Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors[1]

Ye Zhang†, **Lawrence Rauchwerger**‡, and **Josep Torrellas**†

†Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801
y-zhang2,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/iacoma/
‡Computer Science Department
Texas A&M University, College Station, TX 77843
rwerger@cs.tamu.edu

Abstract

Run-time parallelization is often the only way to execute the code in parallel when data dependence information is incomplete at compile time. This situation is common in many important applications. Unfortunately, known techniques for run-time parallelization are often computationally expensive or not general enough. To address this problem, we propose new hardware support for efficient run-time parallelization in distributed shared-memory (DSM) multiprocessors.

The idea is to execute the code in parallel speculatively and use extensions to the cache coherence protocol hardware to detect any dependence violations. As soon as a dependence is detected, execution stops, the state is restored, and the code is re-executed serially. This scheme, which we apply to loops, allows iterations to execute and complete in potentially any order. This scheme requires hardware extensions to the cache coherence protocol and memory hierarchy of a DSM. It has low overhead. In this paper, we present the algorithms and a hardware design of the scheme. Overall, the scheme delivers average loop speedups of 7.3 for 16 processors and is 50% faster than a related software-only method.

1 Introduction

While there has been much work on automatic extraction of parallelism at compile time for multiprocessors [3, 5, 8], current parallelizing compilers often have only limited success. One of the reasons for this is that access patterns sometimes depend on the input data and, therefore, the information available at compile time is incomplete. This is common in applications with irregular domains or interactions. A few examples are SPICE for circuit simulation, DYNA–3D and PRONTO–3D for structural mechanics modeling, GAUSSIAN and DMOL for quantum mechanical simulation of molecules, CHARMM and DISCOVER for molecular dynamics simulation, and FIDAP for modeling complex fluid flows. Therefore, it has become clear that static analysis must be complemented with methods capable of extracting parallelism at run-time.

Most previous software approaches to run-time parallelization for multiprocessors have concentrated on developing methods for constructing execution schedules for partially-parallel loops. These are loops whose parallelization may require synchronization to ensure that the iterations are executed in the correct order. These methods are often based on an *inspector* loop that analyzes the data access patterns ([4, 10, 13, 15] to name a few). The inspector usually yields a partitioning of the iteration space into subsets called wavefronts. Each wavefront is then executed in parallel by the *executor*, with barriers separating the wavefronts. This inspector-executor method is also applied to fully-parallel loops. Unfortunately, in general, the inspector may be both computationally expensive and have side-effects. Consequently, it can be argued that the inspector-executor approach is not a generally applicable method.

Recently, we have introduced a framework for software run-time parallelization for multiprocessors [13]. It has two main characteristics. First, instead of finding a valid parallel execution schedule for the loop, it focuses on simply deciding whether or not the loop is fully parallel. Second, instead of distributing the loop into inspector and executor, it executes the loop speculatively as a `doall`. At the end, a run-time test checks whether there were any cross-iteration dependences. If the test fails, then the variables updated are restored to their original values and the loop is re-executed serially.

This framework has several advantages. A simple pass-fail test requires less computation time than a full schedule of the loop. A second advantage is that, by using speculation, the technique is made generally applicable. Unfortunately, one important shortcoming of the scheme is that the loop must be completely executed before it can be determined whether or not it was fully parallel. To eliminate this shortcoming and reduce the overhead of the scheme, in this paper we propose to perform the speculative run-time parallelization in hardware.

The scheme proposed performs speculative run-time parallelization in an efficient manner for distributed shared-memory (DSM) multiprocessors. The idea is to execute the code in parallel speculatively and use extensions to the cache coherence protocol hardware to detect any dependence violation. As soon as a dependence is detected, execution stops, the state is restored, and the code is re-executed serially. The scheme requires hardware extensions to the cache coherence protocol and memory hierarchy of a DSM. It operates with low overhead. In this paper, we present the algorithms and a hardware design of the scheme. We show that the scheme delivers average loop speedups of 7.3 for 16 processors and is 50% faster than the software-only method described above.

The scheme proposed is related to several concurrently-proposed schemes for speculative parallelization inside a multiprocessor chip [7, 11, 14]. While all schemes have in common that they add hardware extensions to the cache coherence protocol, ours aims at larger machines. Our scheme allows the iterations of a loop to execute and complete in potentially any order and places no limitation on the working set of the tasks. The on-chip designs place some restrictions on these issues. However, thanks to their small size, they are able to exploit both full and partial parallelism and recover easily from wrong speculation.

This paper is organized as follows: Section 2 outlines speculative run-time parallelization in software, Section 3 presents our hardware scheme, Section 4 shows an implementation of it, Section 5 discusses how we evaluate it, and Section 6 evaluates it.

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from Intel and IBM.

2 Speculative Run-Time Parallelization in Software

We have recently proposed the LRPD test, a new algorithm that uses speculation to parallelize loops at run time [13]. In this section, we first describe some preliminaries of loop parallelization, and then outline the algorithm and discuss some of its shortcomings.

2.1 Preliminaries of Loop Parallelization

A loop can be executed in parallel without synchronization only if the outcome of the loop does not depend upon the order of the execution of the different iterations. To determine whether or not the order of the iterations affects the semantics of the loop, we need to analyze the data dependences across iterations (or cross-iteration dependences) [1]. There are three types of data dependences, namely *flow* (read after write), *anti* (write after read), and *output* (write after write). If there are no anti, output, or flow dependences across iterations, the loop can be executed in parallel. Such a loop is called a `doall` loop. If, instead, there are flow dependences across iterations, the loop cannot generally be executed in parallel. For example, the loop in Figure 1-(a) cannot be executed in parallel because iteration i needs the value that is produced in iteration $i-1$. Finally, if there are anti or output dependences only, the loop must be modified to remove all these dependences before it can be executed in parallel. While not all such situations can be handled efficiently, an effective transformation is *privatization.*

```
do i=1,n
A(i) = A(i) + A(i−1)
enddo
```
(a)

```
do i=1,n/2
tmp = A(2*i)          (S1)
A(2*i) = A(2*i−1)
A(2*i−1) = tmp        (S2)
enddo
```
(b)

```
do i=1,n
A(f(i)) = ...         (S1)
... = A(g(i)) + ...   (S2)
enddo
```
(c)

Figure 1: Examples of loops.

In privatization, we create, for each processor participating in the execution of the loop, private copies of the variables that cause anti or output dependences. The loop can then be executed in parallel. For example, the loop in Figure 1-(b) has an anti dependence between statement `S2` of iteration i and statement `S1` of iteration $i+1$. This dependence can be removed by privatizing variable `tmp`.

In the following algorithm, we consider an array privatizable if each of its elements behaves in either of two ways: it is read-only or every read of it is preceded by a write to it in the same iteration. In general, privatizable variables are temporary variables used as workspace within an iteration.

2.2 Outline of the Algorithm

Consider a loop for which the compiler cannot determine whether or not cross-iteration dependences exist. An example of such loop is shown in Figure 1-(c), where arrays *f()* and *g()* are set by the inputs to the program. In this case, we speculatively execute the loop as a `doall` and, right after that, execute code to determine whether the loop was in fact parallel. In addition, if it is suspected that some data dependences could be removed by privatization, the compiler can speculatively privatize array A. Finally, if the run-time test finds that the loop was not parallel, then the loop is re-executed sequentially. Similarly, if an exception occurs during the speculative parallel execution, the execution is aborted and restated serially. Overall, to parallelize a loop speculatively in this manner, we need two supports, namely a way of saving and restoring state for possible sequential re-execution of the loop, and a method to detect cross-iteration dependences that occurred during execution. We consider these issues in turn.

2.2.1 Mechanism for Saving and Restoring State

Before a loop can be executed speculatively, we need to save the state of the arrays that will be modified in the loop. If the pattern of access to an array is dense, it makes sense to save the whole array. However, if the pattern of access is sparse, it is better to save individual elements into a sparse storage data structure like a hash table just before they are modified. This is done in software [13]. Note that the compiler only needs to save modifiable shared arrays – not read-only or privatized arrays. Finally, it is also possible to reduce the amount of backup requirements by identifying and checkpointing a point of minimum state in the program prior to the loop. In all cases, after the loop is executed, if it is not found parallel, we restore the arrays from their backups.

2.2.2 The LRPD Test to Detect Dependences

This test only flags the existence of cross-iteration dependences – it does not identify them. Given a loop, this test needs to be applied only to those arrays whose dependences cannot be analyzed at compile-time. This test can be applied to privatized and non-privatized arrays. In the former case, the test has an extra step and the arrays take more memory space. However, there is a higher chance of finding the loop parallel. The compiler and the programmer can use heuristics to decide whether or not to use privatization. This test can also validate parallelized reductions but, for brevity, this feature is not presented in this paper.

This algorithm has two phases, namely *Marking*, performed during the speculative execution of the loop in parallel, and *Analysis*, performed after the speculative execution. Before the loop is executed, for each shared array $A(1:s)$ whose dependences cannot be determined at compile time, we make a backup copy. In addition, we declare three shadow arrays, namely the read $(A_r(1:s))$, write $(A_w(1:s))$, and non-privatization $(A_{np}(1:s))$ shadow arrays, all initialized to zero. $A_{np}(1:s)$ is only useful for the arrays that are privatized. In addition, we initialize scalar Atw to zero. The steps of the algorithm are as follows:

1. *Marking Phase.* In each iteration of the loop, do:

(a) If we write to $A(i)$: set $A_w(i)$.

(b) If we read from $A(i)$: if $A(i)$ is not written in this iteration (neither before nor after the read), set $A_r(i)$; if $A(i)$ is not written in this iteration before this read, set $A_{np}(i)$.

(c) At the end of the iteration, count how many different elements of A have been written in this iteration and add the count to Atw.

2. *Analysis Phase.* For each shared array A do as follows. The last two steps apply only to privatized variables.

(a) Compute Atm as the number of non-zero $A_w(i)$ for all elements i of the write shadow array.

(b) If $any(A_w(:) \wedge A_r(:))$[1], that is, if the marked areas are common anywhere, then the loop is not a `doall` and the execution is aborted. This case means that at least one element $A(i)$ is written in one iteration and read (and not written) in another. There is, therefore, at least one flow or anti dependence. Since we do not know which iteration happened first, we assume the worst case of a flow dependence. Note that, if the iterations that read the element also wrote it, A_r would not be set and this test would not flag any problem. We will, however, detect this case later.

[1] *any* returns the "OR" of its vector operand's elements: $any(v(1:n)) = (v(1) \vee v(2) \vee \ldots \vee v(n))$.

(c) Else if $Atw = Atm$, then the loop is a `doall` without privatizing array A. If this test is true, no two iterations of the loop write to the same element of A. Consequently, there are no dependences, since the combination of tests (b) and (c) checks for all dependences.

(d) Else if $any(A_w(:) \wedge A_{np}(:))$, A is not privatizable and the loop, as executed, is not a `doall`. If this condition is true, it means that an element $A(i)$ is read before being written in the same iteration. Again, we have to assume the worst case, which is the reading iteration being preceded by the writing one.

(e) Otherwise, the loop was transformed into a `doall` by privatizing the shared array A.

Figure 2 shows an example of a loop. In most cases, each element of the shadow arrays holds the iteration number where the read or write occurred, instead of just one bit. This is necessary to implement the marking phase efficiently [13]. Therefore, if we want to support loops of up to 2^{16} iterations, for example, we need 2 bytes per element for each shadow array.

```
do i=1,5
  z = A(K(i))
  if (B1(i) .eq. .true.) then
     A(L(i)) = z + C(i)
  endif
enddo

 B1(1:5) = (1 0 1 0 1)
 K(1:5) = (1 2 3 4 1)
 L(1:5) = (2 2 4 4 2)
 A(:) has 4 elements
```

(a)

```
do i=1,5
  markread(K(i))
  z = A(K(i))
  if (B1(i) .eq. .true.) then
     markwrite(L(i))
     A(L(i)) = z + C(i)
  endif
enddo
```

(b)

Variable	Value			
	1	2	3	4
Aw	0	1	0	1
Ar	1	1	1	1
Anp	1	1	1	1
Aw(:) ∧Ar(:)	0	1	0	1
Aw(:) ∧Anp(:)	0	1	0	1
Atw	3			
Atm	2			

(c)

Figure 2: `Do` loop (a) that is transformed for speculative execution (b). The `markwrite` and `markread` operations update the appropriate shadow arrays. The shadow arrays are shown in (c). In the example, the test fails.

2.2.3 Implementation and Compiler Integration

The implementation of this algorithm is described in [13]. In a DSM system, each processor allocates a private copy of the shadow arrays in its local memory. The marking phase is performed locally on the private copy. Then, as part of the analysis phase, the contents of the private shadow arrays are merged in parallel. This is called the *merging* step. It causes most of the analysis overhead because processors access remote data. In addition, if parallelization requires privatization, each processor allocates a copy of the array under test in its local memory. In this case, the compiler replaces the original references with references to the private arrays and, therefore, there is no need to back up the array under test.

We envision this algorithm to be integrated in a front-end parallelizing compiler. When the compiler tries to parallelize a program, it may fail on some loops. Then, the user can examine the compiler feedback and, based on her intuition of the overhead of run-time parallelization, force the compiler to perform run-time parallelization on some of the non-analyzable loops. Alternatively, the compiler may use heuristics and statistics about the parallelization success-rate in previous executions and automatically decide when run-time parallelization could be profitable. If run-time parallelization is to be performed, the compiler inserts code to back up arrays, update the shadow arrays every time that the arrays under test are accessed, perform the analysis and, if the analysis fails, restore the data and restart the loop on one processor.

2.2.4 Improvements

We can parallelize more loops if we extend the algorithm. A loop can still be run in parallel if the following holds for each element of the array under test: if one iteration reads the element, the same iteration first writes the element or no previous iteration has ever written the element. For example, all the loops in Figure 3 can be run in parallel if the single array element accessed is privatized, and the private copies initialized. To parallelize these loops, the algorithm needs to use an extra shadow array ($A_w min(1:s)$) similar to the write shadow array [12].

It 1	It 2	It 3		It 1	It 2		It 1	It 2		It 1	It 2
Wr	Wr	Wr		Rd	Wr		Rd	Rd		Rd	Wr
Rd	Rd	Rd			Rd			Wr		Wr	Rd
								Rd			

Figure 3: Examples of iterations of loops that can be parallelized by extending the algorithm.

Under privatization, another issue is the ability to copy data selectively between the shared array under test and its private copies, and vice-versa. While we can make a full copy of the shared array into the private arrays before the loop starts, it is more efficient to copy on-the-fly only those array elements that are really needed. These copy operations are called ***read-ins***. Furthermore, a relatively small fraction of loops requires that some elements from the private arrays be copied to the shared array after the loop finishes. These copies are called ***copy-outs*** [13].

Finally, the algorithm as presented detects cross-iteration dependences. We refer to it as the *iteration-wise* test. Sometimes, a ***processor-wise*** test, which tests only for cross-processor dependences, delivers higher performance. Checking only for cross-processor dependences does not require any algorithmic modifications. It is achieved by partitioning the iteration space into a number of blocks of contiguous iterations equal to the number of processors. Each processor's work can then be considered a "super-iteration" and all the rules of the previously-presented algorithm apply. This approach has two advantages. First, a loop that is not fully parallel passes the processor-wise test when data-dependent iterations are assigned to the same processor. Second, each entry in a shadow array now only needs to be one bit [13]. Consequently, shadow arrays are now accessed with bitmap operations, resulting in significant space savings. The disadvantage of the processor-wise test is that it requires blocked static iteration scheduling. This is necessary to insure that each processor gets a block of contiguous iterations. This scheduling policy may cause load imbalance, severely reducing performance.

2.3 Shortcomings

This scheme has at least two important shortcomings. The first one is the overhead of the analysis phase and the extra instructions necessary for the marking. For the loops where most of the work performed needs to be shadowed, this overhead may be significant. The second shortcoming is the slowdown incurred when the parallelization fails: we know that the parallelization failed only after loop completion.

3 Speculative Run-Time Parallelization in Hardware

To address these problems, in this paper we propose a novel scheme that performs the speculative run-time parallelization in hardware. This scheme reduces the run-time overhead, keeps failure penalty to a minimum by aborting the parallel execution as soon as a cross-iteration data dependence occurs, is more scalable with the number of processors, and delivers extra functionality. In this section, we present the scheme's algorithms, while in the next one, we present a detailed hardware design.

3.1 The Main Idea

This scheme extends the cache coherence protocol hardware of a DSM multiprocessor with extra transactions to flag any cross-iteration data dependences. When a dependence is detected, the parallel execution is immediately aborted. We try to embed the new operations on existing cache coherence protocol transactions. If that is not possible, we try to minimize the complexity of any new transaction. These transactions are designed such that, like the baseline ones, get serialized in the directory. This minimizes races in the protocol. In addition to these transactions, we add some extra state in the tags of all the caches, and some fast memory in the directories.

The new transactions are grouped into two sets, which we call the *non-privatization* and *privatization* algorithms. These algorithms are used by the non-privatized and privatized arrays under test respectively. As before, if the privatization algorithm is used, the arrays take more memory space but there is a higher chance that the loop is found parallel. In all cases, the rest of the arrays in the loop are unaffected by these extensions. We describe these two algorithms next.

3.2 Non-Privatization Algorithm (NPA)

In the non-privatization algorithm, we identify as parallel only those loops where each element of the array under test is either read-only (*ROnly*) or is accessed by only one processor (*NoShr*, for not-shared). A pattern where an element is read by several processors and later written by one is flagged as not parallel. Consequently, our algorithm is as conservative as the software approach of Section 2.2. However, as we will see, it has the advantage over the software approach that it is processor-wise under any iteration schedule.

The fast memory that we add to the directory keeps some state for each element of the array under test. First, it keeps a *ROnly* bit and a *NoShr* bit, which tell whether the element is read-only or not-shared respectively. In addition, we need *First*, a field that keeps the ID of the processor that first accesses the element. Before the loop starts, all these bits are cleared. If the first access is a write, both *First* and *NoShr* are set. If, instead, the first access is a read, only *First* is set. In that case, if what follows is a read by a different processor, we set *ROnly*, while if it is a write by the same processor, we set *NoShr*. In all cases, a write to a *ROnly* element causes the failure of the parallelization. Similarly, a processor different from *First* trying to access to a *NoShr* element or trying to write the element causes a failure. These algorithms are shown in Figure 4-(a) and 4-(b).

If every time that a processor accessed the array under test the directory had to be accessed, the cost would be too high. Instead, since the array elements are cachable, the *First*, *NoShr*, and *ROnly* fields are also sent to the cache and stored in the tags of the corresponding cache line. There is, however, no need to store the full *First* bits; a processor only needs to know whether the *First* ID points to itself, to no processor, or to another processor. Consequently, only two bits are necessary for *First* in the cache (one state is unused). Figure 5-(a) shows all the state necessary for this algorithm per array element.

```
if ( (First != ThisProc) && (NoShr == 1) )
   FAIL   /* Read data that has been
             written by another proc */
else
   read data
   if (First == NONE)
      First = ThisProc
   else if ( (First != ThisProc) && (ROnly == 0) )
      ROnly = 1
```

(a): Processor read.

```
if ( (First != ThisProc) || (ROnly == 1) )
   FAIL   /* Write data that has been read or
             written by another proc */
else
   write data
   if (First == NONE)
      First = ThisProc
   NoShr = 1
```

(b): Processor write.

Figure 4: Compact form of the non-privatization algorithm. In the algorithm, *ThisProc* is the ID of the processor that accesses the element.

The per-element bits in the tags of the different caches and directory are kept coherent. Since most accesses that a processor initiates will not induce a change in the bits, the directory and the other caches do not need to be notified. Furthermore, even if the bits are changed, if the cache line is dirty, there is no need to notify anyone. This is because, since the line is dirty, any other processor that references any element in the line, will have to get the line from the owner cache. At that point, it gets an updated value of the tag bits as well. Of course, when a dirty line is displaced from a cache or is forced into a write back, the directory is updated with the tag state for each element in the line.

We can now include in the algorithms of Figure 4 the changes in the state of the cache tags. Because of lack of space, Figure 6 shows only the write transactions. The read transactions are shown in [16]. In the figure, the state in the directory and in the cache tags is denoted with the prefixes *dir* and *tag* respectively. In the home node, directory and memory are accessed at the same time. All algorithms assume in-order delivery of messages.

The races in these transactions are handled as in the original cache coherence protocol transactions. No data inconsistency occurs because all transactions directed to the same memory line are serialized in the corresponding directory. For example, consider two processors with shared copies of a given line, that write to the same (previously unaccessed) element in the line. Both writes induce messages to the directory to request ownership of the line. In these messages, we piggyback updates to the *dir.First* and *dir.NoShr* fields of the element. The first request that arrives at the directory causes an invalidation to be sent to the other cache. When the second request reaches the directory, it is bounced or buffered until the first transaction is completely finished. At that point, the second request is processed. Since it finds that the directory points to an owner cache, it causes a write back from that cache. When the data arrives, the *dir.NoShr* is already set. The result is the *FAIL* statement of algorithm (b) in Figure 6.

A characteristic of this algorithm is that reads can also cause races. Again, no state inconsistency is possible because of the serialization in the directory. For example, if two processors read the same element from a cached line for the first time, both will send an update to the *dir.First* field. The first message that reaches the directory is processed. The second one bounces and returns to the sender cache. On arrival at the cache, it sets *tag.First* to OTHER.

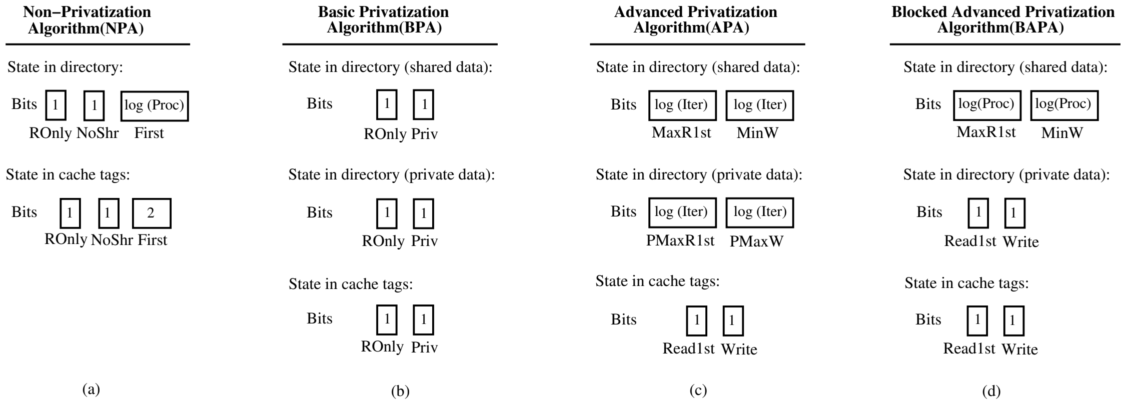


Figure 5: State necessary per array element to implement the non-privatization algorithm (NPA), basic privatization algorithm (BPA), advanced privatization algorithm (APA), and blocked advanced privatization algorithm (BAPA). Of these algorithms, NPA and BAPA are processor-wise, while BPA and APA are iteration-wise. In all cases, there is a single set of hardware bits that are used differently depending on the algorithm used.

```
if (Cache_hit)
   if ( tag.First == OTHER || tag.ROnly == 1 )
      FAIL
   else
      if (cache_state == CLEAN)
         Send write_req with state-bits update to home
      else
         Write to cache
else
   Send write_req with state-bits update to home
/*no need to tell the directory*/
tag.First = OWN
tag.NoShr = 1
```

(a): Processor write.

```
if (dir_state == SHARED)
   send invalidations to sharers
   wait for acks
   /*no need to update the directory*/
else
   if (dir_state == DIRTY)
      send writeback and invalidate to owner node
      wait for reply
      update dir.First, dir.NoShr, and dir.ROnly
      /*update directory using the tag state of
        all the words of the dirty line*/
if ( (dir.First != RequestorID && dir.First != NONE)
         || (dir.ROnly == 1) )
   FAIL
else
   dir.First = RequestorID
   dir.NoShr = 1
   Copy dir state to tag state for all the words in the line
   Send line and tag to requester
```

(b): Home receives a write request.

Figure 6: Write transactions for the non-privatization algorithm in extended form.

3.3 Privatization Algorithm (PA)

In this algorithm, each processor works on a private copy of the array under test. The algorithm needs state in the directories of both the shared array and its private copies. Following the discussion in Section 2.2.4, we have different versions of this algorithm. The Basic Privatization Algorithm (BPA) is an iteration-wise algorithm that parallelizes the same loops as the software scheme of Section 2.2.2. For a loop to be parallel, individual elements of the array under test must either be read-only or, if not, be written before being read in each iteration that accesses them. The support for this algorithm is the simplest. The Advanced Privatization Algorithm (APA) is an iteration-wise algorithm that parallelizes the loops parallelized by the BPA plus those shown in Figure 3 and discussed in the first paragraph of Section 2.2.4. This algorithm is the most general and requires the most support. Finally, the Blocked Advanced Privatization Algorithm (BAPA) is a processor-wise version of the APA. As a result, it parallelizes a superset of the loops parallelized by the APA. This algorithm, by restricting the schedule of the iterations to blocked static scheduling, requires less support than the APA. In this paper, we present the APA for generality. We also outline the other algorithms.

Given an array element, if an iteration reads it before the same iteration writes it, we call the iteration a *read-first* iteration for the element. Using this concept, the APA works as follows. The directory of the shared copy of the array maintains two time stamps for each array element. One keeps the number of the highest *read-first* iteration for the element executed so far by any processor (*MaxR1st*). The second one keeps the number of the lowest iteration executed so far by any processor that involved writing the element (*MinW*). The parallelization fails when *MaxR1st* is larger than *MinW*.

The algorithm serializes all the transactions related to an array element in the directory of the shared copy of that element. In the most general implementation, the sizes of *MaxR1st* and *MinW* are equal to the logarithm of the number of loop iterations (Figure 5-(c)). However, if the loop has so many iterations that the time stamps would overflow, we synchronize all processors periodically after a fixed number of iterations have been executed. At synchronization points, the time stamps are reset.

During the speculative execution, processors access private data. To identify read-first iterations, the directories of the private copies of the array keep, for each element, two time-stamps: the number of the highest read-first iteration for the element executed so far by the processor (*PMaxR1st*, where *P* stands for private), and the number of the *highest* iteration executed so far by the processor, that involved writing the element (*PMaxW*). We need the highest write iteration because, in a read, we compare *PMaxW* to the current iteration number to determine whether we found a read-first iteration.

Finally, to avoid having to access these directories too frequently, the tags of the caches keep a summary of the directory state. For each element, they keep 2 bits to indicate whether the current iteration is read-first for the element (*Read1st*), and whether the iteration has written to the element (*Write*). *Read1st* and *Write* are cleared at the beginning of each iteration. When an iteration reads an element of the array under test, if both *Read1st* and *Write* for the element are zero, it sets *Read1st*. The complete state is shown in Figure 5-(c).

The algorithm proceeds as follows. Every time that a processor reads an element of the array under test, it checks whether this is a *read-first* iteration for the element. For the check, it can use the state of the cache tags (both *Read1st* and *Write* are zero) or, if the line is not in the cache, the state of the directory for the private array (*PMaxR1st* and *PMaxW* are both lower than the current iteration number). If the iteration is *read-first*, the directory for the shared array is notified. In the directory, the current iteration number is compared to *MinW*. If the former is larger, the parallelization fails; otherwise, *MaxR1st* is set to the maximum of its current value and the current iteration number. Finally, the state in tags and directory is updated as necessary.

Every time that a processor writes to an element of the array under test, it checks whether this is the first write to the element in this iteration. For the check, it can use the state of the cache tags (*Write* is zero) or, if the line is not in the cache, the state of the directory for the private array (*PMaxW* is less than the current iteration number). If this is the first write, the directory for the private array is notified for two reasons. The first reason is to update *PMaxW* to the current iteration number. *PMaxW* needs to be kept uptodate because, in conjunction with *PMaxR1st*, identifies *read-first* iterations. The second reason is that, it is possible that this may be the very first write of this processor to this element (*PMaxW* is still zero). If that is the case, the directory for the shared array is notified. In the latter directory, the current iteration number is compared to *MaxR1st*. If the former is lower, the parallelization fails; otherwise, *MinW* is set to the minimum of its current value and the current iteration number. More details can be found in [16].

Since the private copies of the array under test start-off uninitialized, every time that a processor accesses a memory line for the first time, it performs a read-in. This is done by the protocol controller in the directory of the private array. Specifically, if the element requested by the processor has the *PMaxR1st* and *PMaxW* fields equal to zero, and all the other words of the line do as well, the protocol controller issues a read to the shared array.

In the relatively infrequent case where the privatized array is live after the loop, the algorithm must support copy-outs. In this case, the APA can be extended as follows. The directory of the shared array should keep, per element, the highest iteration executed so far by any processor that involved writing the element (*MaxW*), and the ID of the processor that executed it (*MaxWID*). While the loop is in progress, for every first write to an element in a given iteration, the directory of the shared array is notified. If the iteration number is higher than *MaxW*, both *MaxW* and *MaxWID* are updated. At the end of the loop, in software, we read *MaxWID* for each element and copy the element from the corresponding private array to the shared array. Overall, since copy-outs require some extra hardware support, we may choose not to support them and use a software routine instead.

The BPA is much simpler because we only need to ensure that each element is either read-only or, if not, is written before being read in each iteration that accesses it. There is no need to worry about the sequence of the iterations that access it. Consequently, the directory of the shared array keeps only two bits per element to flag the two legal behaviors: *ROnly* and *Priv* (Figure 5-(b)). If both bits get set, the algorithm fails. To avoid having to access the directory of the shared array too frequently, the state is replicated in the directories of the private arrays and in the cache tags.

Finally, the BAPA parallelizes a superset of the loops that APA parallelizes. However, it forces the loop to be blocked-scheduled statically. This simplifies the algorithm significantly because each processor executes one single "super-iteration". As a result, the *MaxR1st* and *MinW* fields in the directory of the shared array are only log(Proc) bits wide, and the fields in the directory of the private arrays, only 1 bit wide (Figure 5-(d)). Finally, copy-outs are done as in APA. Overall, if the loop is load balanced, this algorithm is very attractive because it is simple and, thanks to its processor-wise nature, can parallelize even some loops with cross-iteration dependences.

3.4 Comparing the Software and Hardware Schemes

To compare the two schemes, we recall the steps of the software scheme. Before the loop starts, an *initialization phase* backs up the array under test (if no privatization is attempted) or copies the array into the private copies (if privatization is attempted). The latter operation can also be done with on the fly read-ins. Then, a *zeroing phase* clears all the shadow arrays. During the loop execution or *marking phase*, additional instructions inserted by the compiler mark the shadow arrays. After loop execution, the *analysis phase* checks the shadow arrays for any dependence violation. If the parallelization succeeded, the *conclusion phase* performs some calculations to collect the results and maybe performs copy-outs. If the parallelization failed, the *restore phase* restores the array unless it was privatized, and execution restarts in sequential mode. In the hardware scheme, the following is performed in hardware or not performed: on the fly read-ins, and the zeroing, marking, and analysis phases. All the other phases are the same.

The hardware scheme has several advantages over the software scheme. The biggest one is that failure to parallelize is detected on the fly, as soon as the dependence violation occurs. A second advantage is that, as discussed above, several operations are performed in hardware, therefore reducing overheads. The savings come largely from better memory system performance and, to a lesser extent, from fewer instructions executed (Section 6.1). The former results mainly from not issuing the remote requests of the merging step of the analysis phase, and from not bringing large shadow arrays into the cache as in the marking phase.

A third advantage is that the hardware scheme has better scalability with the number of processors. This is because the total amount of work involved in the zeroing phase and in the analysis phase of the software scheme increases with the number of processors. The hardware scheme does not have these phases.

A fourth advantage of the hardware scheme is that it has less space overhead. Consider the most space-consuming tests, namely the iteration-wise tests. The software scheme requires, per array element and processor, 3 shadow locations (for the basic scheme of Section 2.2.2) or 4 shadow locations (for the advanced scheme of Section 2.2.4). Each shadow location needs $log(iterations)$ bits but, because we access them in software, each location can only be efficiently allocated 1, 2 or 4 bytes. According to Figure 5, the hardware scheme requires, per array element and processor, only 2 bits for the basic scheme (state in directory of private data for the BPA) or $2 * log(iterations)$ bits for the advanced scheme (state in directory of private data for the APA). In our calculations, we are neglecting the state in the directory of shared data and cache tags. Overall, therefore, the savings in space are very large, especially for the basic scheme.

A fifth advantage of the hardware scheme is that the non-privatization test is processor-wise without requiring static scheduling. Finally, another advantage of the hardware scheme is that it can be applied to pointer-based C code more efficiently than the software scheme. This is because the shadow state that we add to each word is allocated in a short tag that is associated with the memory line. This minimizes shadow memory consumption and eliminates the need of knowing the bounds of access regions.

Of course, the major shortcoming of the hardware scheme is that we need to add extra hardware to the memory subsystem. The actual support required is considered next.

4 Implementation

We now consider some implementation issues and propose a hardware design.

4.1 Implementation Issues

The previous discussion of the hardware scheme gives rise to several implementation issues. The first one is how the hardware knows whether to use the plain cache coherence protocol or any of the proposed extensions. A good approach is for the compiler to insert a different type of load and store instruction depending on the algorithm to be used. Based on the instruction, the cache tags bits and directory state (Figure 5) will be interpreted differently and different protocol transactions will be used. We can use three different loads (and stores): one for the plain protocol, one for the non-privatization algorithm, and one for the privatization algorithm. That is all we need because the processor could be pre-configured to use the BPA or the APA for the privatization algorithm, and because the BAPA is simply a degenerate case of APA. If necessary, we do not even need to support the non-privatization algorithm because the privatization one is more general. In this case, we will suffer a modest loss in performance. This type of support is similar to the one used in existing microprocessors for speculative execution.

A less desirable alternative would be to choose the algorithm based on the address of the array accessed. This requires a lookup table that keeps address ranges and is checked before accessing the cache tags. This design is hard because a single array may span several pages and different types of arrays may share the same page. In addition, this table would be in the critical path of a cache access and, therefore, could slow down the microprocessor. In any case, it is possible that, in pointer code, we do not even know the boundaries of a data structure.

A second issue is how to clear the cache tag and directory state (Figure 5), as it is sometimes required in the algorithms presented. We start with the cache tags. While all algorithms clear the tags at the beginning of the loop, the advanced privatized algorithm additionally clears them at the beginning of each iteration. Consequently, we need a low-overhead, selective way of clearing the tags. Selectivity is accomplished by storing a special bit pattern in the (per-word) tags that belong to privatized data. This bit pattern is the unused code of the *First* field of the non-privatization algorithm. Recall that the non-privatization algorithm uses only 3 states of *First* and that the APA uses only 2 tag bits (the *First* bits are unused). Consequently, only the tags with this bit pattern will be cleared. This bit pattern is automatically inserted into the tag every time that a location is accessed by the special privatization load and store instructions. Therefore, there is no need to explicitly initialize these tags when a privatized array is allocated.

To clear these tags with low overhead, we need special circuitry. We can design circuitry that clears all these tags in 1 cycle and is enabled by issuing a store to a reserved address. This store is issued at the beginning of each iteration of any loop that has at least one array under test that uses the APA. For security reasons, we can encapsulate this store inside the synchronization step needed by each APA iteration to get the next iteration number. Overall, with this support, at the beginning of each iteration we can clear, in 1 cycle, all the cache tags that use the APA. Given reasonably-sized loop bodies, this overhead is negligible. Note that both primary and secondary cache tags must be cleared.

All our algorithms clear the cache tags and our special directory state at the beginning of the loop. To clear the tags, we can again use a hardware signal that clears all cache tags and is enabled by a store to a reserved address. To clear the directory state, we can also use hardware support that minimizes processor overhead. For example, the special memory where the state is stored can have a reset signal, or the protocol controller can write a zero on each location. In all these cases, the processor only needs to issue a single store. Since these activities only happen at the beginning of the loop, we can afford to use a fast system call to invoke them. Even if we had no hardware support and the processor had to write a zero on each location of the special memory, codes would not necessarily see this overhead. For example, the clear operations could be scheduled at the end of loops, while all processors except the one executing the serial section, are idle. If there are several parallel loops in a row and, therefore, there is no time to clear the state in between, we could have each loop use a fraction of the storage, and only clear everything at the end of the series of loops. Similar scheduling considerations must be taken into account by the protocol controller if the latter is responsible for clearing the storage by writing zeros.

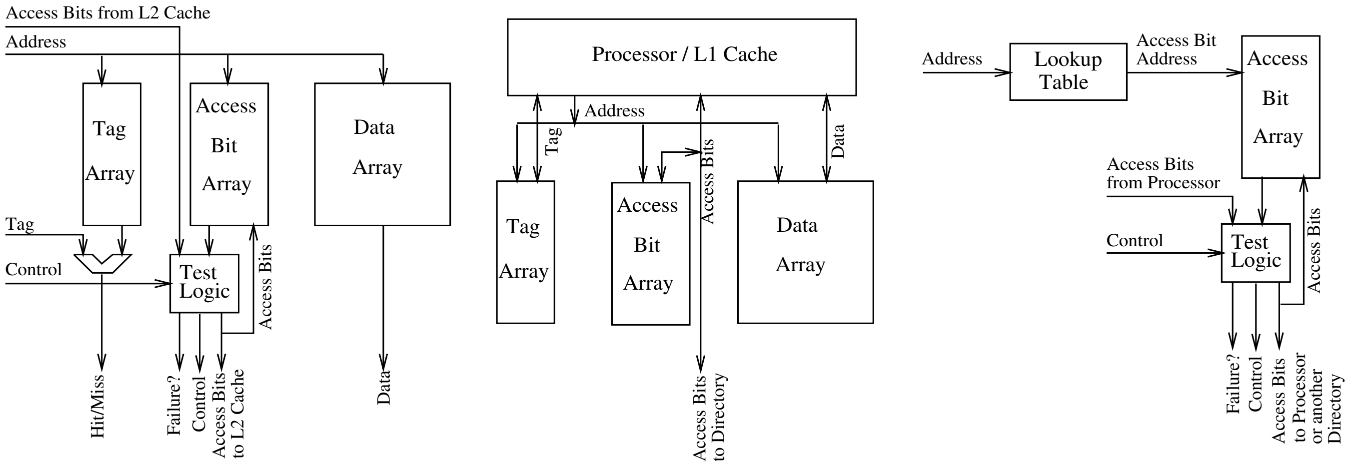
Note that, for all these clear operations to work, the processor must not be context switching between two processes that are trying to parallelize speculatively two different loops. Otherwise, the clear operations could interfere with each other.

In our implementation, we have not precluded the use of different algorithms on different words of the same cache line. The only constrain is that a word that uses the privatization algorithm cannot share the line with other types of data. This is because this algorithm performs line read-ins. This constrain, however, is not limiting because privatized arrays are always allocated separately by the compiler for speculative execution.

A third issue is how to reduce the space and parallelization overheads of the algorithm. Some applications declare arrays with 8-byte or 16-byte elements. This does not save any tag space because we need to keep hardware bits to support at least 4-byte elements. An unrealistic way to reduce the space overhead would be to use the compiler to eliminate all false sharing and ensure that all the array elements in a cache line are accessed together by the same processor. Then, we would need only one set of state bits per cache line. Completely eliminating false sharing, however, is unrealistic.

Finally, we can save some space in the directory state shown in Figure 5. Instead of allocating these bits for every single memory location, we allocate them only for the memory lines that use our algorithms. Consequently, we store these bits in a dedicated memory that is close to the directory and is accessed at the same time as the directory. To know what entries to access in this memory, however, we need a Lookup Table module that takes as input a physical address as it gets to memory, and outputs a pointer to the dedicated memory. This table must be loaded and maintained by the operating system. Despite the overhead that this may involve, the resulting memory savings are likely to make this approach attractive.

The parallelization overhead is easier to reduce. For the APA, we can group contiguous iterations in chunks and use block cyclic scheduling to schedule the chunks. Each chunk becomes a superiteration. This scheme reduces the overhead because the cache tags need to be cleared only at the beginning of a superiteration, the size of the time stamps decreases



(a) Primary cache (b) Secondary cache (c) Directory

Figure 7: Hardware for speculative run-time parallelization.

because there are fewer effective iterations, and the number of read-first iterations and, in general, messages and parallelization tests decreases. A similar result can be accomplished with loop unrolling. In the extreme case, the superiterations are so large that each processor is assigned only one. This algorithm we called BAPA in Section 3.3. The disadvantage is possible load imbalance. Alternatively, we can use the less powerful BPA.

The previous optimization does not apply to the NPA. The latter is intrinsically processor-wise. There is freedom of iteration assignment and scheduling. There is no need to assign contiguous iterations to a processor. The only constraint is that a processor must execute its iterations in increasing order.

A final issue is how this scheme interacts with speculative instructions that, by accessing unnecessary locations, could cause false dependences. In general, following the assumption in our scheme that these loops are often dependence-free, we also assume that, even with speculative loads, they are often dependence-free. However, we can handle this situation without aborting parallel execution with extra hardware. For example, if a speculative load would trigger a dependence on a cache read, the processor does not update any tag state or issue any protocol transaction. Instead, it stalls until the access becomes non-speculative or is proven incorrect. Most other cases can also be handled with extra hardware. This issue is not considered in our design.

4.2 Hardware Design

We now present the design of an architecture that supports our algorithms. In the following, we refer to the state bits in Figure 5 as the *access bits*. To implement our algorithms, we need three supports, namely storage for the access bits, logic to test and change them, and a table in the directory to allow us to find the access bits for a given physical address. With these supports, we need to modify three parts of the machine, namely the primary and secondary caches and the directory.

The design for the primary cache is shown in Figure 7-(a). The access bit portion of the cache tags is stored in a SRAM table called the *Access Bit Array*. The access bit array, the tag array and the data array all have the same number of entries. The desired entry is selected with the address lines. Once the correct access bit entry is selected, the *Test Logic* performs the operations discussed in Section 3. The actual operation performed is determined by the *Control* input, which identifies whether the processor issued a read or a write, and the type of algorithm being used. The test logic is simple enough to generate the new access bits, control signals, and a signal indicating whether the test failed, at the same time as the tag comparison is done. If the new access bits are different than the old ones, they will be saved back into the access bit array in the next cycle. In addition, if the corresponding cache line is not exclusive in the primary cache, the new access bits are immediately propagated down the memory hierarchy to the secondary cache and directory. Overall, therefore, except for the access bit update, the complete operation is hidden behind the cache access.

For the secondary cache, we also need to provide an access bit array (Figure 7-(b)). After a primary cache miss, if the secondary cache hits, the secondary cache provides both the data and the access bits to the primary cache. The access bits are sent directly to the test logic in the primary cache. If the test logic generates a set of access bits that are different from the old ones, they are propagated down to the secondary cache and directory. In any case, the bits generated are stored in the access bit array of the primary cache.

Finally, Figure 7-(c) shows the directory hardware. In the figure, we use the optimization proposed in Section 4.1 of using a small dedicated memory for the access bits and a lookup table. The address coming from the processor is used to access the lookup table and generate a pointer to the right entry in the access bit array. The rest of the figure is similar to the previous figures except that the access bits generated by the test logic are sent to the processor. In the privatization algorithms, these bits may be sent instead to another directory. Overall, the complete transaction, which includes accesses to the lookup table and access bit array, and the operation of the test logic, is overlapped with the memory and directory access.

Figure 7-(c) corresponds to a directory without a protocol processor. If there is a protocol processor, the function of the test logic and part of the function of the lookup table are performed by the protocol processor.

5 Experimental Setup

We now evaluate the performance of the proposed algorithms. In this section, we present our evaluation methodology and, in the next one, the results.

5.1 Simulation Environment

Our evaluation is based on execution-driven simulations of a CC-NUMA shared-memory multiprocessor using Tango-lite [6]. The multiprocessor modeled has the hardware support described in Section 4.2 to implement the proposed hardware scheme. In addition, since we also evaluate the software scheme of Section 2.2, the simulator is interfaced directly to the output of the Polaris parallelizing compiler [3].

The architecture modeled has 200-MHz simple RISC processors. Each processor has a 32-Kbyte on-chip primary cache and a 512-Kbyte off-chip secondary cache. Both caches are direct-mapped and have 64-byte lines. The write buffers are 4-entry deep. We selected such small caches because the only loops that we can run have smaller working sets than real-life ones. The caches are kept coherent with a DASH-like cache coherence protocol [9]. Each node has part of the global memory and the corresponding section of the directory. We model contention in the whole system except in the global network, which is given a fixed latency. The average contention-free round-trip latencies to the on-chip primary cache, secondary cache, memory in the local node, memory in a remote node with 2 hops, and memory in a remote node with 3 hops are 1, 12, 60, 208 and 291 cycles respectively. These figures increase with resource contention.

In our evaluation, we model and account for all the overheads of both the software and hardware schemes. In either scheme, before the loop starts, the code backs up the array under test. In addition, after loop execution, the code performs the conclusion phase. If parallelization fails, the data is restored. The software scheme is fully automated. It is evaluated with code generated with the Polaris parallelizing compiler. Polaris inserts all the instructions to perform read-ins, and the zeroing, marking and analysis phases. While the instruction overheads of these operations would likely be smaller if we used a wide-issue superscalar in our simulations, we will see in Section 6.1 that most of the performance gains of the hardware over the software scheme come from a better memory system performance rather than from fewer instructions.

For the hardware scheme, our simulator models the extra overheads in the way discussed in Section 4.1. The APA clears the cache tags of all the privatized lines at the beginning of each iteration. This is done in 1 cycle in hardware. All the proposed algorithms clear all the cache tags and the directory state at the beginning of the loop. Even though tag clear and directory state clear are each triggered with a single processor write, we allow 50 cycles for the combined execution of both. This is because, since these operations happen only once per loop, we can afford to use a fast system call. Finally, the time it takes to broadcast a cross-processor interrupt to all processors when a processor detects a dependence violation and the parallel execution must be aborted is 30 microseconds.

5.2 Loops

To evaluate the proposed scheme, we use loops from applications in the Perfect Club set [2] and one application from the National Center for Supercomputing Applications. These applications are very large and cannot be run to completion with our limited simulation resources. Consequently, we test our scheme on individual loops within these applications. The loops chosen must both account for a large fraction of the execution time and not be analyzable by Polaris. Polaris is one of the most advanced parallelizing compilers that currently exist [3]. It is equipped with leading-edge dependence analysis techniques. While examining individual loops is not an ideal situation, we feel that it gives a representative notion of what the proposed system can do. The loops are *ftrvmt5_do9109* from *Ocean*, *pp_do100* from *P3m*, *run_do20* from *Adm*, and *nlfilt_do300* from *Track*. In this paper, we identify the loops with the name of the application they belong to.

Table 1 shows the contribution of the chosen loops to the execution time of the applications *for serial execution*. The measurements are taken on a SGI Powerchallenge. For *Adm*, the table includes the contribution of *run_do20* and 5 other loops that have very similar dependence patterns and structure. From the table, we see that these loops account for about 40% of the time. This figure is the worst case: when the applications are parallelized, this figure increases because the rest of the code consumes less time.

Table 1: Contribution of the chosen loops to the *serial* execution time of the complete applications.

Application	Loop Name	% of *Serial* Time
Ocean	*ftrvmt5_do9109*	36.9
P3m	*pp_do100*	51.1
Adm	*run_do[20,30,40,50,60,100]*	20.6
Track	*nlfilt_do300*	40.9
Average		37.4

Each loop is executed many times in its application, each time with a possibly different number of iterations and access pattern. In our simulations, we perform all the executions. We perform two sets of simulations, namely one where caches are flushed between loop invocations and one where cache states are preserved across loop invocations. The results are within 2% of each other, which implies that the transient regime is negligible. The results reported correspond to the first set of simulations.

Ocean is executed 4129 times, with 32 iterations most of the times. It has a small working set, namely 258 * 64 complex array elements. Data is accessed with different strides in different executions of the loop. We use the non-privatization algorithm for the software and hardware schemes. Since there is good load balance, we use the processor-wise test for the software scheme.

P3m is executed only once, with 97,336 iterations. The loop has a very large working set, with several arrays needing the privatization algorithm for the software and hardware schemes. In the latter, we use the BPA. The array elements are 4 bytes. The load is highly imbalanced and, therefore, dynamic scheduling is required.

Adm is executed 900 times, with 32 or 64 iterations in each case. The working set is small, although it has some arrays that need the non-privatization scheme and some that need the privatization scheme (the APA for the hardware). The array elements are 8 bytes. Since there is not much load imbalance, we use the processor-wise test for the software scheme.

Finally, *Track* is executed 56 times, with an average of 480 iterations per execution. The working set is small, and contains four arrays that use the non-privatization scheme. The array elements are 4 or 8 bytes. Of the 56 loop executions, 5 are not fully parallel and, as a result, the iteration-wise software scheme fails. However, the processor-wise software scheme passes and, despite the load imbalance of the iterations, delivers good performance. The plain dynamically-scheduled hardware scheme with iterations scheduled in small blocks passes all loops.

5.3 Environments Compared

We compare four environments: *Serial*, *Ideal*, *SW*, and *HW*. *Serial* is the uniprocessor execution of the loop, with all the data allocated in the memory local to the processor. The other environments correspond to the parallel execution of the loop. The loops are run with 16 processes except *Ocean* which, due to its small number of iterations and small working set, is run with 8 processes. Processes synchronize using locks and barriers. The pages of shared data are allocated round-robin across the different memory modules. We chose this allocation because these loops tend to have unpredictable reference patterns and relatively poor page locality. Private structures are allocated locally. Of the parallel environments, *Ideal* is the `doall` execution of the loop without any tests for correctness. The only overheads it includes are iteration scheduling and load imbalance. However, we select the scheduling that produces the fastest execution. Finally, *SW* and *HW* correspond to the software and hardware schemes respectively.

6 Evaluation

In our evaluation, we consider three issues, namely speedup in the execution of parallel loops, slowdown due to failure of the test, and scalability of the algorithm.

6.1 Loop Execution Speedup

The speedups of the *Ideal*, *SW*, and *HW* parallelization of the loops are shown in Figure 8. Recall that *Ocean* runs with 8 processors, while the rest of the loops run with 16. For the 16-processor runs, the average speedups delivered by *Ideal*, *SW* and *HW* are 9.5, 3.2, and 7.3 respectively. The relatively modest *Ideal* speedups are limited by the long latencies involved in accessing data from remote memories, the scheduling overheads, and the load imbalance. Comparing *SW* and *HW*, we see that, for all the loops, *HW* significantly outperforms *SW*. The speedups of the hardware scheme are good. In two out of four loops, the *HW* speedup is very close to that of *Ideal*.

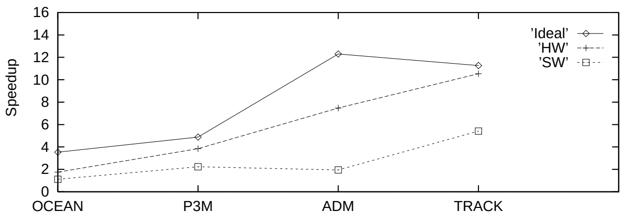


Figure 8: Speedups of the parallel executions of the loops. Recall that *Ocean* runs with 8 processors only.

To understand why we get these speedups, Figure 9 shows the execution time of the loops broken down into time executing the instructions (*Busy*), synchronizing at locks or barriers (*Sync*), or waiting for data from the memory system (*Mem*). The latter includes the negligible contribution of the stall due to write buffer overflow. For each loop, all bars are normalized to *Serial*. The labels for the bars have the number of processors tagged as a suffix for clarity reasons.

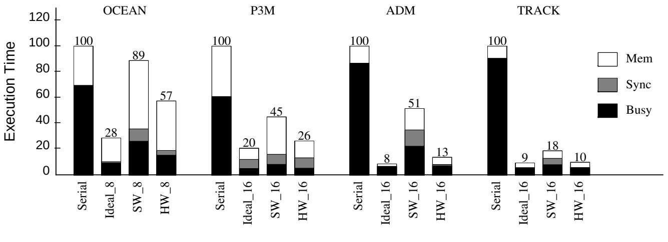


Figure 9: Execution time of the loops.

The figure shows that *HW* is an average of 50% faster than *SW*. This makes *HW* an attractive scheme. It has lower *Busy*, *Mem*, and even *Sync* time than *SW*. The lower *Busy* and *Mem* time are a result of the fewer operations executed by *HW*. As indicated in Section 3.4, the read-ins and the zeroing, marking, and analysis phases of *SW* are either performed in hardware or not performed at all, in *HW*. Note, however, that the time savings come more from better memory system performance (*Mem*) than from fewer instructions executed (*Busy*). Finally, the lower *Sync* time is both a side effect of the lower *Busy* and *Mem* times and, sometimes like in *Track*, a result of *HW*'s higher scheduling flexibility. In *HW*, the non-privatization algorithm is processor-wise (and, therefore, cheaper) even with dynamic scheduling. In *SW*, the non-privatization algorithm is either processor-wise and statically scheduled, or iteration-wise and dynamically scheduled. In the first alternative, load imbalance may boost *Sync* time, while in the second one, the large shadow arrays brought into the cache degrade cache performance and boost *Mem* time.

If we now compare *HW* to *Ideal*, we see that *HW*'s *Mem* time is the most important obstacle to higher speedups. In some loops like *Ocean*, *Ideal* already has a large *Mem* time because its memory references do not have much locality or reuse data much. In that case, it only gets worse in *HW*. In *P3m*, however, *Ideal* has little *Mem* time and the new operations added for parallelization in *HW* still induce *Mem* time.

To understand these trends, Figures 10 and 11 break down the execution time of *SW* and *HW* respectively into their component phases. For *SW*, Figure 10 starts with the *Ideal* bar and then gradually adds the contributions of the marking, initialization, zeroing, analysis and, if significant, conclusion phases. For *HW*, Figure 11 starts with the *Ideal* bar and adds the contributions of the marking, initialization and conclusion phases. For *HW*, the marking phase is simply the loop execution without initialization or conclusion phases. Both figures also include the *Serial* environment normalized to 100 for reference purposes.

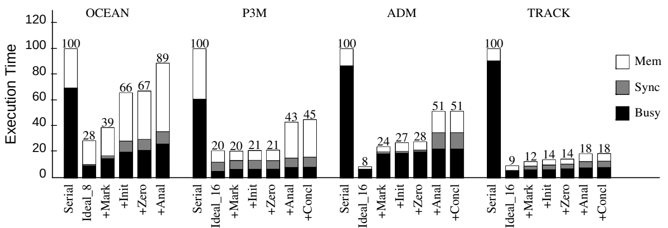


Figure 10: Phase by phase execution time breakdown of the software scheme.

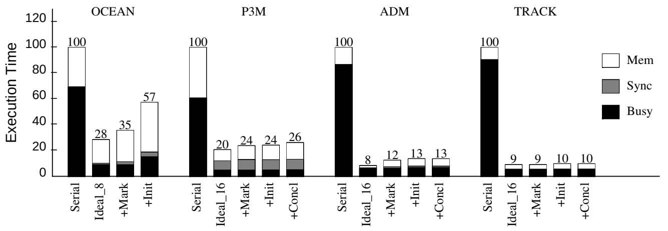


Figure 11: Phase by phase execution time breakdown of the hardware scheme.

Figure 10 shows that the overhead of *SW* comes largely from the analysis phase. In addition, for *Ocean*, the initialization phase also contributes significantly. For *HW*, Figure 11 shows that the only major overhead is the initialization phase in *Ocean*. The slowdown in the *SW* analysis phase is due to its merging step, where processors merge their semi-processed shadow arrays. Since each processor allocated its shadow arrays in its memory for locality, these arrays are spread across the machine. While this step is optimized and parallelized, the large number of remote memory accesses induced slows down the algorithm. This step of major exchange or combination of bookkeeping information is required in *SW*. In *HW*, this step is unnecessary because the bookkeeping information is exchanged together with the data, while the data is being accessed. This exchange of bookkeeping information causes more and longer cache coherence messages in *HW*. The result is the *+Mark* overhead in Figure 11. This overhead, however, is usually small. It is only significant in *Ocean*, where accessing the array under test accounts for most of the work in a loop iteration.

The overhead of *Ocean*'s initialization phase in both *SW* and *HW* is caused by the cache misses suffered while backing up the array under test. Interestingly, the overhead of array backup is negligible in all other loops. It is important in *Ocean* only because the work done by the loop is so tiny. Finally, in *SW*, a modest instruction overhead is induced by the marking phase in *Ocean* and *Adm*. On average, however, the instruction overhead is less important than the memory time overhead.

6.2 Slowdown Due to Failure

If the speculative parallel execution of the loop fails, both the hardware and the software schemes restore the arrays under test (in the non-privatization algorithm only) and re-execute the loop sequentially. As indicated before, in the hardware scheme, the failure is detected as soon as the dependence violation occurs. In the software scheme, however, the failure is detected only after the execution of the loop is completed.

To determine how important this difference is, we force the failure of one instance of each of our loops. For *P3m* and *Adm*, we do not privatize the arrays under test and run the non-privatization algorithm. The result is test failure. For *Ocean*, we insert a cross-iteration dependence between the first iterations executed by two processors. Finally, for *Track*, we simply run the iteration-wise tests on one of the loops that need the processor-wise tests to pass. We compare the execution times under the *Serial*, *SW* and *HW* schemes. The resulting execution time is shown in Figure 12. In the figure, the bars are broken down into the familiar categories. In addition, for *HW*, we add an extra category, namely *Intr*. *Intr* is the time taken by the cross-processor interrupt that signals the abortion of the parallel execution. As indicated in Section 5.1, this interrupt takes 30 microseconds. For each application, the bars are normalized to *Serial*.

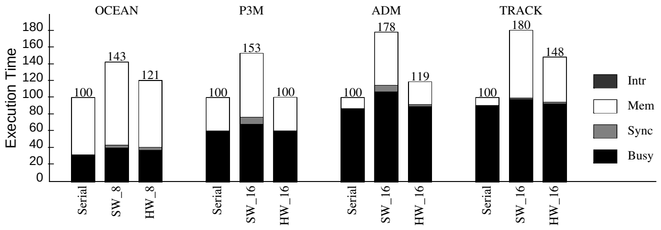


Figure 12: Execution time of one instance of each loop when it fails the test.

In the figure, the *Serial* bars correspond to the execution of a single loop. Therefore, they are not the same as in Figure 9. *SW* includes the parallel execution of the loop (including data backup), data restore, and serial execution. The latter takes longer to execute than *Serial* because the shared data is now distributed across several memory modules. Finally, *HW* includes the parallel execution of the loop until dependence detection (including data backup), a global synchronization step via a cross-processor interrupt, data restore, and serial execution.

The figure shows that, except in *Track*, *HW* does not take much longer than *Serial* to execute. There are several reasons for that. The first one is that *HW* detects the dependence violation soon. Ignoring *Ocean*, whose dependence was inserted artificially, the dependence violation is found after executing 1% of the iterations in *P3m* and *Adm*, and a higher 22% of the iterations in *Track*. In addition, the overhead of the cross-processor interrupt is small compared to the other times – it is hard to see the contribution of *Intr* in the figure. Finally, the other two sources of overhead, namely initialization with backup and data restore generally make only a modest contribution. Overall, therefore, *HW* performs acceptably when the test fails.

SW is quite slower than *Serial* for every single loop. On average, *SW* takes 63% longer than *Serial*. The reason is that it executes the loop to completion before identifying a possible dependence violation. Consequently, *SW* is not good at handling parallelization failure either.

6.3 Scalability

Finally, we consider the scalability of the software and hardware schemes. As the number of processors increases, the ***work per processor*** decreases in all the phases of these schemes except in the zeroing and analysis phases of the software scheme. In these two phases, the work per processor is kept constant. In the hardware scheme, the zeroing phase is performed in hardware and there is no analysis phase. As a result, the hardware scheme has better scalability than the software scheme.

Our experiments confirm this prediction. Figure 13 shows the speedups of our loops running on 8 and 16 processors. We do not show *Ocean* because, due to its small size, cannot run well with 16 processors. For each loop, we show curves for *Ideal*, *HW*, and *SW*. From the figure, we see that the *SW* curves saturate earlier than the *HW* ones. In fact, in *P3m*, the *SW* curve is lower for 16 processors than for 8 processors. The reason for the slowdown is that, with more processors, a larger fraction of the data is remote and there are more remote misses. The hardware scheme is less sensitive to this problem.

7 Related Work

The work most related is three schemes that have been proposed concurrently with ours [7, 11, 14] and which support speculative parallelization inside a multiprocessor chip. These schemes, proposed by Oplinger *et al* [11], Steffan and Mowry [14], and Gopal *et al* [7] are very similar to each other. The cache coherence protocol inside a chip is extended with versions or time stamps similar to ours. Parallelism is exploited by running one task (for example one loop iteration) on each of the processors on chip. One of the tasks is marked non-speculative, while the others are speculative with a certain order. Correct ordering of accesses is ensured by time stamping the data with the ID of the task that accesses the data. Tasks are scheduled for execution and committed in order. The data written by a speculative task is kept in the private cache or write buffer until the task becomes non-speculative. At that point, the updates can be merged with memory. Before that, the lines with speculative state must not be displaced from the cache or buffer. If a reference by a task would force the displacement, the processor stalls. Recovery from a wrong speculation is relatively simple: the cache lines with speculative data are invalidated and the task is restarted. Since on-chip fine-grain synchronization is relatively cheap, it is feasible to run codes with many dependences and exploit partial parallelism. Overall, these schemes are targeted to small-scale parallelism.

The scheme proposed is this paper is targeted to the larger-scale, coarser-grain parallelism exploited in DSM multiprocessors, and codes that are usually parallel. Our scheme does not distinguish between speculative and non-speculative tasks since they are all speculative. The updates issued by processors, instead of being temporarily buffered in caches until task commit time, are all merged with the state in memory. This eliminates any limitation on the working set of the tasks. In addition, it enables flexibility in task scheduling and eliminates the restriction of in-order completion. Tasks can potentially execute and complete in any order. Unfinished lower-numbered tasks do not prevent processors from completing higher-numbered tasks and starting others. All these operations are safe because the original data has been backed up in advance. However, if a dependence is detected, execution must be restarted from the latest checkpoint. This is because a higher-numbered task has already modified memory. Finally, no partial parallelism is exploited due to the cost of fine-grain synchronization in DSM machines. However,

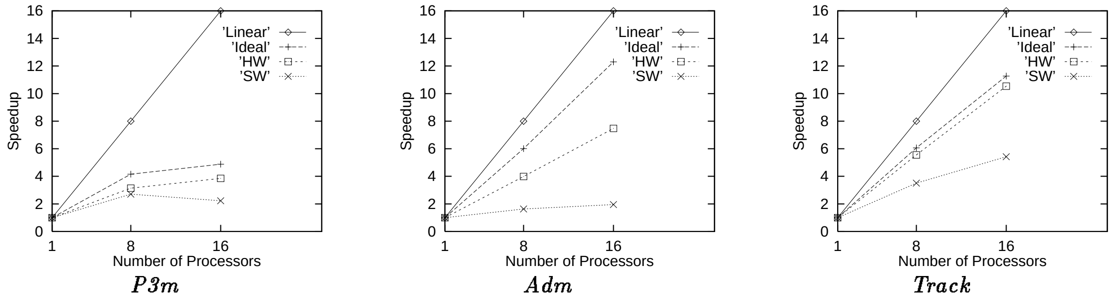


Figure 13: Speedup of the software and hardware schemes for 8 and 16 processors.

we apply parallelism-enabling transformations like privatization and hide intra-processor dependences with processor-wise tests, thus allowing a number of dependent loops to execute as doalls.

8 Conclusions

We have proposed a scheme to perform speculative run-time parallelization efficiently for DSM multiprocessors. The concept is to execute the code in parallel speculatively and use extensions to the cache coherence protocol hardware to detect any dependence violation. As soon as a dependence is detected, execution stops, the state is restored, and the code is re-executed serially. The scheme requires hardware extensions to the cache coherence protocol and memory hierarchy of a DSM. It operates with low overhead and allows great flexibility in the scheduling and completion order of tasks. In this paper, we presented the algorithms and a hardware design of the scheme. We showed that the scheme delivers average loop speedups of 7.3 for 16 processors and is 50% faster than a related software-only scheme.

Our hardware scheme can be easily integrated with reduction parallelization in the same way as the software scheme has been [13]. We have not presented the algorithm due to lack of space. The scheme can also be designed so that the data produced is regularly checkpointed to minimize the cost of failure. We are currently extending our work in two areas. The first one is to adapt it to support pointers well. Our design is already suited to very sparse, pointer-accessed data structures often used in C, because each individual memory line is associated with a short tag. This minimizing shadow memory consumption and eliminates the need of knowing the bounds of access regions. The second area is implementing the data backup phase in hardware. An efficient scheme can be designed with extensions to the scheme presented here.

Acknowledgments

We thank the referees, Sarita Adve, and members of the I-ACOMA group for their valuable feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[2] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[4] D. K. Chen, J. Torrellas, and P. C. Yew. An Efficient Algorithm for the Run-Time Parallelization of Do-Across Loops. In *Supercomputing '94*, pages 518–527, November 1994.

[5] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. M. Kinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope Parallel Programming Environment. *Proc. IEEE*, 81(2):244–263, February 1993.

[6] S. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[7] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[8] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[10] S.-T. Leung and J. Zahorjan. Improving the Performance of Runtime Parallelization. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

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

[12] L. Rauchwerger. Run-Time Parallelization: A Framework for Parallel Computation. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, September 1995.

[13] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[14] J. G. Steffan and T. C. Mowry. The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors. Technical Report CSRI-TR-350, Computer Science Research Institute, University of Toronto, February 1997.

[15] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime Compilation Methods for Multicomputers. In D. H. Schwetman, editor, *Proceedings of the 1991 International Conference on Parallel Processing*, pages 26–30. CRC Press, Inc., 1991. Vol. II - Software.

[16] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. Technical Report 1523, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, July 1997.