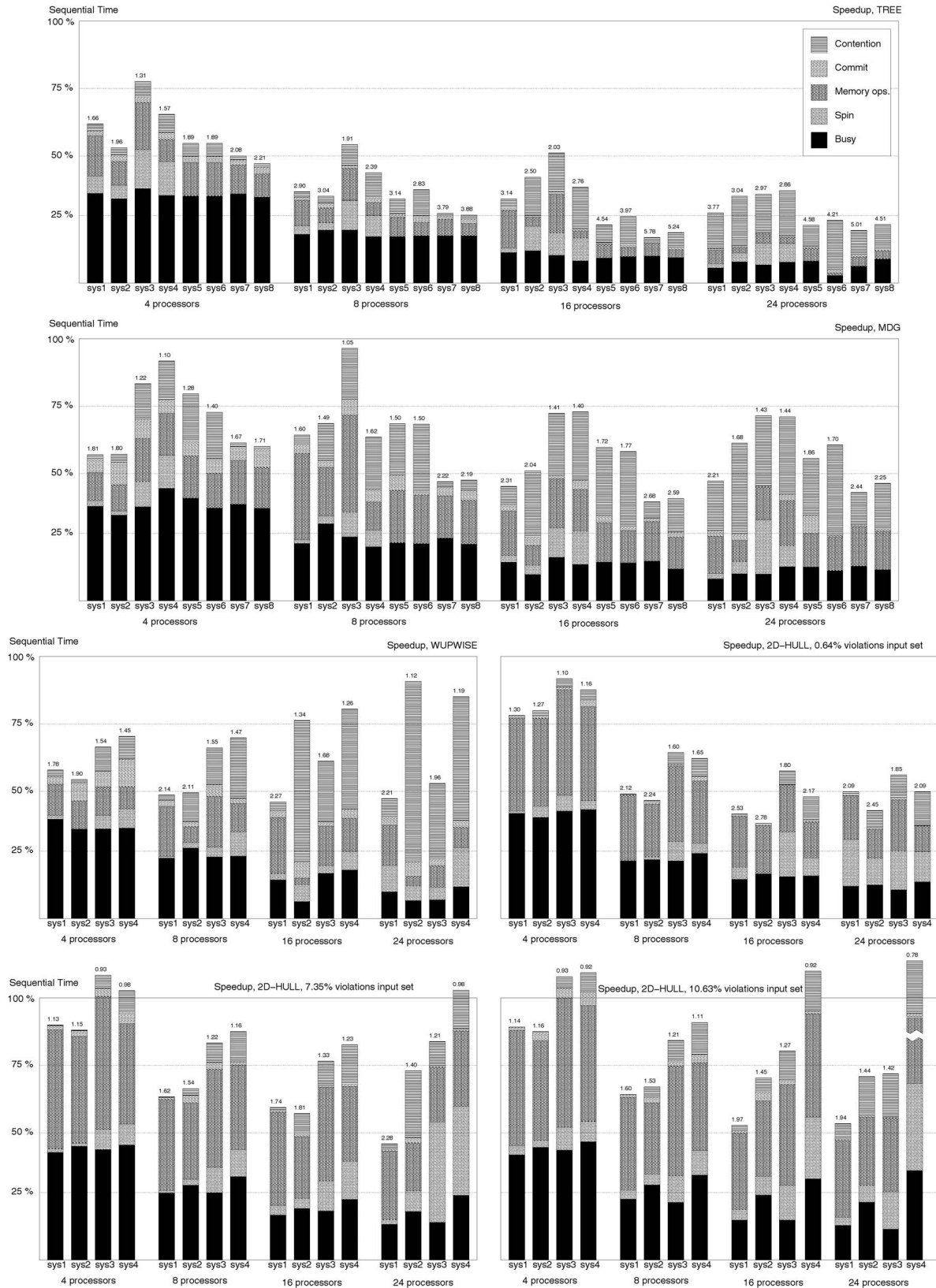
Fig. 5. Speedups and execution time breakdowns for the speculative sections only. Window sizes (where applicable) are equal to 1x the number of processors.

- Partial commits can significantly reduce the commit overheads of dynamic systems, as can be seen by comparing *sys5* and *sys6* with their counterparts with partial commits, *sys7* and *sys8*. In fact, the latter systems consistently outperform all other systems for *TREE* and *MDG*.

A detailed comparison of the performance of these schemes against hardware-based schemes is difficult because the applications used do not usually overlap and hardware-based papers tend to report numbers for the speculative parallelization of the whole applications. Nevertheless, we were able to find reported numbers for a very optimized hardware scheme for *TREE* in [19]: approximately 7-fold and 14-fold speedups for 8 and 16 processors, respectively. Such performance numbers are clearly better than those achieved by software schemes, but come at the cost of additional dedicated hardware.

5.2 Scheduling Policy and Window Size

To evaluate the effects of the policy for scheduling threads, we compare the schemes that vary in their scheduling policies: *sys1*, *sys3*, *sys5*, and *sys7*. For the window-based systems, we vary the window size to evaluate its impact on the load imbalance overhead, commit overhead, and execution times. Fig. 6 shows the execution time breakdowns and speedups of the speculative sections only with different window sizes.

From the figure, we can draw the following observations:

- Increasing the window size from one time to twice the number of processors for both *sys1* and *sys3* improves the performance of these schemes significantly (compare with Fig. 5), with only a few exceptions. For *sys1*, further increases in window size lead to only small incremental performance gains, as a window size of twice the number of processors already practically eliminates all load imbalance overhead. The less aggressive window scheme, *sys3*, however, can still benefit from larger window sizes, especially for smaller configurations of 4 or 8 processors. The exception to this occurs with *2D-HULL* running on larger number of processors, where the large window size leads to poorer performance for *sys3*. This is mainly due to the increase in the time taken to scan predecessor and successor access state information at the speculative memory operations. This effect is not so noticeable for *sys1* as the dynamic commits mean that the average window size is smaller than the nominal size.
- Within the small range of values evaluated, the window size has very small impact on the commit overhead.
- With small window sizes of two to four times the number of processors, *sys1* performs very close to the systems with dynamic scheduling for *TREE* and *MDG* (recall that it was not possible to run the dynamic scheduling systems for *WUPWISE* and *2D-HULL*).

In Section 5.1 and Fig. 5, we observed that checking for violations on memory operations does not lead to significant overheads and, in fact, can lead to better performance for larger system configurations. We have evaluated the impact of the window size on the relative performance of the different policies for checking for violations. We do not show results due to space limitations, but our experiments show that the relative performance variation between *sys1* and *sys3* and their counterparts with less aggressive checks for data dependence violations, *sys2* and *sys4*, does not change noticeably with the window size.

5.3 Effects of Problem Size

To study the effect of problem size in the performance of our baseline system (*sys1*), we compared the speedups obtained in the execution of the applications with different problem sizes. The problem sizes used lead to changes to the iteration space, the number of times the speculative loop is executed, and the amount of speculative data. Fig. 7 shows the execution time breakdowns and speedups of the speculative sections only for the different problem sizes. With *TREE* and *WUPWISE*, we set the window size to twice the number of processors, while with *2D-HULL* we use a minimal window size equal to the number of processors, as the window size had negligible impact for this application.

From the figure, we can draw the following observations:

- Larger problem sizes lead to slightly better performance. For *TREE* and *WUPWISE*, this is mainly due to better workload division, better locality of data, and reduced contention. For *2D-HULL*, this is also due to a reduction is the relative number of violations and squashes.
- Our system produces good speedups even with problem sizes that can be considered unrealistically small for these applications, and even when data dependence violations are relatively frequent.

6 RELATED WORK

Runtime speculative parallelization in software was introduced in the LRPD test [20]. Data dependence violations are checked at the end of the tentative parallel execution, and the loop is reexecuted sequentially if a violation is detected. Thus, this scheme can only handle fully parallel loops. The scheme in [9] proposed a series of runtime 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, [8] 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 [22] 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 nonscalable byte-vector implementation of the access structures are used to avoid races in the protocol. The work in [18] takes a different approach to software speculative parallelization by placing most of the operations in the software distributed coherence engine. In [6], we proposed our scheme with an aggressive sliding window, with checks for data dependence violations on speculative stores with reduced synchronization constraints, and with fine-tuned data structures. This current work performs a more complete evaluation of that scheme and quantitatively explores a larger section of the design space.

Several hardware approaches for speculative parallelization have been proposed (e.g., [10], [25], [31]). While these alleviate many of the overheads of speculative parallelization
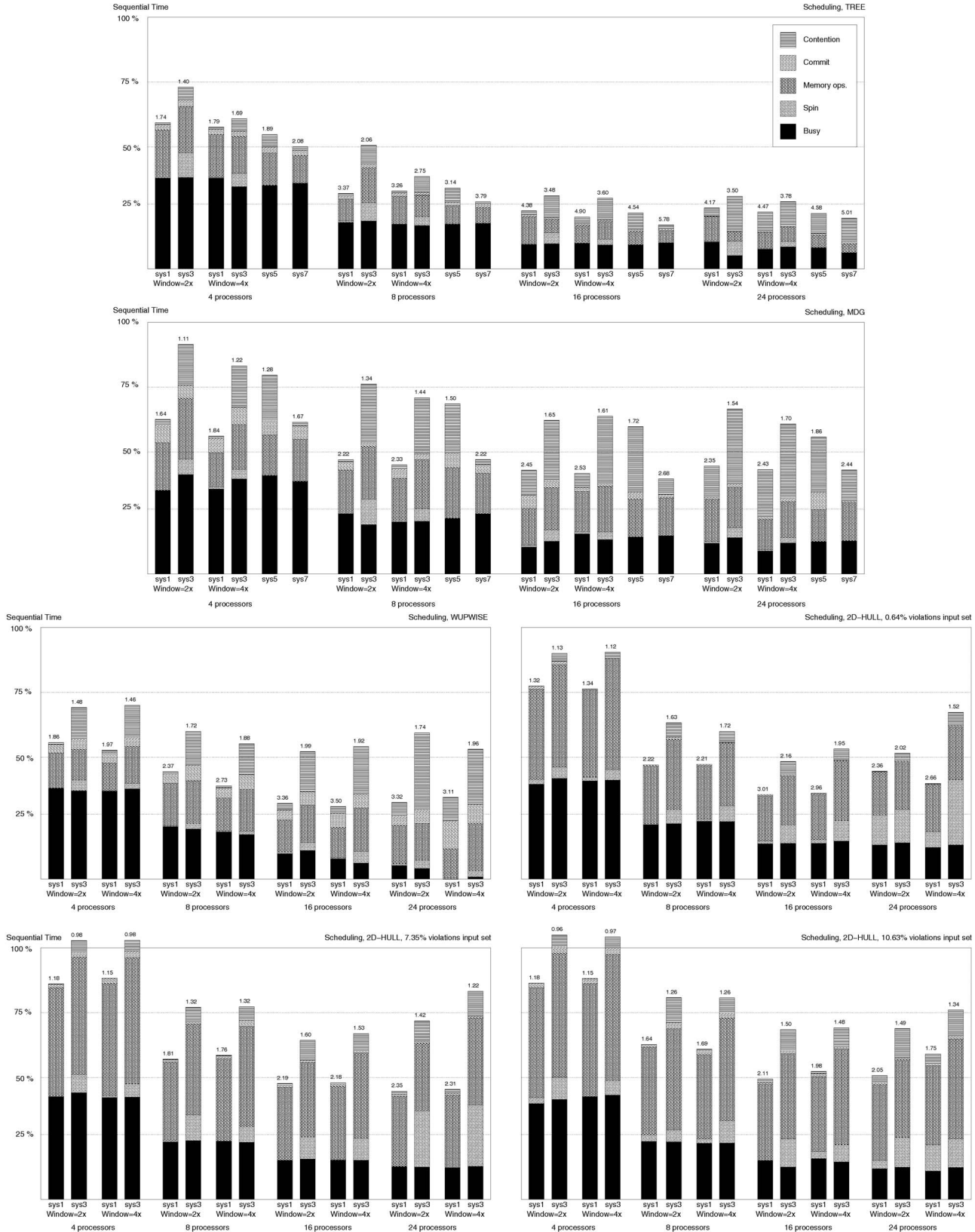
Fig. 6. Behavior of window-based approaches with respect to no-window solutions.

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 ([14], [23]) precompute the reference stream and use the dependence information to execute the
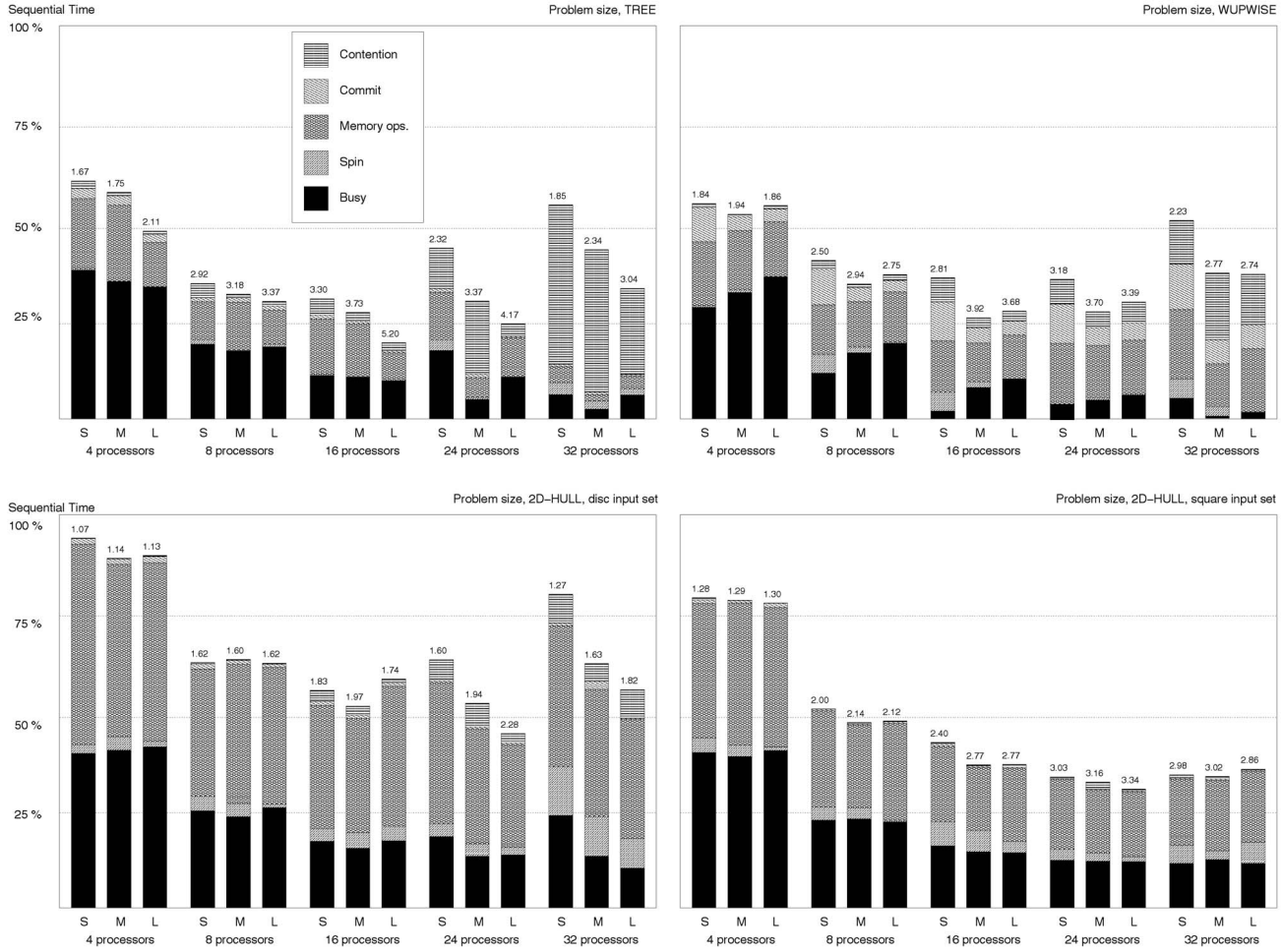
Fig. 7. Effects of different problem sizes with *sys1*. S, M, and L correspond to the Small, Medium, and Large problem sizes listed in Table 1, respectively.

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 ([11], [13]), including hardware-assisted schemes ([12], [16], [21]). 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 prebarrier before postbarrier 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 discussed in detail the design options available for a recently proposed software-only speculative parallelization scheme, and quantitatively evaluated the performance implications of these options. We also correlated the design choices and their performance to application characteristics and system size (i.e., number of processors). From the experimental results, we conclude the following:

- Checking for data dependence violations on every speculative store, as opposed to at commit time, leads to little performance degradation in the worst case and to significantly better performance with large configurations.
- Scheduling policies based on windows can perform very close to fully dynamic policies with a fraction of the memory overhead.
- The serialization of commits can become a significant bottleneck for fully dynamic systems when applications have nontrivial speculative data set sizes. Mitigating techniques such as partial commits and parallel commits (the latter is not evaluated in this paper) can reduce this overhead significantly.
- Our baseline scheme delivers speedups even in the case of an application with RAW data dependences where up to 15 percent of the threads suffer data dependence violations.
- The speculative execution overheads were not significantly affected by changes in the problem
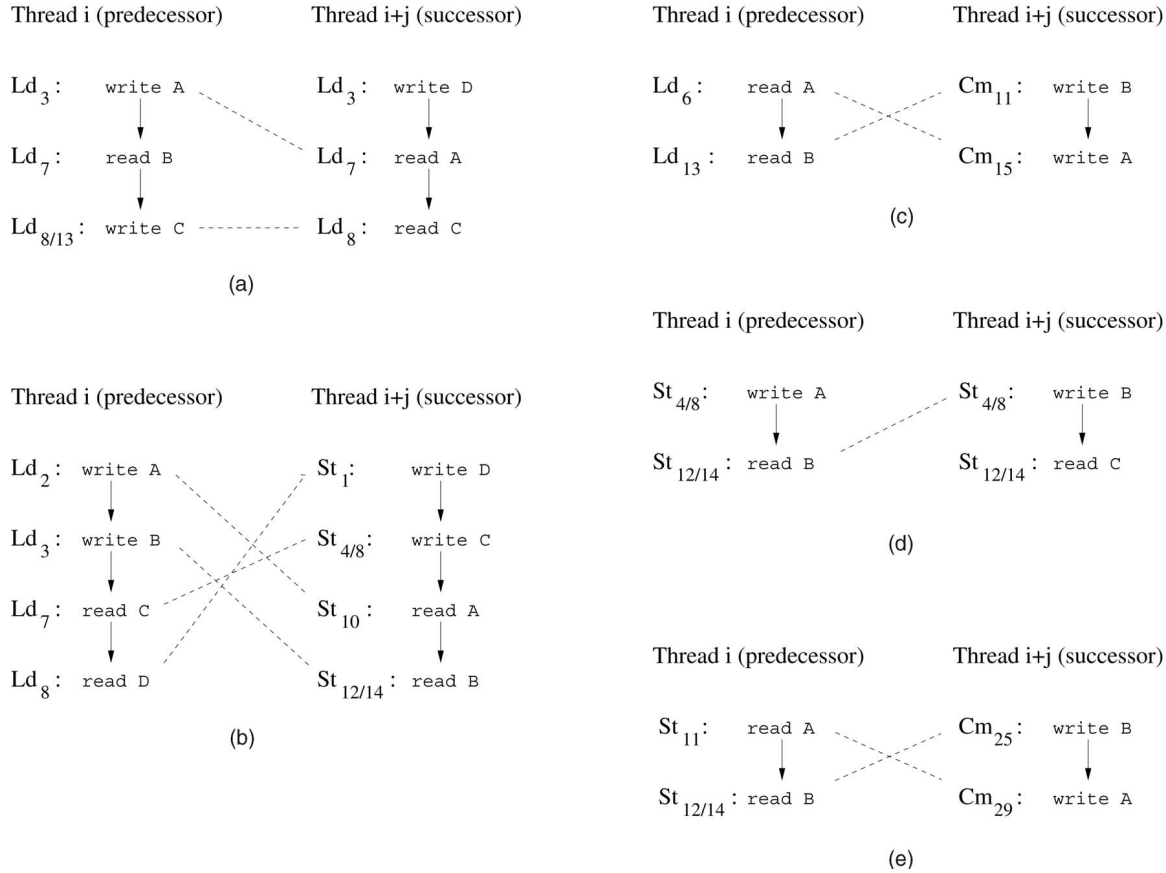
Fig. 8. Conflicting accesses: (a) between two speculative loads, (b) between speculative loads and stores, (c) between speculative loads and commits, (d) between two speculative stores, and (e) between speculative stores and commits. As in [24], we represent program order dependences as solid arrows and memory access conflicts as dashed lines.

size, even for unrealistically small problem sizes, which were expected to stress the relative impact of these overheads.

- Despite drops in efficiency for larger system configurations, most of the design choices evaluated for our software speculative parallelization scheme provide good speedups.
- While these speedups are lower than those obtained with hardware-based schemes, they come at no extra hardware costs.

Overall, we conclude that with an efficient software speculative parallelization scheme such as the baseline system we propose and some of its variants evaluated here, speculative parallelization is an attractive technique to parallelize irregular loops that cannot be parallelized at compile time.

APPENDIX

PROTOCOL RACES

In [6], we appealed to intuition to identify all the races in our protocol and to show that the set of memory fences we proposed is appropriate and minimal to guarantee sequentially consistent execution of the protocol. Here, we use the more formal *delay set analysis* of [24] to prove these assertions. The first step in the delay set analysis is to consider pairwise the routines that could be executed simultaneously by any two threads. In our case we, must consider the following pairs of threads:

1. Speculative Load versus Speculative Load,
2. Speculative Load versus Speculative Store,
3. Speculative Load versus Commit,
4. Speculative Store versus Speculative Store, and
5. Speculative Store versus Commit.[3]

The next step is, for each pair of routines, to compute the *conflict relation (C)*, which is, informally, the set of pairs of conflicting memory operations.[4]

Fig. 8 shows from the codes in Fig. 4 only the memory accesses that can potentially conflict. For instance, "$Ld_2 :$ write A" and "$St_{10} :$ read A" in Fig. 8b refer to the store and load of the same element of GlExpLd in line 2 of the speculative load operation (Fig. 4a) and in line 10 of the speculative store operation (Fig. 4b), respectively. Obviously, if the speculative load and store operations are to different data, then they will access different elements of GlExpLd and there will be no conflict. Similarly, "$Ld_3 :$ write A" and "$Ld_7 :$ read A" in Fig. 8a refer to the

---

3. The squash() operation in line 15 of Fig. 4b updates the most-speculative pointer, which is also accessed by the speculative store operation in line 11 of this figure. This, however, can only cause the store to search for data dependence violations on already squashed threads, and we do not consider this conflict further.

4. Two memory operations conflict if they access the same memory location and at least one is a write.

store and load of the same element of AM in lines 3 and 7 of the speculative load operation (Fig. 4a). Note that for this conflict to occur, the thread performing the store to AM must be less speculative than the thread performing the load to AM.

After computing C, the next step is to compute all the cycles in the union of C with the *program order relation (P)*, which is, informally, the set of relations on memory operations required by program order inside threads. *Cycles* in the union of C and P correspond to data races in the program. Finally, from the cycles of $P \bigcup C$, we select those that are *critical cycles*[5] and pairs of operations from the same thread that occur in some critical cycle are considered *critical pairs*. The set of critical pairs then corresponds to the minimal set of orderings that must be explicitly enforced and is thus called the *delay set (D)*:

- **Speculative Load versus Speculative Load:** Fig. 8a shows the potentially conflicting memory accesses between concurrent speculative load routines. The resulting conflict relation is: $C = \{(Ld_3, Ld_7), (Ld_{8/13}, Ld_8)\}$. Note that line 8 contains both a load and a store, which can conflict. There are no critical cycles and, thus, there are no races and there is no need to insert fences.

- **Speculative Load versus Speculative Store:** Fig. 8b shows the potentially conflicting memory accesses between concurrent speculative load and speculative store routines. The resulting conflict relation is: $C = \{(Ld_2, St_{10}), (Ld_3, St_{12/14}), (Ld_7, St_{4/8}), (Ld_8, St_1)\}$. The critical cycles in $P \bigcup C$ are: $(Ld_2, Ld_8, St_1, St_{10}, Ld_2)$, $(Ld_2, Ld_7, St_{4/8}, St_{10}, Ld_2)$, $(Ld_3, Ld_7, St_{4/8}, St_{12/14}, Ld_3)$, and $(Ld_7, Ld_8, St_1, St_{4/8}, Ld_7)$, which lead to the following critical pairs: $D = \{(Ld_2, Ld_8), (Ld_2, Ld_7), (Ld_3, Ld_7), (Ld_7, Ld_8), (St_1, St_{10}), (St_{4/8}, St_{10}), (St_{4/8}, St_{12/14}), (St_1, St_{4/8})\}$.

  To enforce the ordering imposed by the relation D, it suffices to add full memory fences after Ld_3, Ld_7, St_1, and $St_{4/8}$. However, as the ordering of Ld_7 and Ld_8 is enforced by a control dependence it is safe to remove the memory fence after Ld_7.

- **Speculative Load versus Commit:** Fig. 8c shows the potentially conflicting memory accesses between concurrent speculative load and commit routines. The resulting conflict relation is: $C = \{(Ld_6, Cm_{15}), (Ld_{13}, Cm_{11})\}$. The critical cycle in $P \bigcup C$ is: $(Ld_6, Ld_{13}, Cm_{11}, Cm_{15}, Ld_6)$, which leads to the following critical pairs: $D = \{(Ld_6, Ld_{13}), (Cm_{11}, Cm_{15})\}$.

  To enforce the ordering imposed by the relation D, it suffices to add full memory fences after Ld_6 and Cm_{11}. However, as the ordering of Ld_6 and Ld_{13} is enforced by a control dependence, it is safe to remove the memory fence after Ld_6.

- **Speculative Store versus Speculative Store:** Fig. 8d shows the potentially conflicting memory accesses between concurrent speculative stores. The resulting conflict relation is: $C = \{(St_{4/8}, St_{12/14})\}$. There are no critical cycles and, thus, there are no races and there is no need to insert fences.

- **Speculative Store versus Commit:** Fig. 8e shows the potentially conflicting memory accesses between concurrent speculative store and commit routines. The resulting conflict relation is: $C = \{(St_{11}, Cm_{29}), (St_{12/14}, Cm_25)\}$. The critical cycle in $P \bigcup C$ is: $(St_{11}, St_{12/14}, Cm_{25}, Cm_{29}, St_{11})$, which leads to the following critical pairs: $D = \{(St_{11}, St_{12/14}), (Cm_{25}, Cm_{29})\}$.

  To enforce the ordering imposed by the relation D, it suffices to add full memory fences after St_{11} and Cm_{25}. However, as the ordering of St_{11} and $St_{12/14}$ is enforced by a control dependence it is safe to remove the memory fence after St_{11}.

This analysis shows that the set of fences described in Section 3.5 and shown in Fig. 4 is enough to guarantee a sequentially consistent execution of the protocol in systems that support some relaxed memory consistency model.

ACKNOWLEDGMENTS

REFERENCES

[1] S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer,* vol. 29, no. 12, pp. 66-76, Dec. 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," *Proc. Workshop OpenMP Applications and Tools,* pp. 1-10, July 2001.

[3] J.E. Barnes, Inst. for Astronomy, Univ. of Hawaii, ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode/, 2004.

[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," *Int'l J. Supercomputer Applications,* vol. 3, no. 3, pp. 5-40, Fall 1989.

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

[6] M. Cintra and D.R. Llanos, "Toward Efficient and Robust Software Speculative Parallelization in Multiprocessors," *Proc. Symp. Principles and Practice of Parallel Programming,* pp. 13-24, June 2003.

[7] M. Cintra, D.R. Llanos, and B. Palop, "Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm," *Proc. Int'l Workshop Computational Geometry and Applications,* pp. 188-197, May 2004.

[8] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops," *Proc. Int'l Parallel and Distributed Processing Symp.,* pp. 20-29, Apr. 2002.

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

[10] L. Hammond, M. Wiley, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 58-69, Oct. 1998.

---

5. A cycle is critical if it is a simple cycle of $P \bigcup C$ and has no chords in P.

[11] M. Herlihy, "Apologizing versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Trans. Database Systems,* vol. 15, no. 1, pp. 96-124, Mar. 1990.

[12] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. Int'l Symp. Computer Architecture,* pp. 289-300, May 1993.

[13] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems,* vol. 6, no. 2, pp. 213-226, June 1981.

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

[15] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors," *Proc. Int'l Conf. Supercomputing,* pp. 365-372, June 1999.

[16] J.F. Martínez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 18-29, Oct. 2002.

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

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

[19] M. Prvulovic, M.J. Garzaran, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. Int'l Symp. Computer Architecture,* pp. 204-215, June 2001.

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

[21] R. Rajwar and J.R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proc. Int'l Symp. Microarchitecture,* pp. 294-305, Dec. 2001.

[22] P. Rundberg and P. Stenström, "An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors," *J. Instruction-Level Parallelism,* vol. 3, Oct. 2001.

[23] J. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Trans. Computers,* vol. 40, no. 5, pp. 603-611, May 1991.

[24] D. Shasha and M. Snir, "Efficient and Correct Execution of Parallel Programs," *ACM Trans. Programming Languages and Systems,* vol. 10, no. 2, pp. 282-312, Apr. 1988.

[25] G. Sohi, S. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. Int'l Symp. Computer Architecture,* pp. 414-425, June 1995.

[26] *The SPARC Architecture Manual Version 9,* SPARC Int'l Inc., Englewood Cliffs, N.J.: Prentice Hall PTR, 2000.

[27] Standard Performance Evaluation Corp., http://www.spec.org/, 2004.

[28] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "A Scalable Approach to Thread-Level Speculation," *Proc. Int'l Symp. Computer Architecture,* pp. 1-12, June 2000.

[29] P.Y. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," *Proc. Int'l Conf. Parallel Processing,* pp. 528-535, Aug. 1986.

[30] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew, "The Superthreaded Processor Architecture," *IEEE Trans. Computers,* special issue on multithreaded architectures, vol. 48, no. 9, pp. 881-902, Sept. 1999.

[31] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Architecture,* pp. 161-173, Feb. 1998.

Marcelo Cintra received the BS and MS degrees from the University of Sao Paulo in 1992 and 1996, respectively, and the PhD degree from the University of Illinois at Urbana-Champaign in 2001. He is an assistant professor of computer science at the University of Edinburgh. His research interests span parallel and multithreaded architectures and optimizing compilers. He is a member of the ACM, the IEEE, and the IEEE Computer Society. More information about his current research activities can be found at http://www.homepages.inf.ed.ac.uk/mc.

Diego R. Llanos received the MS and PhD degrees in computer science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Dr. Llanos is an associate professor of computer architecture at the University of Valladolid, and his research interests include parallel and distributed computation, computer system performance evaluation, and automatic parallelization of sequential code. He is a member of the IEEE and the IEEE Computer Society. More information about his current research activities can be found at http://www.infor.uva.es/~diego.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.