Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors

Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas
University of Illinois and Texas A&M University
y-zhang2, torrella@cs.uiuc.edu, rwerger@cs.tamu.edu

Abstract

Speculative parallel execution of non-analyzable codes on Distributed Shared-Memory (DSM) multiprocessors is challenging due to the long-latency and distribution involved. However, such an approach may well be the best way of speeding up codes whose dependences cannot be compiler analyzed. In previous work, we suggested executing the loop speculatively in parallel and adding extensions to the memory hierarchy hardware to detect any dependence violation. If the violation occurs, execution is interrupted, the variables are restored, and the code is re-executed serially. The scheme is targeted to loops where most of the invocations turn out to run in parallel without any dependence violation.

In this paper, we present a more advanced scheme for the speculative parallel execution of loops that have a modest number of cross-iteration dependences. In this case, when a dependence violation is detected, we locally repair the state. Then, we restart parallel execution from that point on. We call the general algorithm the Sliding Commit algorithm. If the loop dependences are of the special form of reduction, we use a specialized algorithm. Simulations indicate significant speedups relative to sequential execution. Finally, we propose hardware for optimizing reductions and obtain very good experimental results.

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

1 Introduction

Automatic parallelization of codes by the compiler has advanced significantly in this decade [3, 7, 10]. Unfortunately, there is still a large body of potentially parallel codes that compilers cannot parallelize because they can not fully analyze the codes’ dependence structure. This is because the dependence structure may be too complicated to analyze for current compiler technology or simply not available at compile time. The latter happens, for example, when the dependences depend on the actual input data to the program. In any case, the result is that the code is forced to run sequentially.

Consider, as an example, the loop shown in Figure 1, where arrays $f$ and $g$ depend on the input data. The compiler cannot let the execution of the iterations proceed in parallel because there may be a cross-iteration dependence: two different iterations may access the same array element and one access may be a write. Consequently, the code executes serially.

Unfortunately, these types of codes are common in many application domains, including sparse matrix computations, domain decomposition, molecular dynamics, molecular biology and image processing. Furthermore, many times, these codes are fully parallel or have only a few cross-iteration dependencies. If these codes could be run fully or partially in parallel in an effective manner on Distributed Shared-Memory (DSM) multiprocessors, some important codes would benefit significantly.
do $i = 1,n$
A((f(i))) = ...
... = A(g(i))
enddo

Figure 1: Example of a loop that cannot be analyzed by the compiler.

To run these codes in a partially parallel manner, some software approaches have been proposed. These schemes use information available at run time to construct execution schedules that are partially parallel. The right schedule is forced with direct synchronization. These methods are often based on an inspector loop that analyzes the data access patterns ([4, 16, 19, 22] to name a few). If the loop is not fully parallel, 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 computationally expensive and have side-effects.

Recently, we have introduced a new framework for run-time parallelization in hardware [24]. The scheme is based on a software-based run-time parallelization scheme that we proposed earlier [20]. The idea is to execute the code (loops) speculatively in parallel. As parallel execution proceeds, extra hardware added to the memory hierarchy detects if there is a dependence violation. If such a violation occurs, execution is interrupted, the variables are restored to their values at the most recent checkpoint, and the code is re-executed serially from that point. Checkpoints are typically done before the speculative execution of the loop is started.

While such a scheme works well if there is no dependence violation, it is costly if there is at least one dependence violation. In that case, all the work since the last general checkpoint is wasted, and the loop is re-executed serially.

In this paper, we extend such a mechanism to effectively handle loops with a modest number of cross-iteration dependencies. We design our algorithms so that, when a dependence violation is detected, we locally repair the state. Then, we restart parallel execution from the point where the dependence violation occurred. We call the general algorithm the Sliding Commit Algorithm (SCA). If the loop dependences are of the special form of reduction, we use a specialized algorithm.

This paper is organized as follows: Section 2 briefly describes the speculative parallelization scheme that was introduced in [24]; Section 3 presents the new speculative parallelization algorithms for loops with cross-iteration dependencies and reduction optimizations; Section 4 evaluates these new algorithms, and Section 5 discusses related work.

2 Speculative Parallelization in Hardware for DSM

In [24], we proposed a scheme for non-analyzable loops to execute speculatively in parallel in a DSM machine. The idea is to extend the directory-based cache coherence protocol of the machine to detect dependencies across iterations in hardware. Before the loop executes, we back up the global variables that can be modified in the loop (array under test). Then, the iterations are distributed to the processors and executed in parallel. If the hardware detects a dependence, a cross-processor interrupt is sent to all processors to stop execution. Then, the array under test is restored and the loop is retried sequentially. If the hardware does not detect any dependence, parallel execution completes successfully. For loops that are usually fully parallel, the scheme gets good speedups.

The scheme can be fleshed out into different algorithms depending on the cost-performance
design point desired. First, there are the non-privatization algorithms and the privatization algorithms. The latter are more costly, but are able to parallelize more loops. Among the privatization algorithms, the Advanced Privatization Algorithm (APA) is usually the most effective [24]. Among the non-privatization ones, the most effective one is the Advanced Non-Privatization Algorithm (ANPA), which was not presented in [24] for space reasons. In the rest of this section, we summarize the ANPA and APA. We assume the general case of a dynamically-scheduled loop. See [24, 23] for details.

2.1 Advanced Non-Privatization Algorithm (ANPA)

We use uncachable extra memory in the distributed directories to keep two arrays, MaxR and MaxW. Each of these arrays has as many entries as, and is distributed across nodes in the same way as the array under test. Each MaxR element keeps the ID of the highest iteration that has read the corresponding element of the array under test so far; each MaxW element keeps the same information for writes. These arrays are operated upon by the directory controller only. In addition, each processor keeps the ID of the iteration that it is working on, namely Curr_Iter.

Ideally, when a processor accesses an element of the array under test, a message forces the directory to access the corresponding MaxR or MaxW entry. The directory takes the processor's Curr_Iter from the message, and executes the algorithms of Figure 2-(a) and 2-(b). These algorithms ensure that the elements of the array under test are not accessed in the wrong order. If they are, a FAIL interrupt is broadcast.

if ( Curr_Iter < MaxW )
    FAIL /* WAR violation */
else
    read data
    MaxR = max( Curr_Iter, MaxR ) /*if RAR*/

(a): Processor read.

if ( Curr_Iter<MaxR || Curr_Iter<MaxW )
    FAIL /* RAW or WAW violation */
else
    write data
    MaxW = Curr_Iter

(b): Processor write.

Figure 2: Compact form of the ANPA.

If two writes to the same location from different iterations are performed out-of-order, we could skip the second update. This would eliminate WAW violations. If we do this, we must skip the write to the array under test proper. However, we do not do this here because, as shown in Section 3, such an optimization is not possible if we want to recover from dependence violations.

In the most general implementation, the size of the MaxR and MaxW elements is equal to the logarithm of the number of loop iterations. We can reduce the overhead by synchronizing all processors and resetting the stamps after a fixed number of iterations have been executed.

To reduce the cost of having to access MaxR and MaxW, the cache tags keep two extra bits per cache line word. These bits filter out many of these accesses. They are the Read and the Write bits. They are set by the hardware the first time in the iteration that the corresponding element is read or written respectively. Cache reads to words with the Read bit set do not generate directory accesses. The same occurs for writes. The Read and the Write bits of the whole cache are cleared in hardware and the beginning of every iteration. Finally, when a line is displaced from the cache, its Read and Write bits are lost. Figure 3 shows the complete read algorithms. The write algorithms are shown in [23].
if( Cache_hit )
  read data from cache
if( Read = 0 )
  /* first read in current iteration */
  send Read_Test_Req(Curr_Iter) to home
else /* cache miss */
  send Read_REQ(Curr_Iter) to home
Read = 1

(a): Processor read.

if( Curr_Iter < MaxW )
  FAIL
else
  if( Curr_Iter > MaxR )
    MaxR = Curr_Iter

(c): Home Dir receives Read_Test_Req(Curr_Iter).

if( Curr_Iter < MaxW )
  FAIL
else
  if( dir state is DIRTY )
    send Net_Read_Req to owner
    wait for Net_Read_Resp
    MaxR = max( Curr_Iter, MaxR )
    send Read_Resp(data) to requesting proc

(b): Home Dir receives Read_Req(Curr_Iter).

Figure 3: Read transactions for the ANPA in extended form.

2.2 Advanced Privatization Algorithm (APA)

In this algorithm, the loop is still dynamically scheduled, but each processor works on a private copy of the array under test. To understand the algorithm, we define read-first iterations. Given an array element, if an iteration reads it before the same iteration may write it, we call the iteration a read-first iteration for the element. A privatized loop is fully parallel if each of the elements of the array under test falls into one of the following: (i) it is read-only, (ii) every read of it is preceded by a write to it in the same iteration or (iii) a read-first iteration for the element is not preceded by any iteration that writes the element. The latter we will call a writing iteration. Figure 4 shows examples of parallel loops.

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rd</th>
<th>Rd</th>
<th>Wr</th>
<th>Wr</th>
<th>Rd</th>
<th>Wr</th>
<th>Rd</th>
<th>Rd</th>
<th>Wr</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Figure 4: Examples of loops that can be parallelized. All reads and writes access the same element.

We detect a failure when we identify a read-first iteration with a higher ID than a writing iteration for the same element. This is irrespective of the order of execution. These two iterations we call a pair if, in the sequential order, there is no intervening iteration that writes the element. A writing iteration may belong to more than one pair (Figure 6-(a)). When the second iteration of a pair is executed (irrespective of the order), the algorithm must detect a failure.

The algorithm needs state in the directories of both the shared array and its private copies. 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 writing iteration executed so far by any processor (MinW). The parallelization fails when MaxR1st is larger than MinW.

During 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 highest read-first iteration for the element executed so far by the processor (PMaxR1st, where P stands for private),
and the *highest writing* iteration for the element executed so far by the processor ($PM_{ax W}$).

Finally, to reduce overhead, the tags of the caches keep a summary of the private directory state. For each element, they keep 2 bits to indicate whether the current iteration is *read-first* for the element ($Read_{1st}$), and *writing* ($Write$). $Read_{1st}$ and $Write$ are cleared at the beginning of each iteration. When an iteration reads an element of the array under test, if both $Read_{1st}$ and $Write$ are zero, it sets $Read_{1st}$.

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 $Read_{1st}$ and $Write$ are zero) or, if the line is not in the cache, the state of the directory for the private array ($PM_{ax R1st}$ and $PM_{ax W}$ 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, and the shared data is copied into the private array. 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 ($PM_{ax W}$ 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 $PM_{ax W}$ to the current iteration number. $PM_{ax W}$ needs to be kept up-to-date because, in conjunction with $PM_{ax R1st}$, 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 ($PM_{ax W}$ is still zero). If that is the case, the directory for the shared array is notified. In the 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 [23].

3 Parallelizing Loops with Dependencies

The algorithms presented thus far are targeted to loops or codes that are fully-parallel most of the invocations. Now, we present new algorithms to handle loops that often have dependences. We first present how to handle the general case and then examine the special case of reductions.

3.1 General Case: The Sliding Commit Algorithm (SCA)

To parallelize loops with cross-iteration dependencies, we extend the algorithms presented with the Sliding Commit concept. The idea is that we do not need to checkpoint the array under test at the beginning of the loop. Instead, iterations will be committed on the fly, as they finish. In some cases, we will perform on-the-fly checkpointing. In any case, if a dependence violation is detected, execution is backtracked up to the dependent iterations, the state repaired, and then parallel execution is resumed again. In the following, we describe the extensions to the ANPA and APA in turn.

3.1.1 Non-Privatization Algorithm

We need to support the following operations: creating an on-the-fly undo log, committing iterations on-the-fly and, in case of a dependence violation, stopping execution, restoring the good data from the undo log, and restarting the parallel execution. We consider each of them in turn.
Creating an Undo Log on the Fly

As processors execute the loop, we need to keep a log of the updates to the array variables under test. In this way, if a dependence violation forces the squashing of some iterations, we can reconstruct the state that existed before the squashed iterations. Since the unit of work is an iteration, the log needs to record, for a given iteration, the initial value of all the array elements that will be updated in that iteration. Consequently, each log entry corresponds to a first update to an element in an iteration. The minimum necessary information needed for reconstructing the state is the Physical_Address and the Old_Value of the element being written to. Depending on the available support, we may store the element index instead of the physical address.

Because this logging occurs frequently, it needs to be supported in hardware. The cache coherence protocol identifies the first update to each array element in each iteration. For instance, in ANPA, the Write bit in the tags of a cache is set when the corresponding array element is written for the first time in thatiteration. At that time, we extend the protocol with a log message to the home directory of that element. The message prompts the directory to store a log record in the local memory of the home node.

The log message includes the current iteration number of the writing processor and the physical address of the element. The directory then uses this information together with the old value of the element stored in the memory and generates a log record. The exception is when the line is dirty in the writing processor's cache. In that case, since the memory does not have a valid copy, the message also includes the old value of the data. For simplicity reasons, we choose to always send the old value of the data. Finally, in a cache miss, when the cache does not have a copy of the data, it is the directory's responsibility to find a valid copy of the data.

The high frequency of logging and the potentially large size of the undo log requires a data structure with fast access time for logging and size management. To satisfy these requirements we have designed a log buffer which can be accessed in constant time. Its size is kept proportional to the window size (number of uncommitted iterations) and can be recycled on the fly in constant time. Figure 5 shows the organization of undo log.

![Figure 5: Organization of Undo Log.](image)

The information stored in the log is partitioned and indexable by iteration number. All data checkpointed by the same iteration, and their corresponding addresses, are stored contiguously in their chunk of memory on their respective home node. These chunks are pre-allocated before
speculative execution and their size is based on a compiler estimation (a quite reasonable task) of the maximum number of distinct write references in an iteration. In the unlikely event that the pre-allocated backup space overflows, a new chunk of memory will be allocated and linked to its predecessor, in much the same way a linked list is constructed. The first entry in all chunks is used to store a pointer to its predecessor (we point back) so that the whole backup record of an iteration can be traversed backwards (form last to first entry) making the restore mechanism fast. This 'per iteration' organization can be very efficiently recycled in a circular manner. Records of committed iterations are reassigned to a new current iteration.

The pointers to the next available entry in the undo log for an iteration are stored in a fully associative cache, the pointer cache, in the directories. The number of entries in this pointer cache is proportional to the window size, i.e., on the order of \( p \) (the number of processors). As iterations commit, their entries in the pointers cache and their log buffers are invalidated, thus making them available for reuse. In case the pointer cache becomes full, its entries can be displaced to memory, which needs more hardware support. Alternatively the scheduler can stop issuing new iterations to the processors when the number of uncommitted iterations is equal to the number of entries of the pointer cache.

The pointer cache uses Curr_Iter and a valid bit as tag and its lines contain the fields START (the starting address of current chunk), END (the end of current chunk), and NEXT (address of next available entry). By comparing END and NEXT fields we can decide whether an overflow condition exists, in which case a new chunk is allocated and linked in, also the overflow bit (Ovfl) in that entry is set. After logging the old data NEXT is incremented. To invalidate the entry of a committed iteration, hardware only need unset the valid bit and copy the value of START into NEXT if the logging of that iteration never overflow. If the Ovfl bit is set for that entry, a software handler will be called to free the memory chunks allocated for that entry. Since the chunk size is estimated according to the number of distinguished writes in one iteration, overflow is a rare case and the occurrence of that software routine is also rare.

**Committing Iterations on the Fly**

Since we use the general case of dynamic scheduling, iterations can finish out of order. Consequently, when an iteration finishes without detecting a dependence violation, it cannot be considered committed. It can only be considered committed if all the lower-numbered iterations are also finished.

To keep track of the state of each iteration, we use three data structures shared by all the processors. One of them is *iter_state*, an array which, in a simple approach, has as many elements as iterations and holds the state of the iteration. The iteration can be *unfinished* or *finished*. We do not need to initialize the *iter_state* array at the beginning of every instantiation of the loop. Instead, we simply toggle the code associated with *finished* between consecutive instantiations. The other data structures are a scalar that holds the last committed iteration (*lst_cmt_iter*), and a lock \( L \) that protects them all.

The algorithm consists in that each processor, when it finishes an iteration, tries to commit the iteration and maybe neighboring ones. This is done by updating the shared data structures. Specifically, we present a simple algorithm (*Simple SCA*) and a more advanced one (*Advanced SCA*). Simple SCA is more intuitive but suffers from lock contention. Advanced SCA is preferred.

In both algorithms, when a processor finishes an iteration, it sets the corresponding *iter_state* to *finished*. In Simple SCA, it then checks if this is the iteration that follows *lst_cmt_iter*. If so, the processor checks consecutively increasing iterations, committing them until it finds the first *unfinished* one. Committing iterations is actually done by updating *lst_cmt_iter* (see Simple SCA below, where *this* is the iteration just finished). All these checks are protected by the lock.
In Advanced SCA, instead, the processor always checks consecutively increasing iterations starting from the \texttt{lst_cmt_iter} one. This is done without lock protection. In addition, when it later tries to commit iterations, if it fails to grab the lock at the first try, it skips the commit (see Advanced SCA below). Therefore, the lock contention is low. This low lock contention more than compensates for the higher traffic caused. This algorithm is prone to races, but correctness is not affected because all processors start checking from the \texttt{lst_cmt_iter} element.

**Simple SCA:**

\begin{verbatim}
iter_state(this) = finished
lock(L)
if( lst_cmt_iter == this-1 )
    while( iter_state(this) == finished )
        this++
    lst_cmt_iter = --this
unlock(L)
\end{verbatim}

**Advanced SCA:**

\begin{verbatim}
iter_state(this) = finished
this = lst_cmt_iter + 1
while( iter_state(this) == finished )
    this++
if( --this > lst_cmt_iter )
    if( test&set(L) == 0 )
        if( this > lst_cmt_iter )
            lst_cmt_iter = this
        reset(L)
\end{verbatim}

Any iteration that commits will invalidate its corresponding entry in the associative cache and make it available for recycling. When the global sliding the commit point advances, its value (\texttt{lst_cmt_iter}) is available to all processors and the directories will invalidate all the entries associated with iterations below this point (a constant time operation). Processors may obtain the latest commit point at any time but will recycle their undo logs either before starting a new iteration or if an overflow of the backup log is imminent or has occurred. Alternatively, the buffer may be large enough that no overflow ever occurs.

**Dependence Violation, Data Restoration and Restart**

If the ANPA of Section 2.1 detects a dependence violation, a cross-processor interrupt is sent to all processors. At that point, all processors squash the iterations that they are executing. Alternatively, a more advanced approach would include, in the arguments of the cross-processor interrupt, the ID of the smaller iteration involved in the dependence violation. In that case, only the processors executing iterations with IDs higher than it would squash the iterations; the others would finish off their iterations. In any case, after the interrupt and a global synchronization step, the system is ready for data restoration.

Data restoration is performed in software by several processors concurrently. If the array under test is allocated in pages from different homes, the undo log buffer will be distributed across multiple homes. Then, each processor can restore the data from a different buffer. There is no overlap between the locations stored in different buffers.

After the global synchronization following a failure in the speculative parallel execution, all processors will invalidate the entries corresponding to the committed iterations and start restoring the needed data from the undo logs in decreasing order of their iteration number. Note that this reverse restoration order is necessary to insure that the newer updates are overwritten by the older updates. After the this repair process is completed all undo log memory space is freed.

For this process to work, two issues have to be addressed. The first one is that the pages that hold the array under test can not change their mapping between the writing and the reading of the log. This can be ensured by pinning those pages in memory. The second one is that the user process that restores the data reads a physical address from the log record. Unless special support is assumed, that process can only write virtual addresses. Consequently, the process must interrogate the operating system to know the mapping of these pages. This can be done before
the first recovery is started. The information can be set up in a table and used for this recovery and subsequent ones. While this activity certainly has overhead, it only occurs in the hopefully relatively infrequent case of dependence violation.

An alternative approach is for the undo log records to store the index of the array element they refer to, instead of the physical address of the element. This would eliminate the two previous issues. However, it makes the module in the directory that stores the undo log records significantly more complicated. That module would have to know the translation of the pages that hold the the array under test and, in hardware, translate the physical address to array index.

Finally, once the restore is complete, a global synchronization step is performed. Then, the parallel execution restarts, starting with the iteration that follows \texttt{lst\_cm\_iter}. To prevent the same dependence violation from occurring again, we give the two dependent iterations to the same processor. This can be done with a simple change to the algorithm that schedules iterations.

Note that, for the algorithm presented to work, the undo log must keep all first-updates to any array element in any iteration. Consequently, we cannot skip any updates in the baseline ANPA of Section 2.1. Skipping updates allowed the baseline algorithm to tolerate the case when two iterations that write to the same element execute out of order. Such an optimization cannot be supported now, under the sliding commit algorithm.

### 3.1.2 Privatization Algorithm

We now extend the APA presented in Section 2.2 to run in parallel privatized loops that have cross-iteration dependences. The extensions will be quite different than those for the ANPA. Specifically, we will not create an undo log with the history of stores. The reason is that, for loops whose arrays are likely to be privatizable, there is very little transfer of information from iteration to iteration. Most iterations start with a write. Consequently, there is little need to store the history of updates and then undo them in order.

The APA of Section 2.2 fails when a directory detects the execution of the second iteration of a \textit{pair} of iterations (Figure 6-(a)). Recall that, since the pair can be executed in any order, the second iteration may be the \textit{read-first} or the \textit{writing} iteration. At that point, we run a software algorithm to repair the state and allow the resumption of the parallel execution. The repair action will simply be to provide the \textit{read-first} iteration with the correct value. The actual operations performed depend on whether the second iteration is the \textit{writing} or the \textit{read-first} iteration. We consider each case in turn.

**The Writing Iteration is Executed Last**

A directory detects the \textit{writing} iteration of a pair when it receives a write with ID lower than \textit{MaxR1st}. At that point, it sends a cross-processor interrupt to all processors. One of the arguments of the cross-processor interrupt is the write ID. The processors executing iterations with higher IDs squash the iterations; the others finish off their iterations. All processors then synchronize. The system is now ready for repair.

The repair simply consists in re-executing the \textit{writing} iteration such that each write to the element also updates the shared copy of the array. This is necessary because the future \textit{read-first} iteration will read from the shared array. In addition, the processor also clears the \textit{MaxR1st} and \textit{MinW} fields of the shared array and the \textit{PMaxR1st} and \textit{PMaxW} fields of all the private arrays for that element. In effect, what we are doing is re-initializing that array element.

Ideally, the only remaining thing to do is to re-execute the corresponding \textit{read-first} iteration of all the pairs that the \textit{writing} iteration belongs to. It is, however, impossible to identify such
read-first iterations. The reason is that each processor had kept in $P_{\text{MaxR1st}}$ the maximum ID of any read-first iteration for that element that it executed. Lower IDs have been lost.

Consider, for example, Figure 6-(b), where processor $P_1$ executes a write in iteration 2 after $P_{\text{MaxR1st}}$ for $P_1$, $P_2$ and $P_3$ have already been set to 0, 6 and 5 respectively. We know that there is a pair between iterations 2-5 and 2-6. However, we do not know whether any processor read-first in 3 or 4, creating pairs 2-3 or 2-4. Consequently, our solution is to be conservative and restart parallel execution of all iterations after the writing iteration.

The Read-First Iteration is Executed Last

A directory detects the read-first iteration of a pair when it receives a read with ID higher than $MinW$. At that point, it sends a cross-processor interrupt to all processors, passing as argument the read ID. As usual, the processors executing iterations with higher IDs squash their iterations and the others finish off their iterations. All processors then synchronize. The system is now ready for repair.

Ideally, the repair consists of executing the corresponding writing iteration of the pair while updating the shared array and clearing the $MaxR1st$, $MinW$, $P_{\text{MaxR1st}}$ and $P_{\text{MaxW}}$ fields. Then, we would re-execute the read-first iteration. Unfortunately, we may not know what the correct writing iteration is. The reason is that each processor keeps only the maximum ID of any writing iteration for that element that it executed. Lower IDs have been lost.

Consider, for example, Figure 6-(c), where processor $P_1$ executes read-first iteration 6. $MinW$ is 1, and $P_{\text{MaxW}}$ for $P_1$, $P_2$ and $P_3$ is 2, 3 and 8 respectively. We do not know whether the pair is 3-6, 4-6 or 5-6 because we do not know whether $P_3$ wrote in iterations 4 or 5. We only keep the maximum writing iteration for each processor.

The restarting iteration number will be decided based on the following rule. The processor detecting the failure will compute the maximum of all $P_{\text{MaxW}}$ values across processors, named $m_{\text{MaxW}}$. If $m_{\text{MaxW}}$ is smaller than the read-first iteration then we re-execute iteration $m_{\text{MaxW}}$ followed by the read-first, after which speculative parallel execution is restarted. If, however $m_{\text{MaxW}}$ is larger than read-first then the restart point is the maximum $P_{\text{MaxW}}$ value which is smaller than the offending read-first iteration.

Figure 6: Examples of pairs (a). In reality, the system keeps only limited information that we can use to identify the pairs (charts (b) and (c)).
Other Issues

In the APA algorithm, the handler that performs the recovery needs to know the virtual to physical mapping of either the \textit{MaxRlist} or the \textit{MinW} array. This is because, when a violation occurs, the hardware only knows the physical address of the \textit{MaxRlist} and \textit{MinW} entries that cause the problem. The handler must have a way to know the index of the element. This is solved as in the ANPA, namely by interrogating the operating system and keeping a translation table. The overhead of building such a table is only suffered in the first recovery. Alternatively, we can allocate one of the arrays in a fixed location.

All the data in a loop can use either the APA or the ANPA, but not both. It is too complicated to support recovery when one array uses one algorithm and another the other. This does not constitute a limitation because the APA algorithm subsumes the parallelization capabilities of NPA.

3.2 Special Case: Reduction

A special and very frequent case of loop dependence patterns is reductions. In this section we describe how an extension to the previous algorithms can handle such dependences and execute the loops in parallel. A reduction variable is a variable whose value is used in one associative and commutative operation of the form \( x = x \otimes exp \), where \( \otimes \) is the operator and \( x \) does not occur in \( exp \) or anywhere else in the loop. A simple example is statement S1 in Figure 7-(a). The function performed by the loop is to add a value computed in each iteration to the value stored in \( A(:,i) \). This type of reduction is sometimes called an \textit{update}.

There are several known parallel methods for performing reduction operations. One method is to transform the do loop into a doall and enclose the access to the reduction variable in an unordered critical section [6, 26]. The drawbacks of this method are that it is not scalable and that it requires potentially expensive synchronizations. A scalable method can be obtained by noting that a reduction operation is an associative and commutative recurrence and can thus be parallelized using a recursive doubling algorithm [12, 13, 14]. In this case, the reduction variable is privatized in the transformed doall. A scalar is then produced using the partial results computed in each processor as operands for a reduction operation (with the same operator) across the processors (Figure 7-(c)).

The real difficulty encountered by compilers in parallelizing loops with reductions arises from recognizing and validating the reduction statements. So far, this problem has been handled at compile-time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement [26].

In the cases where data dependence analysis cannot be performed at compile time, reductions have to be validated at run-time. For example, although statement S3 in the loop in Figure 7-(b) matches a reduction statement, it is still necessary to check at run-time that the elements of array \( A \) referenced in S1 and S2 do not overlap with those accessed in statement S3. Just as before, all other potential dependences caused by the references in S1 and S2 will have to be checked.

We propose to validate these reductions at run-time by extending the capabilities of the previously presented hardware.

3.2.1 Hardware Support for Run-Time Reduction Verification

We speculatively transform the reduction for parallel execution and then, at run time, check if the memory elements referenced in the reduction statement are accessed anywhere else in the loop.
outside this statement, by setting an associated tag.

In a simple case, when only reductions need to be verified, we use two mark bits: RED (reduction bit) which is set if the element is accessed by the reduction statement, and NRED (non-reduction bit) which is set if the element is accessed outside the reduction statement. These bits are cleared at the beginning of the loop. They are used as follows:

```
if ( RED == 1 )
  FAIL
else
  NRED = 1
access data
```

(a): Reference outside
the reduction statement.

References to the data structure under test are replaced with new instructions which encode this algorithm. Any reference outside the reduction statement first checks the RED bit. If it is set, that element has been used in the reduction statement and execution fails. Otherwise, NRED is set. References from the reduction statement will test NRED bit and set RED bit. If all references pass the algorithm test, the reduction parallelization was legal.

In most applications, reduction verification has to be combined with the ANPA or APA presented in the previous sections. We will first present the transactions of ANPA with reduction verification (ANPAR), then APA with reduction verification (APAR).

To enhance ANPA with reduction verification, we add the reduction bit RED to both cache tag and directory entry of the shared data under test. There is no need to have a NRED bit: to detect if the array element has been accessed outside the reduction statement we will simply check if $MazR|MazW$ are not zero for the element. In Figure 8 we show a few examples of the transactions of the new algorithm. We use tag and dir to represent mark bits in tag and directory. Note that the enhancements to the previously presented ANPA algorithm are shown in bold face.

Consequently, for accesses outside the reduction statement, we simply test the RED bit and, if zero, set MazR or MazW as in Figures 2 and 3. For accesses inside the reduction statement, we test MazR or MazW and, if both are zero, set the RED bit. This only needs to be done by only one of the two accesses in the reduction statement, either the read or the write access.

Note that the MazR, MazW and RED bits accessed in the reduction statement refer to the corresponding entry in the shared structure under test, not the private copy of it (Figure 7-(c)). Consequently, the protocol transactions in a reduction statement access involve a message to the directory of the private entry which, in turn, will send a test message to the directory of the shared entry.
if (Cache_hit )
  if (tag.RED == 1 )
      /* has been accessed by red stmt */
      FAIL
  else
      read data from cache
      if (Read == 0 )
          /* first read in current iteration */
          send Read_Test_Req(Curr_Iter) to home
      else
          FAIL
      if (dir.state is DIRTY )
          send Net_Read_Req to owner
          if( Curr_Iter < MaxW )
              FAIL
          else
              if( Curr_Iter > MaxR )
                  MaxR = Curr_Iter
          (c): Home Dir receives Read_Test_Req(Curr_Iter).
          if( Cache_hit )
              read data from cache
              if (tag.PRED == 0 )
                  send RED_Test_Req to home
                  set dir.PRED
              else
                  send RED_Read_Req to local mem
                  tag.PRED = 1
      (d): Processor read from reduction stmt.
  (b): Home Dir receives Read_Req(Curr_Iter).

if (tag.RED == 1 )
  FAIL
else
  if (Curr_Iter < MaxW )
      FAIL
  else
      if (Curr_Iter > MaxR )
          MaxR = Curr_Iter
  (e): Processor read from reduction stmt.
  if (tag.PRED == 0 )
      send RED_Test_Req to home
      dir.PRED = 1
  else
      send data to cache
      if (MaxR | MaxW != 0 )
          FAIL
      else
          dir.RED = 1
      (f): Home receives RED_Test_Req.

To reduce the number of messages that need to be sent to the directory of the shared version of the array, we add an extra bit (private RED or PRED) to both cache tag and directory entry of the private copy of the array. This bit will be tested at the beginning of every reduction access. In the first reduction access on an element, we will set both RED and PRED for the element. In all subsequent reduction accesses to the same element, the test to PRED will find it set and, therefore make it unnecessary to access MaxR, MaxW, and RED.

Transaction examples of APA with reduction verification (APAR) are shown in Figure 9 and the new transactions for reduction verification are shown in bold face. Same as ANPAR, a RED bit is added in the directory of shared copy and MaxR1st|MinW serves as NRED bit. PRED bit is added in both tag and directory of the private array. After privatization of the array under test, all references in the loop body (outside or from the reduction statement) will access the private array and readin is performed if necessary. In APA, any read first or first write in each iteration to a particular data element will signal the directory of the shared array. To verify reduction operation, the directory for the shared array will check the reduction bit RED in APAR. All references from the reduction statement will first detect if that element has been accessed outside of the reduction statement by checking Read1st|Write in tag or PMaxR1st|PMaxW in the directory of private array. If that is the first reduction operation done by the processor, a RED_Test_Req will be sent to the directory of the shared array and the check on MaxR1st|MinW will be performed there.

3.2.2 Failure Recovery

Recovery in the ANPAR requires the allocation of an undo log for each private reduction array. During the speculative loop execution, every update to the reduction elements will be preceded by the generation of an undo log record as in the ANPA case. If failure recovery is needed, the contribution of the squashed iterations to the reduction variables is undone using the log records,
if( Cache_hit )
if( tag.PRED == 1 )
   FAIL
else
   read from cache
if( Read1st|Write == 0 )
   Read1st = 1
   send read-first signal to private directory
else
   send read request to private directory
(a): Processor read outside of reduction statement.

PMaxR1st = Curr_Iter
send read-first signal to shared directory
(b): Directory for the private array receives a read-first signal.

if( dir.PRED == 1 )
   FAIL
else
if( PMaxR1st=PMaxW==0 for all elem. in the mem line )
   send read-in-req to shared directory
   copy the reply into private data
   PMaxR1st = Curr_Iter for the elem. requested
   send line to cache with tag.Read1st=1 for the elem. requested
else
   send line to cache
(c): Directory for the private array receives a read request.

if( dir.RED = 1 )
   FAIL
else
if( Curr_Iter > MinW )
   FAIL
   MaxR1st = max(MaxR1st, Curr_Iter)
(d): Directory for the shared array receives a read-first signal.

if( Cache_hit )
if( Read1st|Write == 1 )
   FAIL
else
   read data from cache
if( tag.PRED == 0 )
   send RED_Test_Req to private directory
else
   send RED_Read_Req to private directory
tag.PRED = 1
(e): Processor read from reduction statement.

if( PMaxR1st|PMaxW == 1 )
   FAIL
else
   dir.PRED = 1
   send RED_Test_Req to shared directory
(f): Directory for the private array receives a RED_Test_Req.

if( PMaxR1st|PMaxW == 1 )
   FAIL
else
if( dir.PRED == 0 )
   send RED_Test_Req to shared directory
dir.PRED = 1
   send data to cache
(g): Directory for the private array receives a RED_Read_Req.

if( MaxR1st|MinW == 1 )
   FAIL
else
   dir.RED = 1
(h): Directory for the shared array receives a RED_Test_Req.

Figure 9: APA with Reduction Verification Transactions.
in a similar way as ANPA recovers from shared data dependence violations. If, at some point during execution, a reduction element is read outside the reduction statement (i.e., as shared data) the recovery procedure first undoes the contributions of all squashed iterations from the partial results of the reduction and then merges the per processor partial results into the shared memory. After that, the loop execution can continue with a read from the shared memory. However, if a reduction element is accessed by a shared write, in effect killing the accumulated value, the recovery procedure, after undoing the effect of all squashed iterations, updates the shared memory with this value and re-initializes the corresponding private reduction elements (data and tags) on all processors. After a failure recovery, the type of an element (reduction or shared), is decided by the type of the next reference to it.

3.3 Hardware Support for Reduction Optimization

In the most common reduction algorithms, the final cross-processor merge of the per-processor partial reduction results often introduces a very significant overhead. Consequently, we propose a new hardware scheme to reduce its impact. It is important to note that this scheme can benefit parallelized reductions in general, irrespective of whether they have been verified at compile or at run time.

The merging phase of the reduction parallelization is proportional, even in its best implementation, to the number of distinct memory references performed during loop execution by each processor. For example, in the case of a dense access pattern, the $p$ private arrays of length $s$ containing the per-processor accumulation will be merged in time $s * p/p = s$, i.e., without speedup. The merging of sparse private data can be done either in time $s$, i.e., proportional to the length of the corresponding dense structure (which can be orders of magnitude larger than the number of accesses) or in time $s * \log p$ if the data is maintained in hash tables. In both cases no speedups can be obtained and the execution time of the entire merging step is added to the critical path of the program.

Most of the cost associated with the merging phase is due to its almost exclusive remote accesses. Additionally, the traversal of the data structures associated with the reduction causes the displacement of all the data in the processor caches, thus leaving them in a flushed state for the subsequent step of the program. The initialization of the private data structures has a similar cache flushing effect on the loop itself, because it is performed before its start.

The other commonly used parallelization technique, namely updates of the shared data in unordered critical sections, involves heavy use of synchronizations. This makes it non-scalable and thus impractical. We would like to combine the advantages of the two previously-mentioned software schemes.

3.3.1 Private Cache-Line Reduction Scheme (PCLR)

This scheme optimizes reductions for both speculative and normal execution modes. The idea is to perform the per-processor reduction operation on a privately-allocated cache line that is kept non-coherent. In addition, the shared data is updated through a home directory operation only when the cache line is displaced. At that point, the directory merges (e.g., adds) the partial result from the private cache line into its shared counterpart. Consistency is guaranteed by flushing all dirty lines associated with the reduction at the end of the loop. This scheme allows private, contention-free accumulation.

The implementation of this scheme requires the following hardware enhancements:

**On-demand private cache line initialization.** A reference to the reduction array that causes a cache miss is serviced by the directory controller by providing a new cache line initialized with
the neutral element with respect to the reduction operation (e.g., 0 for addition and 1 for multiplication). In order to distinguish between regular cache lines and private reduction cache lines, a RED bit per line is provided. In the case of speculative reduction parallelization, we have seen that we need 1 bit per word. As we have seen, these bits are set by the first reference to the reduction element. All other references are handled with the plain cache coherence protocol.

**Modification of the displacement algorithm.** When a private reduction line (identified by the tag bit) is displaced from cache, the directory forwards it to the home of the shared counter-part as a *merge request*. The home node buffers the request and then performs the merge.

**Buffering and Functional Units in Directory Controller.** To merge the cache line in a merge request with its shared copy in regular memory, the directory needs a buffer to hold the requests. Additional simple functional units like an adder, comparator or logic operator are provided for the reduction operation. The buffer holds the memory block and its address in one entry. The memory copy of the block is fetched, operated upon by the functional units element by element, and then written back to memory. When regular and reduction elements are interleaved in the same cache line as in Spice, the per-word RED bit is used as a mask for the directory operations. These merging operations are given lower priority than the regular coherence transactions, thus not slowing the regular memory traffic. Finally, they all have to be completed before the global synchronization at the end of the parallel loop.

**New Protocol Transactions.** Before merging the private reduction cache line with its corresponding shared memory block, the directory state of the latter is checked. In case it is shared or dirty in another processor, proper invalidations and write back transactions are performed. The directory initiates the *merging* only when the memory has the only valid copy of the block.

**Final Merging Phase.** After the execution of the loop, all processors flush to memory all valid reduction lines from their cache. All processors work concurrently.

Overall, this scheme has several extra advantages:

- Since the shared memory updates are performed by the directory, the merging phase can overlap with the parallel loop execution.
- After the loop ends, the final merging operation of the still not displaced cache lines flushes only the subsequently useless data. Recall that the private partial results represent dead values. This merging phase of the residual lines is upper-bounded by the cache size and not by the, usually much larger, data size.
- Initialization of the private data structures is on demand, which, in the case of sparse access patterns, may represent good improvement. Because it is performed during the loop execution itself, the associated memory latency may be overlapped with other computation.

We note that if the reduction array size is smaller than the cache size this scheme is not beneficial. However, the compiler can estimate this size and chose whether to use this hardware optimization or not.

### 4 Experimental Results

We have evaluated the proposed optimizations through simulations. In this section, we present our simulation environment, the workloads, and conclude with the obtained experimental performance measurements.
4.1 Simulation Environment

Our evaluation is based on execution-driven simulations of a CC-NUMA shared-memory multi-
processor using Tangolite [8]. The modeled multiprocessor has the hardware support of the basic
design [24] and the enhancement proposed in this paper. The simulated applications have been
pre-processed with the Polaris [3] parallelizing compiler that has been specifically enhanced to
transform selected loops for speculative run-time parallelization. The compiler has inserted all
necessary instructions to perform the marking and analysis phases.

The modeled architecture has 200-MHz RISC processors, each with 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 cache sizes have been purposely selected so small in order to scale with the reduced
working sets of the chosen applications. Real-life working sets could not be used because they would
have required impractically long simulation times. The caches are kept coherent with a DASH-like
cache coherence protocol [15]. Each node has part of the global memory and the corresponding
section of the directory. We have modeled the contention in the whole system with the exception
of 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 and will increase with
resource contention. Due to lack of time, the presented experimental data has been obtained using
a single-issue processor. In the final version of the paper, all the results will reflect the use of a
4-issue dynamic superscalar processor.

Processes synchronize using locks and barriers. The pages of the shared data have been allocated
round-robin across the different memory modules. We choose this allocation because these loops
have irregular access patterns and iterations are scheduled dynamically. Private arrays are allocated
locally.

4.2 Workloads

Due to the impractically long running simulation times of full-length applications we have extracted
and measured only the performance of representative (in terms of relative execution time) loops
from well-known codes. Table 1 lists the set of the simulated loops, their weight relative to the
total sequential execution time of their respective application (%Tseq, as measured on 1 processor
of a SGI PowerChallenge), the number of times they are instantiated during program execution,
their average number of iterations, the total size of the arrays under test and the algorithm used
to parallelize/optimize them. Adm, Track, and Spice are Perfect Club [2] codes, Euler and Dsmc3d
are HPF-2 applications [5], and Rmv is a Spark98 kernel [17].

<table>
<thead>
<tr>
<th>Appl</th>
<th>Loop Name</th>
<th>% of Tseq</th>
<th>Instantiation Number</th>
<th>Iteration Number</th>
<th>Array Size (bytes)</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adm</td>
<td>run_1o[20,30,40,50,60,100]</td>
<td>20.6</td>
<td>900</td>
<td>3204</td>
<td>803072</td>
<td>ANPA,APA,SCA</td>
</tr>
<tr>
<td>Track</td>
<td>run_1o300</td>
<td>40.9</td>
<td>56</td>
<td>480</td>
<td>72400</td>
<td>ANPA,SCA</td>
</tr>
<tr>
<td>Spice</td>
<td>load_1o_2 BI_T_1o_2</td>
<td>11.6</td>
<td>1106</td>
<td>264</td>
<td>8000000</td>
<td>ANPAR</td>
</tr>
<tr>
<td>Euler</td>
<td>dpHum_1o[100,200], psmo_1o20</td>
<td>89.9</td>
<td>120</td>
<td>58663</td>
<td>763080</td>
<td>ANPA,SCA,PCLR</td>
</tr>
<tr>
<td>Dsmc3d</td>
<td>move1_1o100</td>
<td>32.8</td>
<td>80</td>
<td>383107</td>
<td>158400</td>
<td>ANPA,SCA</td>
</tr>
<tr>
<td>Rmv</td>
<td>local_mvp_for</td>
<td>29.0</td>
<td>1</td>
<td>92160</td>
<td>737280</td>
<td>PCLR</td>
</tr>
</tbody>
</table>

Table 1: Application Characteristics.
The *Adm* loops have 32 or 64 iterations and a small working set. To three of the arrays we have applied ANPA and to other four arrays APA, resulting in a fully parallel loop. In *Track* ANPA is applied to 4 arrays. Interestingly, 5 of the 56 loop executions are not fully parallel. After the first failure recovery all remaining iterations can be executed in parallel without failure. The loops in subroutines *Load* and *BJT* from *Spice* have a highly irregular access pattern (they follow a linked list) but, after applying ANPA and reduction parallelization, they become fully parallel.

The *dflux* loop in *Euler* performs only some simple computation and a compile time verifiable reduction. Because the access pattern of the tested array is sparse, the case can be handled as a regular reduction array or as a shared array. When handled as a regular reduction array, reduction optimization (PCLR) has been applied. If it is handled as a shared array, the loop becomes partially parallel and will cause a dependence related failure just before it finishes. After recovery, the remaining iterations are executed sequentially.

For the loop *move3 goto100* in *Dsmc3d* privatization removes all dependences [1] but, due to its sparse nature, causes high initialization and final merging overhead. By treating the tested array as a shared array and applying SCA we obtain good overall results. *Spark98* is a sparse matrix and dense vector multiplication C kernel. *Rmv* needs only reduction optimization.

### 4.3 Evaluation

We have used *Adm*, *Track*, *Euler*, and *Dsmc3d* to evaluate the sliding commit algorithm (SCA), *Spice* for reduction verification and *Spice*, *Euler*, and *Rmv* for evaluating the hardware for reduction optimization (PCLR). For most loops we have simulated 16 processors, and 8 if their iteration number was small.

#### 4.3.1 Sliding Commit

To compare the sliding commit scheme with the basic design, we have chosen the ANPA and sliding commit (SCA) combination. We present parallel execution time normalized to the serial execution time. Figure 10 shows the comparative performance of applying ANPA and SCA on four applications running on 16 processors. From these only *Adm* is fully parallel, while all others fail during the speculative parallel execution. On average, ANPA provides a speedup of 3.81, while SCA provides a speedup of 8.71.

![Figure 10: Sliding commit (whole loop).](image)

For *Adm* which is fully parallel, the difference in performance between ANPA and SCA is due
to the optimization of backup. While in the ANPA design, the backup is performed in software into storage of the same size array as the array under test, the SCA uses a much smaller undo log implemented in hardware, and thus much more efficient. The SCA design increases the synchronization time, but the overall performance is still improved.

The loop in *Track*, fails in 5 of its 56 instantiations. In the ANPA scheme the loop is continued serially after the first detected dependence and causes a slowdown, but over the life of the program, a speedup of 5.26 on 16 processors is still obtained. Due its fast failure recovery scheme of the SCA the speedup increases to 8.33. Figure 11 shows how the execution of the first failed instantiation of *Track* under ANPA causes a slowdown but still obtains a speedup of 2 under SCA.

In *Euler* the loop body is relatively simple, so the memory time dominates the total time. A failure will occur very late in the execution of the loop. While ANPA discards correctly executed work, the SCA method provides a speedup of 9.1.

*Dsme3d* has dependences in half of its instantiations. To keep the timestamps within their 2 byte maximum length the loop is scheduled dynamically in chunks of 16. This method reduces synchronization time and improves data locality. With ANPA the execution is slowed down only due to the failure recovery phase. With SCA we obtain a speedup of 8.33.

### 4.3.2 Reduction Verification

The *load* loop in *Spice* can be parallelized by applying ANPA and reduction verification. Because the total iteration number of the loop is small, we only simulate 8 processors. Figure 12 shows the breakdown of the execution time into the private array initialization and actual loop computation phase (*Loop*), and the merge phase (*Merge*) during which the partial results are added into the shared array. On average, the speedup is 2.65 on 8 processors. The experiments also show that the merging phase introduces a large amount of memory, synchronization, and instruction overhead which hints to the need for reduction optimization.

![Figure 11: Sliding commit (one instantiation).](image1)

![Figure 12: Reduction Verification (8 proc).](image2)

### 4.3.3 Reduction Optimization

In this section we evaluate the performance of the PCLR reduction optimization hardware for *Spice*, *Euler*, and *Rmv*. The reductions in *Euler* and *Rmv* are regular and their loops are fully parallel. *Spice* needs both ANPA and reduction verification. Figures 13 and 14 show the performance gains of the PCLR scheme over the standard software approach on 8 processors and, respectively, 16 processors. In the graphs, *SW* and *HW* are, respectively, the normalized execution times of the software and hardware optimized merging phases. The bars of *SW* and *HW* are broken down into the *Loop* (initialization and actual loop workload), and *Merge* phases. Note that for the software scheme, *Merge* represents the time to merge the private arrays, while for the hardware scheme it represents the time needed to flush the primary and secondary caches.
The size of the $SW$ points to the relative importance of optimizing the *merge* of the private arrays, i.e., the need for optimizing the most commonly used method in parallelizing reductions. The size of the $HW$ bars proves our concept: for *Euler* and *Spark*, the overhead of flushing caches is less than 1%, an excellent result. The difference between $SW$ and $HW$ for *Load* is very small because the size of reduction elements is small, comparable to the cache size. As mentioned in Section 3.3.1 the compiler can evaluate the size of the private structures needed for the partial results and chose the appropriate technique: with or without PCLR.

5 Related Work

Some work related to ours is four schemes that support speculative parallelization inside a multiprocessor chip [9, 11, 18, 21]. These schemes are relatively 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. 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. Recovery from a wrong speculation is relatively simple: the cache lines with speculative data are invalidated and the successor tasks are squashed and restarted. Overall, these schemes are targeted to small-scale parallelism. The scheme proposed in this paper, instead, is targeted to the larger-scale, coarser-grain parallelism exploited in DSM multiprocessors.
6 Conclusions

Speculative parallel execution of statically non-analyzable codes on Distributed Shared-Memory (DSM) multiprocessors is challenging because of the long latency and distribution present. However, such an approach may well be the best way of speeding up codes whose dependences cannot be compiler analyzed. In this paper, we have extended past work by proposing a hardware scheme for the speculative parallel execution of loops that have a modest number of cross-iteration dependences. In this case, when a dependence violation is detected, we locally repair the state. Then, we restart parallel execution from that point on. The general algorithm is called the Sliding Commit Algorithm. If the loop dependences are of the special form of reduction, we propose a specialized algorithm. Simulations indicate good speedups relative to sequential execution. The hardware support for reduction optimizations brings, on average, 50% performance improvement.

References


