Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors*

Ye Zhang†, Lawrence Rauchwerger‡, and Josep Torrellas†
†Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801
y-zhang2, torrella@cs.uiuc.edu
‡Computer Science Department
Texas A&M University, College Station, TX 77843
rwerger@cs.tamu.edu

Center for Supercomputing Research and Development Technical Report 1523
July 1997

Abstract

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

The idea is to execute the code in parallel speculatively and use the cache coherence protocol hardware to flag any cross-iteration data dependence. Often, such dependences naturally trigger a coherence transaction, like an invalidation when a processor writes to a variable that was read by another processor. However, with appropriate extensions to the cache coherence protocol, all such dependences are detected. If a dependence is detected, execution stops, the state is restored, and the code is re-executed serially. This scheme, which we apply to loops, has low overhead and detects serial loops very quickly. We present the algorithms and a hardware design of the scheme. Overall, the scheme delivers a speedup of 7 for 16 processors and is twice faster than a related software-only scheme.

Keywords: scalable shared-memory multiprocessors, cache coherence protocols, run-time parallelization, speculative execution.

1 Introduction

While there has been much work on parallelism extraction at compile time for multiprocessors [3, 6, 9], 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, there is no information available at compile time. This is common in applications with irregular domains or interactions that use

---

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9619399 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from Intel and IBM.
subscripted subscripts (Figure 1-(c)). 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 of organic systems, 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 [4].

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 the extraction of an \textit{inspector} loop that analyzes the data access patterns ([5, 12, 13, 17] 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 \textit{executor}, with barriers separating the wavefronts. Unfortunately, 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 new framework for software run-time parallelization for multiprocessors [13, 14]. We apply the scheme to loops, but it can also be applied to other code. 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 values before the loop and the loop is re-executed serially.

This form of speculative execution is different from the commonly proposed for advanced uniprocessors. Indeed, much work has been done in uniprocessors trying to issue loads ahead of stores even though the addresses of the locations accessed are unknown [7, 10, 15, 16]. This type of speculation is orthogonal to our work. Our framework exploits parallelism across processors in a multiprocessor. It can use these traditional techniques within each thread. We are effectively adding a second dimension of speculation.

There are several advantages to our framework. A simple pass-fail test requires less computation time than that of a full schedule of the loop. A second advantage is that, by using speculation, the technique is made generally applicable. Unfortunately, the scheme has several shortcomings. One is its overhead, which includes bookkeeping necessary to save state during execution for the test at the end of the loop. A second shortcoming is that the loop must be completely executed before it can be determined whether or not it was fully parallel.

The goal of this paper is to address these shortcomings by performing the speculative run-time parallelization in hardware. We propose a new scheme that minimizes the run-time overhead and keeps the penalty of failure to a minimum by aborting the parallel execution as soon as a cross-iteration data dependence occurs. The idea behind this scheme is to use the cache coherence protocol hardware in the DSM to flag any cross-iteration data dependence. Often, such dependences naturally trigger a coherence transaction, like an invalidation when a processor writes to a variable that was read by another processor. With appropriate extensions to the cache coherence protocol, all cross-iteration dependences can be detected. Overall, the proposed scheme is very effective: it delivers an average speedup of 7 for 16 processors and is 50% faster than the software scheme.

This paper is organized as follows: Section 2 briefly outlines speculative run-time parallelization in software; Section 3 presents our proposal for speculative run-time parallelization in hardware; Section 4 presents a hardware design of our scheme; Section 5 discusses the methodology used to evaluate our scheme; and Section 6 presents the evaluation.
2 Speculative Run-Time Parallelization in Software

We have recently proposed a new algorithm that uses speculation to parallelize loops at run time \[13, 14\]. The algorithm is called the LRPD test. 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 \[1\] (or cross-iteration dependences). There are three possible 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 iterations of 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 no flow dependences but there are anti or output dependences, then 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.

\[
\begin{align*}
\text{do } & i = 1, n \\
A(i) & = A(i) + A(i-1) \\
\text{enddo} & \\
\text{do } & i = 1, n/2 \\
\text{tmp} & = A(2^i) \\
A(2^i) & = A(2^{i-1}) \\
A(2^{i-1}) & = \text{tmp} \\
\text{enddo} & \\
\text{do } & i = 1, n \\
A(f(i)) & = \ldots \quad (S1) \\
\ldots & = A(g(i)) + \ldots \quad (S2) \\
\text{enddo} & \\
\end{align*}
\]

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, consider the loop in Figure 1-(b). There is an anti dependence between statement \(S2\) of iteration \(i\) and statement \(S1\) of iteration \(i + 1\). This dependence can be removed by privatizing the temporary variable \(\text{tmp}\). Then, the loop can be executed as a doall loop.

In the following algorithm, we will consider an array privatizable if each of its elements behaves in either of two ways: either it is read-only or every read access to it is preceded by a write access 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 of the loop, 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/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

Clearly, 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. The first-time-written information can be kept by either hardware or software. Note that, in any case, the compiler only needs to save modifiable shared arrays; read-only and privatized arrays need not be saved. 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 found not parallel, we restore the arrays from their backups and re-start serial execution.

2.2.2 Algorithm to Detect Dependences: The LRPD Test

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 arrays that are either privatized or not. In the former case, the test has an extra step. If the arrays are privatized, they take more memory space, but there is a higher chance of finding the loop parallel. The compiler or the programmer can use heuristics to decide whether or not the arrays should be privatized.

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 back up 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 declare scalar \( Atw \) and initialize it 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 \( \text{any}(A_w[:] \land A_r[:]) \), that is, if the marked areas are common anywhere, then the loop is not a doall and the execution is aborted. If this condition is true, it means that an element \( A[i] \)

\footnote{\text{any} returns the "OR" of its vector operand's elements: \( \text{any}(v[1:n]) = (v[1] \lor v[2] \lor \ldots \lor v[n]) \).}
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 have to
assume conservatively the worst case of a flow dependence. Note that, if the iterations that
read the element also wrote it, \( A_w \) would not be set and this test would not flag any problem.
We will, however, detect this case later.

(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 are writing to the same element of \( A \). Consequently, there are
no dependences. The combination of tests (b) and (c) checks for all dependences.

(d) Else if any(\( A_w[:] \land A_{np}[:] \)), \( A \) 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 and, in a
different iteration or in the same iteration, is written. Again, we have to assume the worst
case, which is the reading iteration being preceded by the writing one.

(e) Otherwise, the loop was made a doall by privatizing the shared array \( A \). This is the case
when any given \( A[i] \) is either read-only or any read to it is preceded by write to it in the same
iteration.

Figure 2 shows an example of a loop. In practice, 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. 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.

\[
\begin{align*}
\text{do } i=1,5 & \\
& z = A(K(i)) \\
& \text{if } (B1(i) \text{ .eq. } \text{true.}) \text{ then} \\
& \quad A(L(i)) = z + C(i) \\
& \text{endif} \\
& \text{enddo} \\
\end{align*}
\]

\[
\begin{align*}
\text{do } i=1,5 & \\
& z = A(K(i)) \\
& \text{if } (B1(i) \text{ .eq. } \text{true.}) \text{ then} \\
& \quad A(L(i)) = z + C(i) \\
& \text{endif} \\
& \text{enddo} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>( Aw(,:) )</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>( Ar(,:) )</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>( Anp(,:) )</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>( Atw )</td>
<td>3</td>
</tr>
<tr>
<td>( Atm )</td>
<td>2</td>
</tr>
</tbody>
</table>

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

### 2.2.3 Improvements

We can parallelize more loops if we complicate the algorithm a bit more. Indeed, a loop is still
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. In this condition, the order of the iterations is important. For example, all the loops
in Figure 3 are parallel if the single array element accessed is privatized. To run these loops in
parallel, a processor may need to read the array element from the shared location into the privatized
location, work on the privatized location, and then copy the array element from the privatized to
the shared location before exiting the loop. These two copy operations are called \textit{read-in and copy-
out}. To parallelize these loops, it can be shown that the algorithm needs to use an extra shadow
array \( A_{wmin}[1:n] \) similar to the write shadow array.
Figure 3: Examples of iterations of loops that can be parallelized with a more complicated privatization algorithm.

Overall, our algorithm can be implemented efficiently as described in [13, 14]. For example, 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, the contents of the private shadow arrays are merged into global shadow arrays in parallel. This is called the merging phase.

The algorithm, as presented, detects cross-iteration dependences. We will 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 chunks 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 can potentially pass the processor-wise version of the test when data-dependent iterations are assigned to the same processor. Second, each entry in a shadow array now only needs to be 1 bit. These entries are accessed with bitmap operations, resulting in significant space savings. The disadvantage of using the processor-wise test is the constraint of static scheduling, necessary to insure chunks of contiguous iterations for each processor. This scheduling policy may cause load imbalance, severely affecting performance.

2.2.4 Integration with the Compiler

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 can use heuristics and statistics about the parallelization success-rate in previous executions and automatically decide when run-time parallelization can be profitable. If run-time parallelization is to be performed, the compiler inserts code to back up arrays, update the shadow arrays every time the arrays under test are accessed, perform the analysis and, if the analysis fails, restart the loop serially on one processor.

2.3 Shortcomings

An analysis of the approach presented reveals at least two important shortcomings. The first one is the overhead of the extra instructions necessary for the marking, merging, and analysis phases. For the loops where most of the work performed has 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. This implies that the slowdown due to failure may be significant.
3 Speculative Run-Time Parallelization in Hardware

To address the problems just described, 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

The idea behind this scheme is to make use of the cache coherence protocol hardware in the multiprocessor to flag any cross-iteration data dependences. Often, such dependences naturally trigger a coherence transaction. For example, when a processor writes to a variable that was read by a second processor, the first processor sends an invalidation or update to the second one. With appropriate extensions to the cache coherence protocol, all cross-iteration dependences can be detected. When a dependence is detected, the parallel execution is immediately aborted.

The extensions added to the cache coherence protocol are designed to be simple, minimize the increase in traffic, and add only a few simple data races. There are two sets of extensions, which we call the non-privatization and privatization algorithms. These extensions are used by non-privatized and privatized arrays under test respectively. Recall that, if the privatization algorithm is used, the arrays take more memory space but there is a higher chance of finding the loop parallel. In any case, the rest of the arrays in the program are unaffected by these extensions. We describe these two algorithms next.

3.2 Non-Privatization Algorithm

To keep the non-privatization algorithm simple, we use two main principles. First, we restrict the types of data accesses that loops can include to qualify as parallel. Specifically, a given element of the array under test must be either read-only ( RO only) or accessed by only one processor ( No Shr, for not-shared). A pattern where an element is read by several processors and later written by one is flagged as non parallelizable. 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 scheduling. The second principle is that the transactions added to the cache coherence protocol are designed so that they are all serialized in the directory. This is like the base coherence transactions. This approach keeps the data races to a minimum. However, it requires adding some fast memory in the directory to keep the state of the elements of the array under test.

For each element of the array under test, this fast memory keeps the following state. First, it keeps the RO only and No Shr bits, which tell whether the element is read-only or not-shared respectively. Before the loop starts, these bits are cleared. To help set these bits, the directory needs an extra field that keeps the ID of the processor that first accessed the element ( First ). If the first access is a write, both First and No Shr 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 RO only, while if it is a write by the same processor, we set No Shr. In all cases, a write to a RO only element will cause the failure of the parallelization. Similarly, a processor different from First trying to access to a No Shr 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,
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 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. Figure 5-(a) shows all the state necessary for this protocol per array element.

---

Figure 5: State necessary per array element to implement the non-privatization (Chart (a)), privatization without read-in/copy-out (Chart (b)) and privatization with read-in/copy-out (Chart (c)) protocols. There is a single set of hardware bits that is used differently depending on the algorithm used.

The bits in the tags of the different caches and directory are kept coherent. Since most accesses will not induce a change in the bits, the directory and the other caches do not need to be updated. Even if the bits are changed, however, if the associated cache line is dirty, no message needs to be sent to the directory. This is because, if the line is dirty, any other processor that references any elements in the line, will have to get the line from the owner cache. At that point, it can get an updated value of the tag bits as well. Of course, when a dirty line is displaced from a cache or is forced to be written back, the directory is updated with the state in the line tag for each element in the line.
We can now easily extend the algorithms in Figure 4 to include the changes in the state of the cache tags. The resulting algorithms are shown in Figures 6 and 7. In the figures, the state in the directory and in the line tags are 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.

There are several cases when the directory receives a dirty line from a cache. This occurs when a dirty line is displaced from a cache or is forced to be written back. In these cases, for each word in the line, the directory state is updated with the state in the line tag. This is shown in Figures 6 and 7.

The races in these transactions are handled as in a plain cache coherence protocol. No data inconsistency occurs because all transactions directed to the same cache 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 update $\text{dir.First}$ and $\text{dir.NoShr}$ for the element. The first message that arrives at the directory causes an invalidation to be sent to the other cache. When the second message reaches the directory, it will be bounced or buffered until the first transaction is completely finished. At that point, the second message is processed. It finds that the directory points to an owner cache, causes a write back from that cache, and finally finds that $\text{dir.NoShr}$ is already set. The result is the $\text{FAIL}$ statement of algorithm (d) in Figure 6.

A characteristic of this protocol is that reads can also cause races (algorithms (f), (g), and (h) in Figure 6). Again, no data inconsistency is possible because of the serialization in the directory. Indeed, if two processors read the same element from a cache line for the first time, both will send an update to the $\text{dir.First}$ field. The first message that reaches the directory will be processed (algorithm (f)). The second one will bounce and return to the sender cache. On arrival at the cache (algorithm (g)), it will set $\text{tag.First}$ to OTHER. It is also possible that, when the message arrives, it finds its own $\text{tag.NoShr}$ set. This means that the slower processor not only read but also wrote the data before knowing whether it was the $\text{First}$ processor to access the element. In that case, the algorithm fails. A different situation occurs when two processors try to set $\text{dir.ROnly}$ concurrently. Both send messages to the directory (algorithm (h)). The first one sets $\text{dir.ROnly}$. The second one can safely be ignored. There is no need to bounce it back to the sender because the sender's $\text{tag.ROnly}$ has already the correct value.

Finally, we note that, in transactions where read accesses cause a message to be sent to the directory (algorithms (f) and (h) in Figure 6) and the $\text{dir.First}$ or $\text{dir.ROnly}$ fields are updated, the update is not broadcast to all current sharers. The other sharers can still keep $\text{tag.First}$ set to NONE or $\text{dir.ROnly}$ set to 0. We do this simply to reduce message traffic. Such tag inconsistency does not cause problems because when any of these other sharers wants to update its $\text{tag.First}$ or $\text{tag.ROnly}$ fields, it will have to notify the home directory. At that point, if the directory detects a tag inconsistency, it will bounce the message and the fields will become consistent. Similarly, when the line is to become dirty in the cache after a write, algorithm (d) ensures that the fields become consistent.

### 3.3 Privatization Algorithm

In this algorithm, each processor works on a private copy of the array under test. In this paper, we choose to present an advanced version of the algorithm that supports read-in and copy-out as defined in Section 2.2.3 and parallelizes all the loops in Figure 3. The algorithm, therefore, is more aggressive that the software one from Section 2.2.2. While most of the loops do not need read-in or copy-out, we present this algorithm for generality. We will also outline how the hardware cost decreases if we do not handle read-in or copy-out.
if (Cache_hit)
    if (tag.First == OTHER && tag.Priv == 1)
        FAIL
    else
        Read from cache
        if (tag.First == NONE)
            tag.First = OWN
            if (cache_state != DIRTY)
                Send First_update to home
                /* do not bother if line is dirty */
            else
                if (tag.First == OTHER && tag.ROnly == 0)
                    tag.ROnly = 1
                    if (cache_state != DIRTY)
                        Send ROnly_update to home
            else
                Send read_req to home

          (a) Processor read

if (dir_state -- DIRTY)
    send writeback request to owner node
    wait for reply
    update dir.First, dir.Priv, and dir.ROnly
    /* update directory using the tag state of all the words of the dirty lines*/
    if (dir.First != RequestorID && dir.Priv == 1)
        FAIL
    else
        if (dir.First == NONE)
            dir.First = RequestorID
        else
            if (dir.First != RequestorID && dir.ROnly == 0)
                dir.ROnly = 1
        Copy dir state to tag state for all the words in the line
        Send line and tag to requester

        (b) Home receives read request

if (Cache_hit)
    if (tag.First == OTHER || tag.ROnly == 1)
        FAIL
    else
        if (cache_state == CLEAN)
            Send write_req to home
        else
            Write to cache
    else
        Send write_req to home
    tag.First = OWN /* no need to tell the directory*/
    tag.Priv = 1 /* no need to tell the directory*/

        (c) Processor write

Figure 6: Non-privatization algorithm in extended form (first part).
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.Priv, 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.Priv = 1
        Copy dir state to tag state for all the words in the line
        Send line and tag to requester

(d): Home receives write request

Update directory using the tag state of all
the words of the dirty line

(e): Home receives a dirty line displaced from a cache

if (dir.Priv == 1)
    FAIL /*race between a First_update and a write*/
else
    if (dir.First == NONE)
        dir.First = IncomingID
    else
        dir.ROnly = 1
        Send First_update_fail to the processor
        /*race between two First_updates*/

(f): Home receives a First_update from a processor

if (tag.First == OWN && tag.Priv == 1)
    FAIL /*race between two First_updates. This
     processor read and then later wrote*/
tag.First = OTHER
tag.ROnly = 1

(g): Cache receives a First_update_fail from directory

if (dir.Priv == 1)
    FAIL /*race between a ROnly_update and a write*/
else
    dir.ROnly = 1
    /*race between two ROnly_updates. No need to bounce any
    message: the second message that reaches dir is
    plainly ignored*/

(h): Home receives a ROnly_update

Figure 7: Non-privatization algorithm in extended form (second part).
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, our algorithm is very simple. The directory corresponding to the shared array under test keeps 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 ($MaxR1st$ field). The second one keeps the number of the lowest iteration executed so far by any processor that involved writing the element ($MinW$ field). The parallelization test will fail at any time when $MaxR1st$ is larger than $MinW$.

The algorithm minimizes races in the protocol messages by ultimately serializing all the transactions related to an array element in the directory for the shared copy of that element. In the simplest implementation, the size of $MaxR1st$ and $MinW$ is 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 has been executed. At synchronization points, the effective iteration number that would be stored in the time stamps is reset to zero.

Usually, the processor accesses privatized data. To avoid having to access these shared time stamps too frequently, the directories of the private copies of the array under test keep similar information for each processor. They keep, for each element, 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, $Read1st$ is set. The whole state is shown in Figure 5-(c).

The algorithm proceeds as follows (Figures 8 and 9). Every time that a processor reads an element of the array under test (algorithm (a)), 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 has been displaced from the cache, the state of the directory for the private array ($PMaxR1st$ and $PMaxW$ are both lower than the current iteration number. The current iteration number is $Curr_Iter$ in the figures). If the iteration is indeed read-first, the directory for the shared array is informed: if the current iteration number is larger than $MinW$, the parallelization fails; otherwise, $MaxR1st$ is set to 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 (algorithm (b)), 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 has been displaced from the cache, the state of the directory for the private array ($PMaxW$ is less than the current iteration number). If this is indeed 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 updated 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: if the current iteration number is lower than $MaxR1st$, the parallelization fails; otherwise, $MinW$ is set to the minimum of its current value and the current iteration number.

Finally, recall that the private copies of the array under test start-off uninitialized. Every time that a processor accesses a line for the processor's first time, it needs to read-in the line from the
if (cache_hit)
  read from cache
  if ( (Readlist == 0) && (Write == 0) )
    Readlist = 1     /*read-first*/
    send read-first signal to private directory
  else
    send read request to private directory

(a): Processor read

PMaxRlist = Curr_Iter
send read-first signal to shared directory

(b): Directory for the private array receives a read-first signal

if (PMaxRlist==PMaxW==0 for all the elem. in the mem. line)
  /* a read-in*/
  send read-in-req to shared directory
  copy the reply into private data
  PMaxRlist = Curr_Iter for the element requested
  send line to cache with tag.Readlist-1 for the elem. requested
else
  if ( (PMaxRlist < Curr_Iter) && (PMaxW < Curr_Iter) )
    /*read-first*/
    send read-first signal to shared directory
    PMaxRlist = Curr_Iter
    send line to cache with tag.Readlist-1 for the elem. requested
  else
    send line to cache

(c): Directory for the private array receives a read request

if (incoming Curr_Iter > MinW)
  FAIL
MaxRlist = max(MaxRlist, incoming Curr_Iter)

(d): Directory for the shared array receives a read-first signal

if (incoming Curr_Iter > MinW)
  FAIL
  send the line of the shared array
MaxRlist = max(MaxRlist, incoming Curr_Iter)

(e): Directory for the private array receives a read-in-req

Figure 8: Read transactions of the privatization algorithm.
if (cache_hit)
    write to cache
if (Write == 0)
    Write = 1  /* first write to elem in iter*/
    send first-write signal to private directory
else
    send write request to private directory

(f): Processor write

if (PMaxW == 0)
    /* first write to element in whole loop*/
    PMaxW = Curr_Iter
    send first-write signal to shared directory
else
    if (PMaxW < Curr_Iter)
        /* first write to element in iteration*/
        PMaxW = Curr_Iter

(g): Directory for the private array receives a first-write signal

if (PMaxW == 0)
    /* first write to element in whole loop*/
    if (PMaxRlst=PMaxW==0 for all the elem. in the mem. line)
        /* a read-in for write*/
        send read-in-req for write to shared directory
        copy the reply into private data
    else
        /* private data is valid*/
        send first-write signal to shared directory
    PMaxW = Curr_Iter
    send line to cache with tag.Write=1 for the elem. requested
else
    if (PMaxW < Curr_Iter)
        /* first write to element in iteration*/
        PMaxW = Curr_Iter
        send line to cache with tag.Write=1 for the elem. requested

(h): Directory for the private array receives a write request

if (incoming Curr_Iter < MaxRlst)
    FAIL
MinW = min(MinW,incoming Curr_Iter)

(i): Directory for the shared array receives a first-write signal

if (incoming Curr_Iter < MaxRlst)
    FAIL
    send the line of the shared array
MinW = min(MinW,incoming Curr_Iter)

(j): Directory for the shared array receives a read-in-req for write

Figure 9: Write transactions of the privatization algorithm.
shared copy of the array. This is done by the protocol engine that manages the directory for the private array. Indeed, if the element requested by the processor has the $PMaxR1st$ and $PMaxW$ fields set to zero, and all the other words of the line do as well, the protocol engine issues a read to the shared array (algorithms (c) and (h)). Conversely, if the privatized array is live after the loop, the last values written to the private copies need to be copied-out to the shared array. This requires that, as execution progresses, we keep, for each element of the shared array, the last value written so far, together with the iteration number when the write occurred. For simplicity, we do not include this operation in Figures 8 and 9.

It can be shown that, if neither read-in nor copy-out need to be supported, the hardware requirements of the algorithm decrease significantly. Specifically, the state in the directories reduces to only two bits. The complete state requirements are shown in Figure 5-(b). This simple support is all that is needed in the large majority of parallelizable loops, where read-in and copy-out are unnecessary.

### 3.4 Comparing the Software and Hardware Schemes

From the previous discussion, we can see that the hardware scheme has several advantages over the software scheme. The first one is that failure to parallelize is detected on the fly as soon as the dependence occurs. A second advantage is that it has less instruction overhead. Indeed, the software algorithm needs extra instructions to update the shadow arrays in the marking phase and to perform the analysis phase. These instructions consume cycles and, in addition, can hurt cache performance because they increase the code size. A third advantage is that the hardware scheme has better scalability with the number of processors than the software scheme, especially in the case of the privatization test. This is because the total amount of work involved in the analysis phase of the software scheme increases with the number of processors. The hardware scheme does not have this problem because there is no analysis phase.

A fourth advantage of the hardware scheme is that it needs less overhead state than the software scheme for similar iteration-wise tests. As indicated in Sections 2.2.2 and 2.2.3, the software scheme requires, per array element, 3 time-stamps for the shadow locations (if read-in is not supported) or 4 time-stamps (if read-in is supported). The hardware scheme, according to Figure 5, requires the maximum of 2 and $2+\log(Proc)$ bits (if read-in is not supported) or the maximum of 2 time stamps and $2+\log(Proc)$ bits (if read-in is supported). We need the maximum of two quantities because we need to support both the non-privatization and the privatization protocol. Finally, the fifth advantage of the hardware scheme is that, as we will see in the next section, the non-privatization test is processor-wise without requiring static scheduling.

Of course, the major disadvantage 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 details of our scheme 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 to handle the fact that, within the same loop, different arrays use different algorithms. The hardware holding the tag and directory state (Figure 5) is the same in all cases, although it is used differently. One possible approach is to have the compiler replace the loads and
stores with special instructions for each of the three types of algorithms, namely non-privatization, privatization, and plain. This solution, while simple, requires changes in the instruction set. A better approach is to have a simple address-range comparator for the various arrays that decides the type of protocol to be employed based on the address of the array. The compiler inserts system calls that load and unload the comparator appropriately. If the compiler cannot decide the best strategy, then it can apply the most general test, namely privatization with read-in and copy-out.

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 both algorithms clear the tags at the beginning of the loop, the privatization algorithm additionally clears them at the beginning of each iteration. To clear some tags selectively, we use memory allocation. Specifically, we allocate privatized and shared arrays in different physical address ranges. We qualify a reset line with some address bits to clear only tags of lines with privatized data. We perform such a reset with a system call.

In addition, we need to clear the tag state of all the lines in the cache at the beginning of the loop. This is easily done with a general reset signal. Note, however, that the processor must not context switch between two processes that are trying to parallelize two different loops. Otherwise, clearing all the cache tags could be erroneous. Finally, we note that the tags should be cleared in both the primary and secondary caches. This can again be done with a system call.

Clearing the directory tags is easier. Both the non-privatization and the privatization algorithms need to clear the tags at the beginning of the loop. To perform this operation, we may use a system call. Again, for simplicity and correctness, the processor must not context switch between two processes that are trying to parallelize two different loops.

A third issue is how to reduce the space and protocol overheads of the algorithm. Many of the applications that we parallelize use double or quad words as their data types. Unfortunately, we need to keep the bits for each word for applications that use words as their data types. 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.

Another way to save space is to store the state shown in Figure 5 in a dedicated memory that is close to the directory and is accessed at the same time as the directory. With this scheme, we do not waste bits in the directory tags for data that uses the plain cache coherence protocol. This is the scheme that we use in the hardware design of the next section.

Finally, another way to reduce the space requirements is to use only 1 bit for the $PmazR1st$ field and 1 bit for the $PMazW$ field in the directory of private data for the privatization protocol. These fields are used like the Read1st and Write fields of the cache tags in the same protocol. In particular, they are cleared at the beginning of each iteration. In addition, we need one extra bit, WriteAny, that is set if the element has been written in any of the iterations executed so far. This bit is not cleared every iteration. With these three bits, we can build a protocol that has no more messages than the one with $PmazR1st$ and $PMazW$.

The protocol overhead is easier to reduce. For the privatization algorithm, 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 of the privatization algorithm. Specifically, the cache tags need to be cleared only at the beginning of the superiteration. The size of the time stamps decreases because there are fewer effective iterations. The number of read-first iterations and, in general, the number of messages and protocol tests decreases. A very similar result can be accomplished with loop unrolling. The disadvantage is possible load imbalance.

At the extreme, the superiterations are so large that each processor is assigned only one. This
scheme we called processor-wise in Section 2.2.3. The overhead of the privatization algorithm reduces further. The directory of the private array needs to keep only two bits per element, namely \textit{Read1st} and \textit{Write} like the cache tags. The time stamps in the directory of the shared array have reduced so much that the \textit{MaxR1st} and \textit{MinW} become the processor ID. Many messages and protocol tests are eliminated. The shortcoming is that the static scheduling may cause high load imbalance.

None of the previous two optimizations apply to the non-privatization algorithm. 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.

### 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 \textit{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.

![Hardware Diagram](image)

**Figure 10:** Hardware for speculative run-time parallelization.
The new design for the primary cache is shown in Figure 10-(a). The cache-related access bits corresponding to the lines in the primary cache are 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 the algorithms. The actual operation performed is determined by the Control signals, which identify whether the processor is performing a read or a write and the type of algorithm being used. The test logic is simple enough to generate the new access bits and a signal indicating whether the test failed at the same time as the tags comparison is done. If the new access bits are different than the old ones, they need to be saved in the next cycle. Specifically, they are stored in the access bit array and, if the corresponding cache line is not exclusive in the primary cache, the access bits are also propagated down the memory hierarchy to the secondary cache and directory. Overall, therefore, if the access bits are not modified, we perform the complete test while the cache is being accessed.

For the secondary cache, we also need to provide an access bit array (Figure 10-(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, they are stored in the access bit array of the primary cache.

Finally, Figure 10-(c) shows the new directory hardware for the non-privatization algorithm. Only minor changes are needed for the privatization algorithm. In the figure, the address coming from the processor is translated in the Translation Table to index an access bit array. Recall from Section 4.1 that we put the access bits in a dedicated memory with many fewer entries than entries in the directory. The translation table helps identify the access bits required. The translation table is loaded at the beginning of the program with information about the arrays under test allocated in the memory of that node. Specifically, each entry in the translation table contains, for a given array under test, its physical address boundaries, its data type, and a pointer to the beginning of its access bits in the access bit table. Consequently, given an address, the translation table can point to the corresponding access bits. The rest of the figure is similar to the previous figures. Overall, these operations are partially or totally overlapped with a directory and memory access.

Figure 10-(c) corresponds to a directory without a protocol processor. If there is a protocol processor, the test logic and part of the functions of the translation table are replaced by the protocol processor.

5 Experimental Setup

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

5.1 Simulation Environment

Our evaluation is based on execution-driven simulations of a CC-NUMA shared-memory multiprocessor using Tangolite [8]. 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]. For the loops that we want to parallelize at run time, Polaris inserts all the instructions to perform the marking and analysis phases.

The architecture modeled has 200-MHz 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. We selected such small caches because the only workloads that we can run have smaller working sets than real-life applications. The caches are kept coherent with a DASH-like cache coherence protocol [11]. 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 abstracted away as a constant latency. The 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 on average respectively. These figures correspond to an unloaded machine; they increase with resource contention. Processors do not stall on write misses.

5.2 Workloads

Since running long applications is not feasible with our limited resources, we prove our hardware algorithm by examining four loops from four Perfect Club applications [2]. These are loops that Polaris cannot analyze at the time of writing this paper and whose parallelization, therefore, can only be done at run-time. They account for a significant fraction of the time after the application has been parallelized by Polaris. These loops are *ftrvmt.do109* from Ocean, *pp.do100* from P3m, *run.do20* from Adm, and *nffilt.do300* from Track. In this paper, we identify the loops with the name of the application they belong to.

In the applications, each loop is executed many times, each time with a possibly different number of iterations and input data. In our simulations, we perform all the executions. To mimic real conditions, we flush the caches after every execution. Then, when we report results for the loop, we give the average of all the executions.

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, of which we simulate 15,000. The loop has a very large working set, with several arrays needing the privatization algorithm for the software and hardware schemes. The array elements are 4 bytes. No read-in or copy-out is necessary. The load in the different iterations 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 schemes and some that need the privatization schemes. The array elements are 8 bytes. Since there is good load balance, 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 schemes. The array elements are 4 or 8 bytes. The fraction of accesses to these arrays changes from 0% to 44% of all accesses. Interestingly, 5 of the 56 loop executions are not fully parallel and, therefore, the iteration-wise software scheme fails. However, if we use the processor-wise software scheme, the test passes. Unfortunately, there is load imbalance. The plain dynamically-scheduled hardware scheme passes all loops if the iterations are scheduled in blocks of a few iterations each.

We run all the loops with 16 processes. The exception is Ocean which, due to its small number of iterations and working set, is run with 8 processes. The pages of workload data are allocated round-robin across the different memory modules. Processes synchronize using locks and barriers.
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. In the analysis, we compare four scenarios, namely Serial, Ideal, SW, and HW. Serial is the uniprocessor execution of the loop, where all the data is allocated in the memory local to the processor. Ideal corresponds to the doall execution of the loop without any tests for correctness. It includes iteration scheduling overheads and load imbalance but we select the scheduling that produces the fastest execution. The shared data is distributed across the different memory modules. Finally, SW and HW correspond to the software and hardware schemes respectively.

6.1 Loop Execution Speedup

The speedups of the Ideal, SW, and HW parallelization of the loops are shown in Figure 11. Recall that Ocean runs with 8 processors, while the rest of the loops run with 16. If we divide the speedup by the number of processors to get the efficiency, we see that the Ideal system has an efficiency of 0.4-0.8. This rather low efficiency is the result of the longer latencies involved in accessing data from remote memories, scheduling overheads, and load imbalance. Clearly, the HW and SW systems cannot do any better than that. As it is, HW has an efficiency of 0.2-0.5, which corresponds to an average speedup of 6.7 for 16 processors. SW in turn has an efficiency of 0.1-0.3, which corresponds to an average speedup of 2.9 for 16 processors. We see that, for all the loops, the hardware scheme significantly outperforms the software scheme. The speedups of the hardware scheme are good. They are half-way between those of the software scheme and the ideal system.

![Figure 11: Speedups of the different parallel executions of the loops. Recall that Ocean runs with 8 processors only.](image)

To understand why we get these speedups, Figure 12 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). For each loop, the figure shows the four scenarios, all normalized to Serial. The labels for the bars have the number of processors tagged as a suffix for clarity.

The figure shows why HW performs better than SW. On average, HW is about 50% faster than SW. The hardware scheme has lower Busy, Mem, and sometimes even lower Sync time than the software scheme. The lower Busy time is the result of the fewer extra instructions that the hardware scheme adds. Indeed, the software scheme adds instructions to backup the arrays, clear the shadow arrays, perform the marking, merging, and analysis phases and possibly copy-out. The hardware scheme, instead, only needs to perform the backup and copy-out.

These extra instructions needed by the software scheme are responsible for the larger Mem time. They cause misses directly and they induce conflicts in the cache creating misses indirectly. The hardware scheme suffers much less from this effect. Finally, the hardware scheme has sometimes...
less *Synch* time. An example is *Track*. The reason is that *HW* has more scheduling flexibility. In the case of *Track*, *SW* must schedule the iterations statically to pass the test (see Section 5.2). *HW*, instead, can schedule small blocks of a few iterations dynamically.

If we now compare the *SW* and *HW* schemes to the others, we see that the largest overhead of the former schemes is usually a large *Mem* time. In the case of *Ocean*, *Ideal* already has a large *Mem* time because the memory accesses do not have much locality. As a result, it only gets worse in *SW* and *HW*. In *P3m*, however, *Ideal* has little *Mem* time; the new operations are responsible for the *Mem* time in *SW* and *HW*. Instruction overhead also has an important effect. In the applications that suffer high instruction overhead (*Ocean* and *Adm*), accesses to the arrays under test constitute a large fraction of the loop’s work. Finally, synchronization is not an important source of overhead in the loops considered.

### 6.2 Slowdown Due to Failure

If the speculative parallel execution of the loop fails, both the hardware and software schemes copy the backed up versions of the arrays under test to the arrays operated upon and re-execute the loop sequentially. As indicated before, in the hardware scheme, the failure is detected as soon as the cross-iteration dependence 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 on the first instantiation of the loop. The result is test failure. For *Ocean*, we insert a cross-iteration dependence between iterations 1 and 2 and run the first instantiation of the loop. Finally, for *Track*, we simply run the iteration-wise tests on the loop instantiation that needs processor-wise tests to pass. We compare the execution times using the *Serial*, *SW*, and *HW* schemes. The resulting execution time is shown in Figure 13. In the figure, the bars are broken down into the three familiar categories. For each application, the bars are normalized to *Serial*.

The figure shows that, except in *Track*, *HW* takes only a bit longer than *Serial*. On average for all the loops, *HW* takes 22% longer than *Serial*. The *SW*, instead, is quite slower than *Serial* for all the loops. On average, *SW* takes 58% longer than *Serial*. To understand these figures, we recall how the three execution times are related. The *HW* execution time includes the parallel execution up to when the dependence is detected (including data backup), plus the data restore overhead, plus the *Serial* time. The *SW* execution time includes the complete software parallel execution (including data backup), plus the data restore, plus the *Serial* time.

The difference between *HW* and *Serial* shows that, usually, the *HW* scheme takes relatively little
time detecting the failure and restoring the data. Furthermore, the difference between SW and HW shows that, usually, the parallel execution of the loop in software takes relatively significant time. The exception to these two observations is Track, where backing up and restoring the 4 arrays is a large overhead compared to the execution time of the loop itself.

6.3 Scalability

Finally, we consider the scalability of the software and hardware schemes. The software scheme has the following operations: array backup, shadow array zero-out, marking, merging-analysis, and data copy-out. As the number of processors increases, the work per processor decreases in all these operations except for shadow array zero-out and merging-analysis. For these two operations, in the privatization scheme, the work per processor is kept constant. In the hardware scheme, the shadow array zero-out is performed in hardware, and there is no merging-analysis. For this reason, the work per processor always decreases. As a result, the hardware scheme has better scalability than the software scheme.

Our experiments confirm this prediction. Figure 14 shows the speedups for our loops running on 8 and 16 processors. We do not show Ocean because, due to its small size, it cannot run with 16 processors. For each loop, we show curves for the ideal, software, and hardware schemes. From the figure, we see that the software scheme has lower speedups than the hardware scheme and its curves saturate earlier. In fact, in P3m, the curve for the software scheme 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.

Figure 13: Execution time of the loops when they fail the test.

Figure 14: Scalability of the software and hardware schemes.
7 Conclusions

We have proposed a new hardware support for efficient run-time parallelization of codes in DSMs. The idea is to execute the code in parallel speculatively and use the cache coherence protocol hardware to flag any cross-iteration data dependence. Often, such dependences naturally trigger a coherence transaction, like an invalidation when a processor writes to a variable that was read by another processor. However, with appropriate extensions to the cache coherence protocol, all such dependences are detected. If a dependence is detected, execution stops, the state is restored, and the code is re-executed serially. This scheme, which we have applied to loops, has low overhead and detects serial loops very quickly. These are its two main advantages over previously-proposed schemes.

In the paper, we presented the extensions required in the cache coherence protocol to support the scheme. In addition, we showed a hardware design of the scheme. Finally, we evaluated its performance. The scheme delivers a speedup of 7 for 16 processors. Compared to the software-only scheme, the proposed scheme is twice faster, detects failure earlier, and is more scalable. Currently, we are improving the design so that it handles some common types of loops faster.

References

Dependence Analysis for Supercomputing.  
Advanced Program Restructuring for High-Performance Computers with Polaris.  
Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs.  
An Efﬁcient Algorithm for the Run-Time Parallelization of Do-Across Loops.  
The PameScope Parallel Programming Environment.  
Dynamic Memory Disambiguation Using the Memory Conﬂict Buffer.  
Simulation of Multiprocessors: Accuracy and Performance.  
Maximizing multiprocessor performance with the SUIF Compiler.  
Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation.  
The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor.  
Improving the Performance of Runtime Parallelization.
The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization.
The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization.
Multiscalar Processors.
The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation.
In Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT ’96), 1996.
Runtime Compilation Methods for Multicomputers.
Vol. II - Software.