Hardware for Speculative Parallelization in High-End Multiprocessors[1]

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

†University of Illinois at Urbana-Champaign, http://iacoma.cs.uiuc.edu
‡Texas A&M University, http://www.cs.tamu.edu/faculty/rwerger/

Abstract

Speculative parallel execution may well be the best way of speeding up codes whose dependences can not be analyzed by the compiler. Supporting speculative parallelization in hardware in Distributed Shared-Memory (DSM) multiprocessors is challenging because of the long-latency of memory accesses and the physical distribution of the memory system. Our approach is to add extensions to the cache coherence protocol hardware of the machine to automatically detect dependence violations at run time.

In this paper, we briefly summarize some of the results of our work. We present two schemes that we apply to loops. In a first scheme, if a dependence violation is detected at run time, execution is interrupted, the state is rolled back in software to the beginning of the loop, and the loop is re-executed sequentially. In the second, more aggressive scheme, if a dependence violation is detected at run time, execution is interrupted, the state is repaired in software on the fly, and parallel execution is restarted from that point on. The first scheme is targeted to loops where most of the executions turn out to run fully in parallel, while the second scheme is targeted to loops that have a modest number of cross-iteration dependences.

1 Introduction

Compiler-driven parallelization of codes has advanced significantly in this decade [2, 4, 7]. Unfortunately, there is still a large body of potential fully- or partially-parallel codes that compilers cannot parallelize because they can not fully analyze the codes' dependence structure. The dependence structure may be too complicated to analyze for current compiler technology or simply not available at compile time. In any case, 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 is executed 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 have loops that have huge iteration counts and are fully parallel or have only a few cross-iteration dependences. 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))          A: Array under test
enddo
```

Figure 1: Loop that cannot be compiler analyzed.

To run these codes in a fully- or partially-parallel manner, software approaches based on an *inspector-executor* pair have been proposed ([3, 11, 12, 16] for example). An inspector loop analyzes the data access patterns at run time and yields a partitioning of the iteration space into subsets called wavefronts. Each wavefront is then executed in parallel by the *executor*, with synchronization separating the wavefronts. In general, however, the inspector may be computationally expensive and have side-effects.

We propose a new framework for speculative parallelization in hardware [17, 18, 19]. The scheme is based on a software-based run-time parallelization scheme that we proposed earlier [13]. The idea is to execute the code (loops) speculatively in parallel. As parallel execution proceeds, extra hardware added to the directory-based cache coherence of the DSM machine detects if there is a dependence violation. If such a violation occurs, execution is interrupted, the state is rolled back in software to the most recent safe state, and the code is re-executed from that point.

Such a scheme is somewhat related to speculative parallelization inside a multiprocessor chip [6, 8, 9, 14], which also relies on extending the cache coherence protocol to detect dependence violations. Our scheme, however, is targeted to large-scale DSM parallelism. In addition, it does not have some of the limitations of the proposed chip-multiprocessor schemes. Such limitations include the need to bound the size of the speculative state to fit in a buffer or L1 cache, and a strict in-order task commit policy that may result in load-imbalance among processors [6, 8, 9, 14]. On the other hand, our scheme is likely to have a higher hardware complexity, largely due to the caching and distribution of the memory system in a DSM machine.

In this paper, we briefly summarize some of our schemes. We present two schemes that we apply to loops. In the first, base scheme, if a dependence violation is detected at run time, execution is interrupted, the state is rolled back in software to the beginning of the loop, and the loop is re-executed sequentially. In the second, extended scheme, if a dependence violation is detected at run time, execution is interrupted, the state is repaired in software on the fly, and parallel execution is restarted from that point on. The first scheme is targeted to loops where most of the executions turn out to run fully in parallel, while the second scheme is targeted to loops that have a modest number of cross-iteration dependences.

[1] This work was supported in part by NSF under grants Young Investigator Award MIP-9457436, ASC-9612099, MIP-9619351, CAREER Award CCR-9734471 and ACI-9872126, DARPA Contract DABT63-95-C-0097, DOE ASCI Contract B347886, NASA Contract NAG-1-613, NCSA grant ASC980006N and gifts from Intel and IBM.

2 Base Scheme for Speculative Parallelization

In this scheme, the safe state to which we will roll back in software in case of a dependence violation, is established at the beginning of the loop [17, 18, 19]. This scheme can be fleshed out into different hardware algorithms with different cost and performance. We envision the DSM machine to support a few such algorithms mapped to the same hardware, and the compiler to select the algorithm on an array-by-array basis.

First, there are the privatization and the non-privatization algorithms. The latter should be used when the compiler can prove that the variables updated in the loop are not privatizable. This is because privatization, while requiring extra storage, uncovers more parallelism because it eliminates output- and anti-dependences. Among the privatization algorithms, we focus here on the most general one, namely the Advanced Privatization Algorithm (APA). Among the non-privatization ones, we focus here on the equivalent to the APA, namely the Advanced Non-Privatization Algorithm (ANPA). In the rest of this section, we summarize the ANPA and the APA. We assume the general case of a dynamically-scheduled loop. The non-analyzable variables in the loop that may cause a dependence violation we call *arrays under test*.

2.1 Advanced Non-Privatization Algorithm (ANPA)

In this algorithm, the arrays under test are backed up in software before the loop starts. In case of a dependence violation during loop execution, the recovery to a safe state will consist of restoring the arrays to their original values in software.

The intuition behind the ANPA is to keep two shared pieces of information for each element of the array (or arrays) under test: the highest iteration that has read the element so far (*MaxR*) and the highest iteration that has written the element so far (*MaxW*). When a processor executing iteration *Curr* reads or writes the element, the hardware performs the algorithms of Figure 2-(a) or 2-(b) respectively. These algorithms check if the element has been accessed by out-of-order iterations, causing a write-after read (WAR), read-after-write (RAW) or write-after-write (WAW) violation. If so, a FAIL cross-processor interrupt is broadcasted.

```
if (Curr < MaxW)
   FAIL /* WAR */
else
   read data
   MaxR = max(Curr, MaxR)
   /* out-of-order rds OK */
```

(a): Processor read.

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

(b): Processor write.

Figure 2: Compact form of the ANPA.

In reality, out-of-order execution of iterations that write to the same element is fine for the ANPA as long as we skip the second update to the data and to *MaxW*. However, in the extended algorithms of Section 3, where we repair dependence violations on the fly, all writes must be performed and recorded in order. For some of those algorithms, which are based on the ANPA, we will use the complete code of Figure 2.

The actual implementation of the ANPA is as follows. The *MaxR* and *MaxW* arrays are kept in the directory controller of the home node (or nodes) of the corresponding array under test. If the array under test is distributed, so are the *MaxR* and *MaxW* arrays. *MaxR* and *MaxW* are operated upon by the directory controller only. When a processor accesses an element of the array under test, the directory controller is prompted to operate on the corresponding *MaxR* or *MaxW* entry.

Since elements of the array under test can be cached, the scheme described potentially involves much extra traffic between processor caches and directories. To reduce this traffic, we add some state in the tags of each cache in the machine. Note that the directory only needs to be informed of the first read and the first write to each element in the iteration. Consequently, we add a *Read* and a *Write* bit per cache line word. These bits are set by the hardware the first time in the iteration that the corresponding element is read or written respectively. Cache read hits to words with the *Read* bit set do not generate directory accesses. The same occurs for writes and the *Write* bit. The *Read* and *Write* bits of all the lines in the cache are cleared in hardware at the beginning of every iteration. The bits for a cache line are lost if the line is displaced from the cache. These algorithms plus other useful variations are described in detail in [17, 18].

2.2 Advanced Privatization Algorithm (APA)

In this algorithm, each processor makes a private copy of the array under test on demand and operates on it. In case of dependence violation during loop execution, the recovery to a safe state simply consists of discarding the private arrays.

To understand the APA, 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 loop with a privatized array under test is fully parallel if each of the elements of the array satisfies one of the following conditions: (i) it is read-only, (ii) every read to it is preceded by a write to it in the same iteration or (iii) no *read-first* iteration for the element is preceded by an iteration that writes the element. An iteration that writes the element is called a *writing* iteration. Figure 3 shows examples of fully-parallel loops.

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

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

It 1	It 2
Rd	Wr
	Rd

It 1	It 2
Rd	Rd
	Wr
	Rd

It 1	It 2
Rd	Wr
Wr	Rd

Figure 3: Examples of loops that are fully parallel if the element that is read and written is privatized.

The combination of a *writing* iteration and a *read-first* iteration that, in the sequential loop execution, consumes the data generated by the *writing* iteration, we call a *pair* (Figure 5-(a)). From the above definitions, the existence of a *pair* is a necessary condition for a loop not to be fully parallel. Consequently, the APA simply tries to detect the second iteration of a *pair*, and trigger a failure when it finds it. Note that the two iterations of a *pair* may actually be executed in-order or out-of-order.

Our algorithm implementation involves adding state to the directories of both the shared copy of the array and its privatized copies. The directory of the shared array maintains two timestamps 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 other keeps the number of the lowest *writing* iteration for the element executed so far by any processor (*MinW*). The parallelization fails when *MaxR1st* is larger than *MinW*.

During execution, processors operate on private data. Consequently, to identify *read-first* iterations, the directories of the private copies of the array keep, for each element, two timestamps: 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 (*PMaxW*).

Finally, to reduce messaging, the tags of the caches keep a summary of the private directory state. For each word, they keep 2 bits to indicate whether the current iteration is *read-first* for the element (*Read1st*) and/or *writing* (*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* are zero, *Read1st* gets set.

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

When a processor writes to an element of the array under test, it checks whether this is the first write to the element in this iteration. For the check, it can use the state of the cache tags (*Write* is zero) or, if the line is not in the cache, the state of the directory for the private array (*PMaxW* is less than the current iteration number). If this is the first write, *PMaxW* is updated to the current iteration number. *PMaxW* needs to be kept up-to-date because, in conjunction with *PMaxR1st*, identifies *read-first* iterations. In addition, if this is the very first write of this processor to this element (*PMaxW* was still zero), 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 updated. More details can be found in [17, 18].

3 Extended Scheme for Speculative Parallelization

The algorithms presented thus far target loops that are fully parallel in most of the invocations. Now, we extend the algorithms to handle loops that often have cross-iteration dependences. With the new extensions, when a dependence violation is detected, the state of the loop is quickly repaired on the fly and *parallel execution* is resumed from that point on. In the following, we first extend the ANPA and then the APA.

3.1 Extending the ANPA: Sliding Commit

The new ANPA does not need to back up the array under test at the beginning of the loop. Instead, it saves a trace of writes on the fly into an undo log as the execution of the loop proceeds. If a dependence violation is detected, the undo log is used to repair the state. The space of the undo log is continuously recycled as iterations commit. At any point in time, only a window of iterations use the undo log space and, therefore, are uncommitted. This window slides as execution proceeds. This concept we call the *Sliding Commit Algorithm* (SCA). In the following, we describe how the undo log is created, the iterations committed, and the dependence violations handled.

3.1.1 Creating an Undo Log on the Fly

Since the unit of work is one iteration, the log needs to record, for each iteration, the initial value of all the array-under-test elements that are updated in the iteration. Consequently, we generate a log entry at every first update to an element in an iteration. The information that we need to store is the iteration number, the physical address of the array-under-test element and the value of the element before it was written.

Because logging occurs frequently, log record generation and maintenance must be supported in hardware. In our implementation, the cache coherence protocol identifies the first update to each array-under-test element in each iteration. This is done using the *Write* bit in the cache tags described in Section 2.1, which is set when its corresponding element is written for the first time in the iteration. At that time, we extend the protocol with a log message to the home directory of that element. The message contains the iteration number, physical address of the element and, if the cache had the element, the old value of the element. In a cache miss, the directory obtains the old value of the data from other caches or memory.

All this information is handled in hardware by the directory controller using the data structures of Figure 4: the Pointer Cache, the Free Sector Stack and the Undo Buffer. If the array under test is distributed across several nodes, each node will have these data structures. The Undo Buffer is allocated in memory and divided into logical chunks called Sectors. For every iteration that requests space in this node, we assign an entry in the Pointer Cache to keep the iteration number, and a sector in the Undo Buffer to store all the data and address log records.

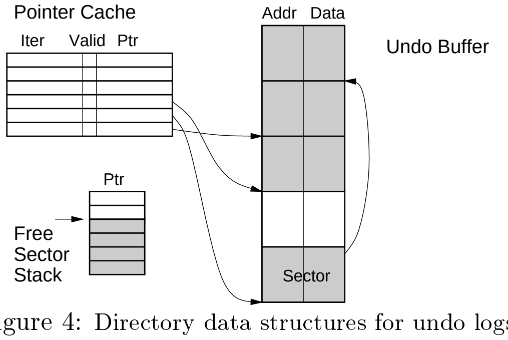

Figure 4: Directory data structures for undo logs.

The Pointer Cache is fully associative. The number of entries is a small multiple of the number of processors. This is the maximum number of iterations that can be uncommitted at any point in time. If the number of uncommitted iterations reaches the maximum value, the scheduler that provides iterations to processors stops supplying iterations until some iterations commit.

The fixed size of the Undo Buffer sectors is set before execution. It can be easily set based on a compiler estimation of the maximum number of distinct write references per iteration. However, an iteration may write to elements from different home nodes and, therefore, allocate space in several nodes. Consequently, to save space, we aggressively reduce the size of the sectors to a fraction of our initial estimate. This may result in sector overflow, which we handle by linking sectors as shown in Figure 4. The Free Sector Stack is used to identify free sectors. Overall, sector allocation, log storage, and sector overflow handling are all done in hardware for performance reasons. Undo Buffer overflow, however, is handled in software by the node's processor to reduce hardware costs [17].

3.1.2 Committing Iterations on the Fly

When an iteration finishes, it cannot be committed until all the lower-numbered iterations are also finished and committed. To keep track of the state of each iteration, processors share three data structures. One of them is *ItSt* (for iteration state), an array which, in a simple approach, has as many elements as iterations, and holds the state of each iteration. An iteration can be *unfinished* or *finished* (FIN). To avoid initializing *ItSt*, we toggle the code associated with *finished* between consecutive loops. The other data structures are a scalar with the last committed iteration (*LstCmtIt*), and a lock L that protects them all.

When a processor finishes an iteration, it tries to commit it by updating *ItSt* and *LstCmtIt*. While it could use a simple and intuitive algorithm to do it, we propose a more advanced, efficient one. In both algorithms, when a processor finishes an iteration, it sets the corresponding *ItSt* element to *finished*. In the simple algorithm, it then checks if this is the iteration that follows *LstCmtIt*. If so, the processor checks consecutively increasing iterations, committing them until it finds the first *unfinished* one. Committing iterations is done by updating *LstCmtIt*. All accesses are protected by the lock.

Simple Algorithm:
```
ItSt(this) = FIN
lock(L)
if(LstCmtIt == this-1)
  while(ItSt(this) == FIN)
    this++
  LstCmtIt = - -this
unlock(L)
```

Advanced Algorithm:
```
ItSt(this) = FIN
this = LstCmtIt + 1
while(ItSt(this) == FIN)
  this++
if( - -this > LstCmtIt )
  if(test&set(L) == 0)
    if(this > LstCmtIt)
      LstCmtIt = this
    reset(L)
```

In the advanced algorithm, instead, the processor always checks consecutively increasing iterations starting from the *LstCmtIt* one. This is done without lock protection. In addition, when it later tries to commit iterations, if it fails to grab the lock in the first try, it skips the commit (see the advanced algorithm). Therefore, the lock contention is low. This low lock contention more than compensates for the higher traffic caused and the fact that the commit may be delayed a bit.

After a processor advances *LstCmtIt*, it passes the new value to all directory controllers. The latter, in hardware, deallocate the Pointer Cache entries and recycle the Undo Buffer sectors of the committed iterations. The space can now be reused. This operation involves very little work. In our design, we optimize this operation to reduce the number of messages sent to directory controllers and to handle the possible deallocation of overflowed Undo Buffers in software. The details are presented in [17].

3.1.3 Dependence Violation and Restart

If a directory controller detects a dependence violation, it sends a cross-processor interrupt to all processors, passing, as argument, the ID of the smaller iteration involved in the violation. Then, the processors executing iterations with IDs higher than the argument squash the iterations; the others finish off and commit their iterations. Then, all processors synchronize. 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 nodes, then the undo log is also distributed across multiple nodes. Therefore, each processor can restore the data from the Undo Buffer in its own node. There is no overlap between the locations stored in Undo Buffers in different nodes. However, the restoration of the data from a given Undo Buffer is necessarily done in decreasing order of iteration number. This is required to ensure that the newer updates are overwritten by the older updates. The restoration stops when we reach the sector of an iteration whose ID is lower than the argument of the cross-processor interrupt. At this point, the whole Pointer Cache and Undo Buffer can be recycled.

For the data restoration to work well, two issues must be addressed. Recall that the undo log contains physical, not virtual addresses. Consequently, for lowest overheads, the restoration must be performed by processes that can write directly to physical addresses. This ordinarily implies some type of supervisory mode. In addition, the mapping of the pages that hold the array under test must not change from the time when the undo log is generated until the time when it is read. Otherwise, the restoration would be updating wrong locations. One way to solve this problem is to pin the array-under-test pages in memory during the execution of the loop.

Finally, once the restoration is completed, all processors synchronize. The iteration scheduler is modified to give out the two dependent iterations to one processor at the same time. Such a small change will prevent the same dependence violation from occurring again. Then, parallel execution restarts, starting with the iteration that follows *LstCmtIt*.

3.2 Extending APA

As indicated in Section 2.2, the APA fails when a directory controller detects the second iteration of a *pair* of iterations. That second iteration can either be the *read-first* or the *writing* one. In any case, the repair action will simply be to supply the correct value to the *read-first* iteration, namely the value generated by the *writing* iteration. The actual recover operations performed vary slightly depending on whether the second iteration executed is the *writing* or the *read-first* one. We consider each case in turn. In no case, however, does the extended APA need an undo log like the ANPA to enable the recovery. This is because, except for the *pair* of iterations, no transfer of information from iteration to iteration ever occurs. Consequently, iterations can be freely re-executed without first having to undo past updates.

3.2.1 The Writing Iteration is Executed Last

A directory controller detects the *writing* iteration of a *pair* when an array-under-test element receives a write with an iteration ID lower than element's *MaxR1st*. At that point, the controller sends a cross-processor interrupt to all processors passing, as argument, the iteration ID of the write. 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 starts by flushing and invalidating private copies of the array element from all the caches. Then, the update that caused the exception is propagated to the copy of the element in the shared array and, from there, to all the copies of the element in private arrays. In reality, we would only need to update the copy in the private array that will be used by the *read-first* iteration. However, we may avoid future dependence violations if we update all private copies. In addition, we also clear the *MaxR1st* and *MinW* fields of the element in the shared array and the *PMaxR1st* and *PMaxW* fields of the element in all private arrays. In effect, we are re-initializing that array element.

Ideally, the only remaining thing to do is to re-execute the corresponding *read-first* iteration of all the *pairs* that the *writing* iteration belongs to. Unfortunately, it may be impossible to identify all such *read-first* iterations. The reason is that each processor keeps in *PMaxR1st* only the *highest* ID of any *read-first* iteration for that element that the processor has executed. Lower IDs have been lost.

Consider, for example, Figure 5-(b), where processor *P1* executes *writing* iteration 2 after *PMaxR1st* for *P1*, *P2* and *P3* have already been set to 0, 6 and 5 respectively. We know that iterations 2-5 and 2-6 form *pairs*. However, we do not know whether any processor executed a *read-first* iteration in 3 or 4, creating *pairs* 2-3 or 2-4. Consequently, our solution is to be conservative and restart the *parallel execution* from the iteration that follows the *writing* one.

3.2.2 The Read-First Iteration is Executed Last

A directory controller detects the *read-first* iteration of a *pair* when an array-under-test element receives a read with an iteration ID higher than element's *MinW*. At that point, the controller sends a cross-processor interrupt to all processors, passing as argument the iteration ID of the read. As usual, the processors executing iterations with higher IDs squash their iterations while the others finish off their iterations. All processors then synchronize.

Ideally, the repair would involve what we described in Section 3.2.1: cache flushing and invalidation, and global state update for the array element that was written in the *writing* iteration. Then, we would re-execute the *read-first* iteration.

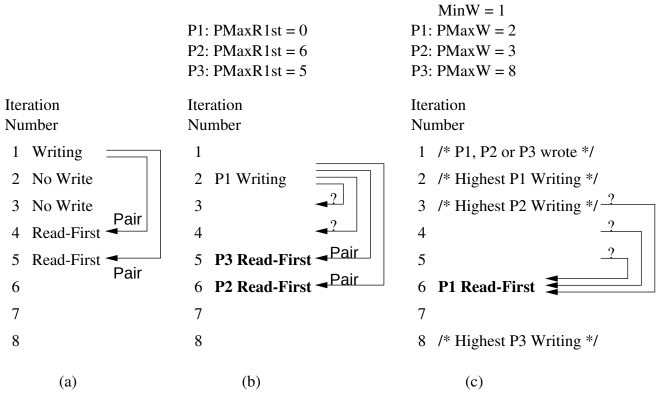

Figure 5: Examples of pairs (Chart (a)). In reality, the system keeps only limited information that we can use to identify the pairs (Charts (b) and (c)).

Unfortunately, we may not know what the correct *writing* iteration is. The reason is that each processor keeps in *PMaxW* only the *highest* iteration ID of any *writing* iteration for that element that the processor has executed. Lower IDs have been lost.

Consider, for example, Figure 5-(c), where processor *P1* executes *read-first* iteration 6. *MinW* is 1, and *PMaxW* for *P1*, *P2* and *P3* are 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 *P3* wrote in iterations 4 or 5. We only keep the highest *writing* iteration for each processor.

Consequently, the *writing* iteration is identified based on the following rule. We compute the highest *PMaxW* for the element across all processors. Let us call this value *mxw*. We have two cases: *mxw* is smaller than the *read-first* iteration or not. In the first case, the *writing* iteration is *mxw*. In this case, as described in Section 3.2.1, we perform the cache flushing and invalidation, and the global state update for the array element that must be communicated from *mxw* to the *read-first* iteration. Then, all processors synchronize and parallel execution resumes starting from the *read-first* iteration.

In the second case, the potential *writing* iterations are: the highest *PMaxW* that is smaller than the *read-first* iteration, and all the iterations that follow it until the *read-first* iteration. We call these iterations that follow it, the *hidden* ones. In Figure 5-(c), the *hidden* iterations are 4 and 5. In this case, the *hidden* iterations are re-executed *in parallel*. After that, based on the *PMaxW* values, we can determine which iteration was the true *writing* one, and can perform the cache flushing and correct global state update of the array element as in Section 3.2.1. Then, all processors synchronize and parallel execution resumes starting from the *read-first* iteration.

4 Experimental Results

We have evaluated the proposed hardware through simulations [17, 19]. We use MINT [15] to perform execution-driven simulations of a 16-node CC-NUMA shared-memory multiprocessor. Each node has a 200-MHz 4-issue dynamic superscalar with 4 integer, 2 load-store and 2 floating-point functional units and a 32-entry instruction queue. Each node has a 32-Kbyte L1 cache and a 512-Kbyte L2 cache. Both caches are direct-mapped and have 64-byte lines. The cache sizes are

Application	Loop Name	Time (% of Tseq)	Number of Invocations	Iterations/ Invocation	AUT Size (Kbytes)	Algorithm
Track	*nlfilt_do300*	40.9	56	480	72.4	ANPA: Base & Ext.
Euler	*dflux_do[100,200], psmoo_do20 eflux_do[100,200,300]*	89.9	120	59,863	703.1	ANPA: Base & Ext.
Dsmc3d	*move3_goto100*	32.8	80	383,107	158.4	ANPA: Base & Ext.
Adm	*run_do[20,30,40,50,60,100]*	20.6	900	32 or 64	803.1	ANPA & APA: Base & Ext.

Table 1: Loop characteristics. AUT stands for array under test.

selected so small to match the scaled-down working sets of the applications that we run. The caches are kept coherent with a DASH-like cache coherence protocol [10]. Each node has part of the global memory and directory. Contention is modeled in the whole system except in the global network, which is abstracted away as a constant latency. The unloaded round-trip latencies to access the L1 cache, L2 cache, local memory, 2-hop remote memory and 3-hop remote memory are 1, 12, 60, 208 and 291 cycles on average respectively. Finally, shared pages are allocated round-robin across the different memories, while private pages are allocated locally.

For the simulations, we have extracted and measured important loops from well-known codes. Table 1 lists the loops with their weight relative to the total *sequential execution time* of the complete application on an SGI PowerChallenge (% of Tseq). The table also shows the number of times that the loops are invoked, their average number of iterations per invocation, the size of the arrays under test and the algorithm used. *Track* and *Adm* are Perfect Club codes [1], while *Euler* and *Dsmc3d* are HPF-2 applications [5]. They all have been pre-processed with the Polaris parallelizing compiler [2]. All these loops have cross-iteration dependences except *Adm*, which turns out to be fully parallel.

Figure 6 compares the execution time of the loops under various algorithms. *Serial1* and *Serial16* correspond to the plain serial execution of the loops with the application data allocated in 1 node or in 16 nodes respectively. Since these loops are part of applications to be executed on a parallel machine, *Serial16* is more realistic. *ANPA* corresponds to the base ANPA. Since *Adm* uses both the ANPA and the APA, we label it *A(N)PA*. Finally, *Extended* corresponds to the extended ANPA and APA. Time is divided into instruction execution (*Useful*), stall due to pipeline hazards and synchronization (*Haz+Sync*) and stall due to memory latency (*Mem*). The bars that are higher than 200 are broken and labeled with the correct height. All bars are normalized to *Serial1*.

From the figure, we see that the extended ANPA/APA perform better than the base ANPA/APA and much better than the serial execution. An in-depth analysis of the results can be found in [17].

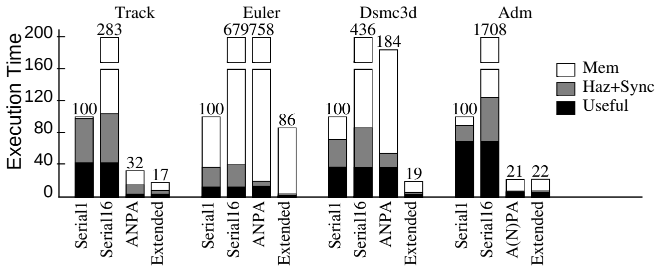

Figure 6: Execution time under various algorithms.

5 Summary

Hardware-based speculative parallelization of non-analyzable codes on DSM multiprocessors is challenging. In this paper, we briefly presented two schemes for speculative parallelization that we applied to loops. In the first, base scheme, if a dependence violation is detected at run time, execution is interrupted, the state is rolled back in software to the beginning of the loop, and the loop is re-executed sequentially. In the second, extended scheme, if a dependence violation is detected at run time, execution is interrupted, the state is repaired in software on the fly, and parallel execution is restarted from that point on. The first scheme is targeted to loops where most of the executions turn out to run fully in parallel, while the second scheme is targeted to loops that have a modest number of cross-iteration dependences. Simulation results suggest that the schemes are promising.

References

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

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

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

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

[5] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

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

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

[8] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[9] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Proceedings of the 1998 International Conference on Supercomputing*, July 1998.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

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

[12] L. Rauchwerger, N. Amato, and D. Padua. A Scalable Method for Run-Time Loop Parallelization. *IJPP*, 26(6):537–576, July 1995.

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

[14] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[15] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

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

[17] Y. Zhang. Hardware for Speculative Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, expected March 1999.

[18] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[19] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.