Hardware for Speculative Reduction Parallelization and Optimization in DSM 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/

1 Introduction

Compiler-driven parallelization of codes has advanced significantly in this decade. 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*, *g* and *k* 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 (the references in S1,S2,S3). Consequently, the code executes serially.

```
do i = 1,n
S1: A(f(i)) = ...
S2: ... = A(g(i))        A: Array under test
S3: A(r(i)) = A(r(i)) + e(i)
enddo
```

Figure 1: Loop that cannot be compiler analyzed.

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.

To run these codes in a fully- or partially-parallel manner, software approaches based on a *inspector-executor* pair have been proposed ([17] 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.

Recently, we have introduced a new framework for speculative parallelization in hardware [22]. The scheme is based on a software-based run-time parallelization scheme that we proposed earlier [15]. 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, the correct access order is enforced and parallel execution is restarted.

A very frequent cause of cross-iteration data dependences are reduction operations. Reductions (also known as updates) are at the core of a very large number of algorithms and applications - both scientific and otherwise. In Fig. 1 the operation in statement S3 performs such a reduction operation on the array A. The data dependences associated with such reductions can be removed only through an algorithm substitution. The generally adopted solution is to accumulate in private storage during a parallel loop and then, in a post parallel phase, update the shared array with the partial results obtained by each processor. The final cross-processor update of the shared data structures with the partial contribution of each processor is very time consuming and, in the case of sparse access patterns, waistful. For example, in the case of a reduction operation on an array, the post-parallel loop phase requires the traversal all private arrays on all processors in order to accumulate and update the shared memory. In a system with p processors we need to collect p contributions from *every* private array element (of dimension n), compute the total contribution and update the shared array. It takes $O(np)$, which when performed in parallel results in an overall slowdown (due to long inter-processor latencies). In the case of sparse access patterns the execution remains proportional to array dimensions rather than scaling with the number of actually accessed elements. Using compacted private data structures (e.g., hash tables) does not necessarily improve the actual performance because each each access is more expensive (only asymptotically it does).

This technique can be applied by the compiler only in the case when it can be proven that the reduction operands (e.g., the elements of array A referenced in S3) are accessed only by the reduction statement and nowhere else in the body of the loop. In the case irregular programs with input dependent access patterns, the compiler cannot prove the validity of the reduction parallelization transformation. For example (Fig. 1), if the value of the subscript functions $f(i), g(i), r(i)$ are not statically anaylzable then it is not possible to prove that the elements referenced in S3 are not also referenced (in some iteration) in S1 or S2 and therefore no parallelization transformation is legal.

In this paper we propose to extend the hardware scheme presented in [19] to:

- speculatively apply the reduction parallelization transformation and validate it during parallel execution for statically unanalyzable reductions (that cannot be legally transformed at compile time) and
- to optimize the final cross-processor update of the shared memory such that most of the time associated with this activity can be overlapped with the first phase of the reduction computation.

With the proposed extensions, when a dependence violation is encountered that is due to a reduction operand being referenced outside a reduction statement, the state is repaired on the fly and parallel execution is resumed from that point on. No backtracking (squashing of previously executed iterations) is necessary. Simulation results suggest that the proposed hardware can parallelize efficiently loops with reductions which would otherwise have to be run serially. Moreover, the hardware support improves the performance of any reduction parallelization by at least 50% over the software implemented solution.

The overall scheme presented in [22] is somewhat related to speculative parallelization inside a multiprocessor chip [7, 8, 9, 16], 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 [7, 8, 9, 16]. The specific support for reduction validation is, to the best of our knowledge, original while the reduction optimization method is related to [11].

In the following, we briefly describe the speculative parallelization scheme that was introduced in [22], then present the new extensions for loops with reductions, and finally evaluate the new scheme.

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

2 DSM Hardware for Speculative Parallelization

In previous work, we proposed a scheme to speculatively execute non-analyzable loops in parallel in a DSM machine [20]. The idea is to extend the directory-based cache-coherence protocol of the machine to detect, in hardware, any violation of a cross-iteration dependence. The loop is executed in parallel. If the hardware detects such a violation, then the state is rolled back to a safe state, the correct access order is enforced and parallel execution is restarted.

The 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 [20].

We have developed the privatization and the non-privatization algorithms. In essence the two algorithms are based on similar hardware support but augment the cache coherence protocol either to verify the cross-iteration independence of references to a shared array (ANPA), or to validate the per-processor privatization of a shared array (APA). Both array attributes (independent or privatizable) allow parallel execution of the iterations of a loop. The choice between the strategies is based on compiler analysis and heuristics.

In the interest of brevity we will focus here only the reduction parallelization based on the advanced non-privatization algorithm (ANPA). The basics of the privatization algorithm and its associated reduction validation are explained in more detail in [22, 23, 20, 18].

We assume the general case of a dynamically-scheduled loop. The non-analyzable variables that may cause a dependence violation we call *arrays under test*.

2.1 Advanced Non-Privatization Algorithm (ANPA)

The main idea of 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.

Ideally, out-of-order execution of iterations that write would be fine as long as we skipped the second update to the data and *MaxW*. In practice, however, all writes must be performed and recorded in order if, as shown in Section 3, we want to repair dependence violations on the fly.

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. 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 further possible optimizations, are described in detail in [23].

2.2 Sliding Commit

The algorithm is designed to detect data dependence violations for loops with relatively infrequent dependences and which are speculatively executed in parallel. Because violations can occur we need to provide a safe state form which execution can be restarted after a violation has been detected. ANPA 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). 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 several 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). *LstCmtIt* holds the value of the lowest committed iteration, and is advanced as soon as all lower iterations in the *ItSt* are in the *finished* state.

After a processor advances *LstCmtIt*, it passes the new value to all directory controllers. The latter, in hardware, deallocate the Pointer Cache entries and Undo Buffer sectors of the committed iterations. The space can now be reused. In our experiments, 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 [23].

2.3 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 3: the Pointer Cache, the Free Sector Stack and the Undo Buffer. The latter 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.

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,

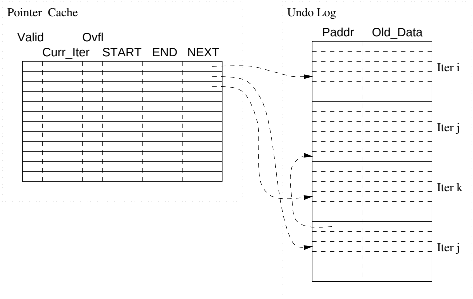

Figure 3: Directory data structures for undo logs.

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 3. 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. It is described in [23].

2.4 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 that the undo log is generated until the time that 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.

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.

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. Speculative Reduction Parallelization

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 4-(a). The operator $\otimes$ is exemplified by the $+$ operator, the access pattern of array $A(:)$ is *read, modify, write*, The function performed by the loop is to add a value computed in each iteration to the value stored in $A(:)$. This type of reduction is sometimes called an *update*.

There are two tasks required for reduction parallelization: *recognizing the reduction variable*, and *parallelizing the reduction operation*. (In contrast, privatization needs only to recognize privatizable variables by performing data dependence analysis, i.e., it is contingent only on the access pattern and not on the operations.) 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 [5, 24]. 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 [10, 12]. 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 4-(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 [24].

```
     do i=1, n
        do j = 1, m
S1:        A(j) = A(j) + exp()
        enddo
     enddo
```

(a)

```
     do i = 1, n
S1:     A(K(i)) = .......
S2:       ............ = A(L(i))
S3:     A(R(i)) = A(R(i)) + exp()
     enddo
```

(b)

```
     doall  i = 1, p
       pA(1:n) = 0
     enddoall
     doall   i = 1, n
S1:     A(K(i)) = .......
S2:        ............ = A(L(i))
S3:     pA(R(i)) = pA(R(i)) + exp()
     enddoall
     doall  i = 1, p
        A(1:n) = A(1:n) + pA(1:n)
     enddoall
```

(c)

Figure 4: (a) Simple example of reduction, (b) Reduction needing run-time validation, (c) Reduction transformed for parallel execution

In the cases where data dependence analysis cannot be performed at compile time, reductions have to be validated at run-time. by extending the capabilities of the previously presented hardware for loops with dependences. For example, although statement `S3` in the loop in Figure 4-(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`. Thus, our hardware scheme must check at run–time that there is no intersection between the references in `S3` and those in `S1` and/or `S2`; 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 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
	if (NRED == 1)
	  FAIL
	else
	  RED = 1
	access data

(a): Reference outside the reduction statement.	(b): Reference from 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 now only present ANPAR, which is the combination of ANPA and the reduction verification algorithm. The combination of APA and the reduction verification algorithm, which we call APAR, is similar. It can be found in [21].

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 *MaxR* or *MaxW* are not zero for the element.

Consequently, for accesses outside the reduction statement, we simply test the *RED* bit and, if zero, set *MaxR* or *MaxW* as in Figure 2.For accesses inside the reduction statement, we test *MaxR* or *MaxW* 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 *MaxR*, *MaxW* 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 4-(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.

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*. More details of these algorithms can be found in [21].

4 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 $s * logp$ 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.

4.1 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.

4.2 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, in a similar way as ANPA recovers from shared data dependence violations. We differentiate two cases in which reduction parallelization can fail.

If a read is performed on a reduction operand outside the reduction statement, then the request will be forwarded to its home node. There all private partial results from the undo logs of all earlier iterations will be merged into the shared memory and the result send to the requester.

If a write is performed on a reduction operand outside the reduction statement, then an invalidation is send to all processors who have the line. All processors with smaller timestamps will kill their partial results (e.g., sums) since they are no longer needed, and the shared data will be written with the new value at the home node (a re-initialization of the shared reduction data structure). From the undo logs of the processors with higher timestamps we will collect all partial results obtained previous to the CTS (the current iteration executing the write outside the reduction statement) and subtract it from the current value stored in their private partial sums, i.e., we will undo the contribution of all iterations previous to the violating iteration.

5 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.

5.1 Simulation Environment

Our evaluation is based on execution-driven simulations of a CC-NUMA shared-memory multiprocessor using Tangolite [6]. The modeled multiprocessor has the hardware support of the basic design [20] 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 [13]. 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.

5.2 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. *Spice* is a Perfect Club [2] code, *Euler* and *Dsmc3d* are HPF-2 applications [4], and *Rmv* is a Spark98 kernel [14].

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.

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. *Spark98* is a sparse matrix and dense vector multiplication C kernel. *Rmv* needs only reduction optimization.

5.3 Evaluation

We have used *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.

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 5 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.

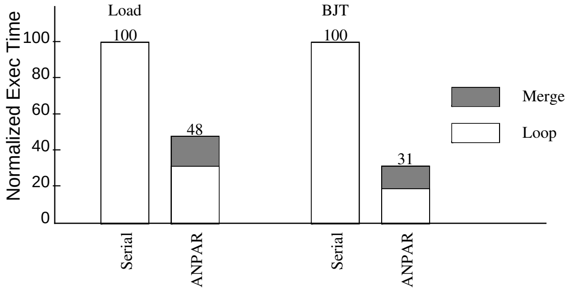

Figure 5: Reduction Verification (8 proc).

5.4 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 6 and 7 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.

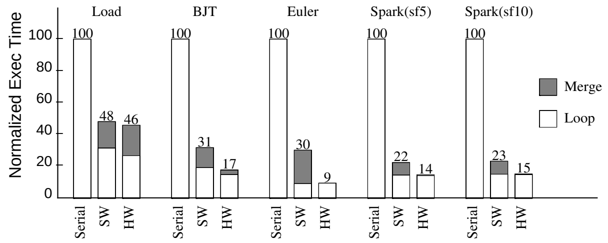

Figure 6: Reduction Execution Time Breakdown (8 proc).

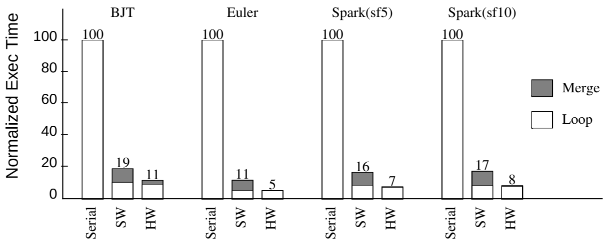

Figure 7: Reduction Execution Time Breakdown (16 proc).

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.

6 Summary

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 can not 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 statically non-analyzable reductions. When the speculation fails we can locally repair the state and then restart parallel execution from that point on. Additonally we provide a scheme for the optimization of parallel reduction which is applicable to both compile time and run-time verified reductions. Performance is mainly enhanced by dramatically reducing the time of the final cross-processor update of the shared memory with the private, per processor contributions. If performed in software, this phase is proportional with the data size (in the dense case) or the data structure dimensions in the worst, sparse case. In our scheme the time is proportional only to the cache size, which is, usually, relatively small. Simulations indicate good speedups relative to sequential execution. The hardware support for reduction optimizations brings, on average, 50% performance improvement.

References

[1] R. Asenjo, E. Gutierrez, Y. Lin, D. Padua, B. Pottenger, and E. Zapata. On the automatic parallelization of sparse and irregular fortran codes. Technical Report 1512, Center for Supercomputing Research and Development, February 1997.

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

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

[4] I. Duff, R. Schreiber, and P. Havlak. Hpf-2 scope of activities and motivating applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[5] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

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

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

[8] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS8*, 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] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 869–876, August 1986.

[11] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *ASPLOS6*, pages 208–218, 1994.

[12] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

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

[14] D. O'Hallaron, J. Shewchuk, and T. Gross. Architectural implications of a family of irregular applications. Technical Report CMU-CS-97-189, School of Computer Science, Carnegie Mellon University, November 1997.

[15] 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.

[16] J. G. Steffan and T. C. 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.

[17] 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.

[18] Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors. Technical Report 1536, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, July 1997.

[19] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of teh 4th International Symposium on High Performance Computer Architecture 1998, (HPCA-4)*, pages 162–173, February 1998.

[20] 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.

[21] Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative Parallel Execution of Loops with with Cross-Iteration Dependences in DSM Multiprocessors. Technical report, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, July 1998.

[22] Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture(HPCA-5)*, January 1999.

[23] Ye Zhang. *DSM Hardware for Speculative Parallelization*. PhD thesis, Department of ECE, University of Illinois, Urbana, Illinois, January 1999.

[24] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.