SPECULATIVE PARALLELIZATION FOR PARTIALLY PARALLEL LOOPS

A Thesis

by

FRANCIS HOAI DINH DANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Computer Science

SPECULATIVE PARALLELIZATION FOR PARTIALLY PARALLEL LOOPS

A Thesis

by

FRANCIS HOAI DINH DANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Nancy Amato
	Marvin L. Adams
Head of Department,	Valerie Taylor

May 2007

Major Subject: Computer Science

ABSTRACT

Speculative Parallelization for Partially Parallel Loops. (May 2007)

Francis Hoai Dinh Dang, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. In our previous work, we have speculatively executed a loop as a `doall`, and applied a fully parallel data dependence test to determine if it had any cross–processor dependences. If the test failed, then the loop was re–executed serially. While this method exploits `doall` parallelism well, it can cause slowdowns for loops with even one cross-processor flow dependence because we have to re-execute sequentially. Moreover, the existing, partial parallelism of loops is not exploited.

We demonstrate a generalization of the speculative `doall` parallelization technique, called the Recursive LRPD test, that can extract and exploit the maximum available parallelism of *any* loop and that limits potential slowdowns to the overhead of the run-time dependence test itself. In this thesis, we have presented the base algorithm and an analysis of the different heuristics for its practical application. To reduce the run-time overhead of the Recursive LRPD test, we have implemented on-demand checkpointing and commit, more efficient data dependence analysis and shadow structures, and feedback-guided load balancing. We obtained scalable speedups for loops from Track, Spice, and FMA3D that were not parallelizable by previous speculative parallelization methods.

ACKNOWLEDGMENTS

I am very fortunate to have had Dr. Lawrence Rauchwerger as my advisor and for giving me an opportunity to do research under his guidance. Without his support, inspiration and technical direction, this thesis would not have been possible. He was a great role model for me with his dedicated efforts in his research. All those conversations with him at his office and outside the Bright building guided my research and my life.

I would like to thank the members of my advisory committee, Nancy Amato and Marvin Adams for reading my thesis and offering their advice and help.

I have had the opportunity to work with excellent students and researchers at Texas A&M, especially the previous and current members of the Parasol group: Julio Carvallo de Ochoa, Guobin He, Tao Huang, Alin Jula, William McClendon III, Lidia Onica, Silvius Rus, Steven Saunders, Timmie Smith, Nageswar Tagarathi, Gabriel Tanase, Nathan Thomas, Hao Yu, and Dongmin Zhang. I want to thank them for the great environment filled with excitement, inspiration, and happiness.

Finally, I would like to thank my family and friends for their support and encouragement during my years at Texas A&M. I can only hope I can repay them in kind.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION AND MOTIVATION

To achieve a high level of performance for a particular program on today's supercomputers, software developers are often forced to tediously hand–code optimizations tailored to a specific machine. Such hand–coding is difficult, increases the possibility of error over sequential programming, and the resulting code may not be portable to other machines. The only avenue for bringing parallel processing to every desktop is to make parallel programming as easy (or as difficult) as programming current uniprocessor systems. This can be achieved through good programming languages and, mainly, through automatic compilation. Restructuring, or parallelizing, compilers can detect and exploit parallelism in programs written in conventional sequential languages or parallel languages (e.g., HPF). Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [1, 2]), current parallelizing compilers cannot extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. Typical examples are complex simulations such as SPICE [3], DYNA–3D [4], GAUSSIAN [5], and CHARMM [6]. Run–time techniques can succeed where static compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run–time dependent conditions, and subscript expressions can all be analyzed unambiguously at run–time.

There are several known run-time parallelization methodologies. However, they

---

The journal model is *IEEE Transactions on Parallel and Distributed Systems.*

each have their strengths and weaknesses. Speculative techniques work well for fully parallel loops but incurs a slowdown proportional to the speculative execution and its overhead for sequential and partially parallel loops. Inspector/executor methods can parallelize partially parallel loops but their major limitation is the assumption that a proper inspector exists. If a dependence cycle exists between data and address computation of the shared arrays, then a proper side-effect free inspector of the traversed address space cannot be obtained (it would be most of the analyzed loop itself). Furthermore, these inspectors often require large data structures proportional to the memory reference trace.

In this thesis, we will present a run-time technique based on speculative parallelization that can be applied to any loop and removes the limitations of previous techniques. We will show several strategies for applying this technique to partially parallel loops and model the performance of these strategies. We will also present several optimizations to reduce the cost in using this technique.

This thesis is structured as follows:

- Review of run-time parallelization approaches (Chapter II)
- Review of speculative parallelizing loops at run-time (LRPD test) and transformation of LRPD test to R-LRPD test (Chapter III)
- Strategies for applying the R-LRPD test (Chapter IV)
- Modeling the performance of the redistribution strategy (Chapter V)
- Discuss the implementation and optimization of the R-LRPD test (Chapter VI)

- Present and discuss some experimental results for codes from the PERFECT and SPEC2000 benchmark suites (Chapter VII)
- Summarize the techniques that have been developed and discuss possible extensions and improvements (Chapter VIII)

CHAPTER II

RUN TIME PARALLELIZATION

Loop parallelization is one of the most effective optimizations for scientific applications on today's supercomputers. A loop can be safely executed in parallel if and only if its later iterations do not use data that was computed in its earlier iterations, i.e. there are no flow dependences. This transformation and others like it (e.g. privatization and reduction parallelization) are checked at compile time to determine the safety of their application through data-dependence analysis. When compiler analysis is not possible, the access pattern is analyzed at run-time either before (inspector/executor) or after (speculation) the execution of the loop. A number of techniques [7, 8, 9, 10, 11, 12, 13, 14] have been developed to detect and exploit loop level parallelism at run-time. Several representative techniques are:

- A speculative execution technique to detect fully parallel loops [7, 13, 11, 14])
- An inspector/executor method to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel) [10]
- Compiler techniques that generate simpler run-time tests that evaluate only a small set of run-time values instead of an exhaustive memory reference analysis [11, 12].
- A DOACROSS mechanism for enforcing data dependences using synchronizations whose locations are determined by performing an inspector on each iteration [15].

Speculative execution methods such as the LRPD test [7, 11] execute a loop as a doall and records the memory references for any statically unanalyzable data.

Subsequently, an analysis performs a data dependence test to determine if the execution was correct. If not, the program state is restored and the loop is re-executed in a sequential manner. While for fully parallel loops the method performs very well, partially parallel loops experience a slow-down equal to the speculative parallel execution time and its overhead.

Gupta and Nim [11] proposed methods to address some of the deficiencies in the LPD test (the LRPD test without reduction parallelization). If during speculative execution, a processor discovers an uncovered read, it waits until all lower numbered processors have completed execution using a signal-wait mechanism. This strategy avoids the potentially expensive cost of restoring program state for incorrect speculative execution but it can lead to serialization of the loop. In another strategy, they set an error flag if a cross processor flow dependence is detected during speculative execution. If this error flag is detected at the beginning of a loop iteration, the loop is re-executed sequentially. This method shares the same flaw at the LRPD test, just one cross processor dependence can lead to sequential re-execution.

Speculative execution methods that use software based data forwarding to resolve cross processor data dependences and thread squashing attempt to reduce the cost of restoring program state for incorrect speculative execution. Cintra and Llanos [13] used a blocked scheduled sliding window similar to our method. For an uncovered read, the processor searches lower numbered processors for the latest written version of the array element in their private copy of the array. For a store operation, the writing processor triggers a squash operation on any higher numbered processor that has performed a read operation on the corresponding array element. Also, the commit operation (after loop execution) is only performed by the first (non-speculative) processor. These strategies introduce additional overhead for each memory reference

to avoid the cost of speculative execution failure. However, these strategies could scale poorly for loops with many memory references. Also the difference between the overheads of the additional operations per memory reference and the restoration of program state upon failed speculative execution could vary per loop. Rundberg and Stenstrom [14] grouped the original shared variable under test, the private copy of the variable processor, and shared shadow arrays. This increases locality but at the potential cost of either additional synchronization for each memory reference or possible contention for the shared shadow arrays in a shared memory system using cache coherence.

Gupta [11] and Rus [12] developed comprehensive compile time techniques that combine control flow and data flow analysis to generate run-time tests that are more efficient that an exhaustive data dependence test at run-time. The tests evaluate a small set of run-time values to determine if a loop is parallel or not. These techniques can reduce run-time overhead dramatically, i.e. from proportional to the number of dynamic memory references to proportional to the size of the data (or even constant). However, they do not offer a solution for partially parallel loops.

For loops which were presumed to be partially parallel, an inspector/executor technique [10] could be applied. The inspector is extracted from the actual loop and records the relevant memory references. Then a sorting based technique is used to construct the iteration dependence graph of the loop and schedule the iterations in topological order for the executor. However, if there is a dependence cycle between data and address computation of the shared arrays, then the address inspector may not be parallelizable and/or side-effect free. The resulting inspector would contain most of the analyzed loop. Furthermore, this method requires large additional data structures to store the trace of memory references.

Kazi and Lilja [15] parallelize partially parallel loops by using a DOACROSS mechanism for enforcing data dependences, executing in private storage and committing in order, after any possibility of further dependence violation has passed. This method requires a setup phase for every iteration during which all potential dependence causing addresses are pre-computed and then broadcast to all processors. This information is used to set tags for future advance/await type synchronizations. This method can never properly exploit large amounts of parallelism and does not remove the need for address pre-computation, i.e., an inspector per iteration. Thus, it cannot parallelize loops in which address and data depend upon one another.

CHAPTER III

SPECULATIVE PARALLELIZATION

This chapter will first describe the speculative LRPD test. A more detailed description can be found in [7]. Then we will show how to use the LRPD test recursively for partially parallel loops.

A. The LRPD Test

The LRPD test [7] speculatively executes a loop as a doall and subsequently tested whether there were any data dependences. If any dependences were found, then the speculative execution was incorrect and the loop was re-executed sequentially. To qualify more loops as parallel, array privatization and reduction parallelization are speculatively applied and their validity tested after loop termination. *Privatization* creates, for each processor executing the loop, privates copies of the program variables. A shared variable is privatizable if it is always written in an iteration before it is read. A *reduction variable* is a variable used in one operation of the form $x = x \otimes exp$, where $\otimes$ is an associative and commutative operator and x does not occur in exp or anywhere else in the loop. There are known transformations for implementing reductions in parallel [16, 17, 18]. For brevity, reduction parallelization is not presented in the following discussion; it is tested in a similar manner as independence and privatization. The algorithm for the LRPD test is shown in Fig. 1.

Consider the *DO* loop for which the compiler cannot statically determine the access pattern of the shared array A in Fig. 2(a). The LRPD test allocates shadow array A_w for marking write accesses, shadow array A_r for marking read accesses, and shadow array A_{np} for marking non-privatizable elements. The loop is instrumented

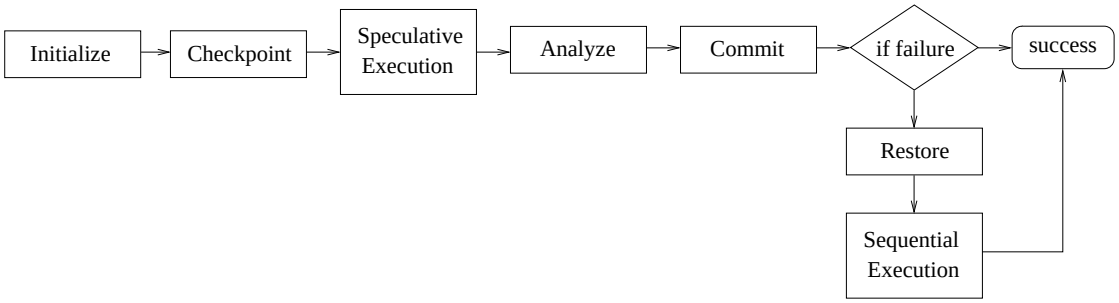


Fig. 1. LRPD algorithm.

with code, shown in Fig. 2(b), that records each reference of A to the shadow array based on specific rules. Any shared arrays that may be modified during execution and are not under data dependence test are copied to checkpoint arrays before the loop is executed. The resulting marks in the shadow array can be seen in Fig. 2(c). For the first write access of an element of A for an iteration, the corresponding element in A_w is marked. If, during any iteration, an element of A is read but not written, then the corresponding element in A_r is marked. If the array element has not been written in this iteration before the read access, then the corresponding element in A_{np} is marked.

After the speculative parallel execution, a post-execution analysis determines whether there were any cross processor dependences for the array A. If $any(A_w(:) \bigwedge A_r(:))$ [1] is true, then at least one element of A is read and written in different iterations (a flow or anti dependence) that was not removed by privatizing A. If $any(A_{np}(:))$ is true, then A is not privatizable since at least one element is read before being written in an iteration. If A_{tw}, the total number of write accesses marked during the speculative parallel execution, is not equal to A_{tm}, the total

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

number of write marks in A_w, then there is at least one element that was written concurrently in different iterations (an output dependence). However, these output dependences can be removed by privatizing A if $any(A_{np}(:))$ is false. If the test fails due to any unresolved dependences, then any modified shared data is restored from checkpoint storage and the loop is re-executed sequentially. If there were no detected cross-processor dependences, then any modified private copies of A are committed by determining the last written value.

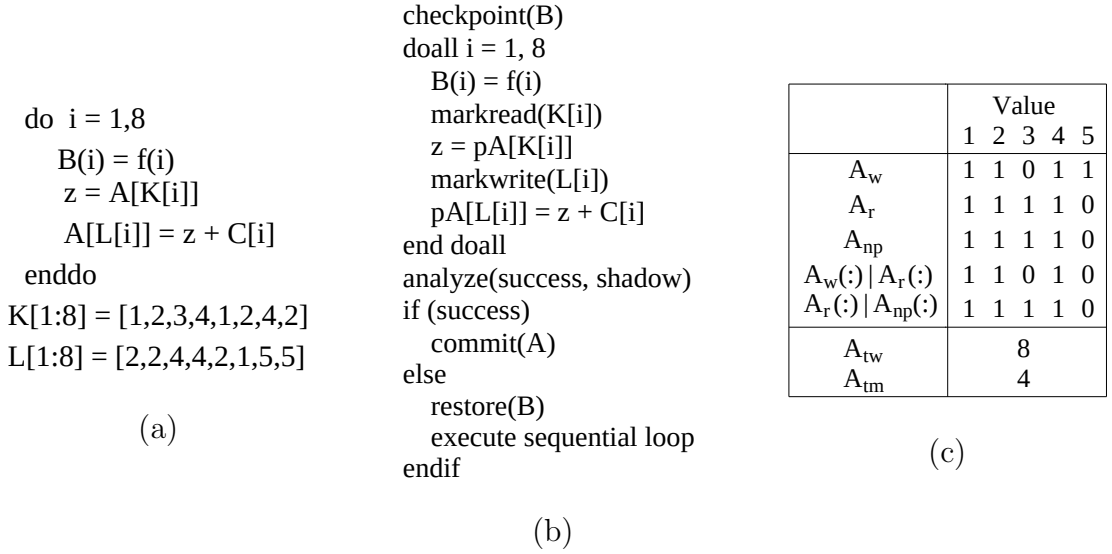


Fig. 2. An example of the LRPD test on a `DO` loop.

The LRPD test is fully parallel and requires $O(r/p + log(p)$, where p is the number of processors, and r is the total number of references made to the array being tested for data dependences in the loop. This method works well for fully parallel loops but loops with even one cross-iteration flow dependence would have a slowdown proportional to the speculative parallel execution time. Moreover, the existing partial parallelism is not exploited.

B. The Recursive LRPD Test (R-LRPD)

We propose a technique to extract the maximum available parallelism from a partially parallel loop that removes the limitations of previous techniques. It can be applied to any loop and will require less memory overhead. The new technique will transform a partially parallel loop into a sequence of fully parallel loops. At each stage, we speculatively execute all remaining iterations in parallel with array privatization, reduction parallelization, and static blocked scheduling.

To reduce the overhead and qualify even more dependent loops as parallel, we test the *copy-in* condition instead of the privatization condition. That is, instead of checking every Read is covered by a Write to a memory location, i.e., a $((Write|Read)*)$ pattern, we check for a reference pattern of the form $(Read * |(Write|Read)*)$. If the condition holds, then the memory location can be transformed for safe parallel execution by initializing its private storage with the original shared data. In practice, we need to test at run-time if the latest consecutive reading iteration (maximum read) is before the earliest writing iteration (minimum write) for all references in a loop.

In addition, we use a *processor–wise* test which checks only for cross-processor dependences rather than loop carried dependences. In a processor-wise test (always preferable), we have to schedule the loop statically (blocked). While this is a limitation, it also simplifies the tested conditions: Highest reading processor <= lowest writing processor. The initialization of the private arrays can be done either before the start of the speculative loop or preferably as an 'on-demand copy-in' (read-in if the memory element has not been written before). Thus the only reference pattern that can still invalidate a speculative parallelization is a flow dependence between

processors (a write on a lower processor matched by a read from a higher processor). For this criteria, only two bits (write and read) are needed for each element in the shadow array that corresponds to the element of the array under data dependence test.

We now make the crucial observation that in any block-scheduled loop executed under the processor-wise LRPD test, the iterations that are less than or equal to the source of the first detected dependence arc are always executed correctly. Only iterations larger or equal to the earliest sink of any dependence arc need to be re-executed. This means that only the remainder of the work (of the loop) needs to be re-executed, as opposed to the original LRPD test which would re-execute the entire loop sequentially.

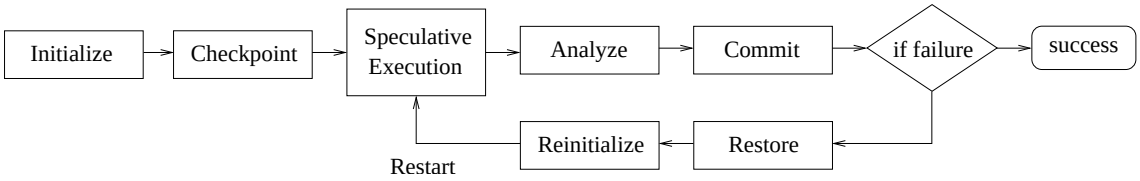


Fig. 3. R-LRPD algorithm.

To re-execute the fraction of the iterations assigned to the processors that may have worked off erroneous data, we repair the unsatisfied dependences by initializing their privatized memory with the data produced by the lower ranked processors. Alternatively, we can commit (i.e., copy-out) the correctly computed data from private to shared storage and use on-demand copy-in during re-execution. We also restore any shared data that was modified by iterations that need to be re-executed from the checkpoint storage. We can then re-apply the LRPD test recursively on the remaining processors, until all processors have correctly finished their work. We call this application of the LRPD test the Recursive LRPD test as shown in Fig. 3.

CHAPTER IV

R-LRPD STRATEGIES

There are several options for implementing the R-LRPD test. They differ in the manner in which the iterations are assigned to the processors. The simplest strategy is presented first and then other strategies with potential optimizations are described. Experimental results will compare the different strategies.

Consider again the `DO` loop for which the compiler cannot statically determine the access pattern of a shared array `A` in Fig. 2(a). A shadow array is allocated for each array under test for marking the read and write accesses. Fig. 4(b) shows the loop augmented with marking code and enclosed in a `while` loop that repeats the speculative parallelization until the entire loop is completed successfully. The marking algorithm is as follows:

- Two bits are used for Read and Write respectively.
- If on a processor, the Read occurs before the first Write, then the Read bit is set. Any subsequent write access will not clear the read bit. (Read First)
- If the Write occurs first, then the Write bit is set and any subsequent Read will not set the read bit. (Write First)
- Repeated references of the same type to an element on a processor will not cause a change in the shadow array.

The array A is first privatized. Read-first references will copy-in on-demand the content of the shared array A. Array B, which is not tested (it is statically analyzable), is checkpointed. The result of the marking after the first speculative `doall`

can be seen in Fig. 4(c). After the analysis phase, we copy (commit) the elements of A that have been computed on processors 1 and 2 to their shared counterpart (by taking their last written value). This step also insures that flow-dependences will be satisfied during the next stage of parallel execution (we will read-in data produced in the previous stage). We further need to restore the section of array B that is modified/used in processors 3 and 4 so that a correct state is established for all arrays. (In our simple example this is not really necessary because we would overwrite B).

A. The Non-Redistribution (NRD) Strategy

In the non-redistribution strategy (NRD), the `Re-Init` step in Fig. 4(b) re-initializes the shadow arrays on all processors that have not successfully completed their assigned iterations yet, which is processors 3 and 4 in this case. Then, a new parallel loop is started on these processors for the remainder of the iterations (5-8 in this case). The final state for the example is shown in Fig. 4(c). At this point all data can be committed and the loop finishes in a total of two steps of two iterations each.

B. The Redistribution (RD) Strategy

Instead of re-executing only on the processors that have incorrect data and leaving the rest of them idle (NRD), at every stage the remaining work is redistributed across all processors. There are pros and cons for this approach. Through redistribution of work, all processors are used all the time and thus decreases the execution time of each stage (instead of staying constant, as in the NRD case). The disadvantage is that new dependences may be uncovered across processors which were satisfied before by executing on the same processor. Moreover, with redistribution, there

```
start = newstart = 1
end = newend = 8
success = .false.
Initialize shadow arrays
Checkpoint B(:)
While (.not. success) do
  Doall i = newstart, newend
    B(i) = f(i)
    z = pA[K[i]]
    markread (K[i])
    pA{L{i}} = z + C[i}
    markwrite (L[i])
  End doall
  Analyze (success, start, end)
  Commit (A(start, newstart−1))
  If (.not. success) then
    Compute(newstart, newend)
    Restore B(newstart, newend)
    Re−Init (Shadows, pA)
  endif
End While
```

(a)

(b)

A()	P1 W	P1 R	P2 W	P2 R	P3 W	P3 R	P4 W	P4 R
1	0	1			1	1		
2	1	0			0	1	1	0
3			0	1				
4			1	0			0	1
5							1	0
iter	1−2		3−4		5−6		7−8	

NRD Step 1

P1 → P3

P2 → P4

(c)

A()	P1 W	P1 R	P2 W	P2 R	P3 W	P3 R	P4 W	P4 R
1					1	1		
2					0	1	1	0
3								
4							0	1
5							1	0
iter					5−6		7−8	

Success

RD Step 1

RD Step 2

Success

(d)

A()	P1 W	P1 R	P2 W	P2 R	P3 W	P3 R	P4 W	P4 R
1	0	1	1	0				
2	1	0	0	1			1	0
3								
4					0	1		
5					1	0	1	0
iter	5		6		7		8	

P1 → P2

(e)

A()	P1 W	P1 R	P2 W	P2 R	P3 W	P3 R	P4 W	P4 R
1	1	0						
2	0	1			1	0		
3								
4			0	1				
5			1	0	1	0		
iter	6		7		8			

Fig. 4. R-LRPD example.

is the potentially large cost of more remote misses. In the experimental results in Chapter VII, the redistribution strategy is used until the number of restarts exceeds the number of processors. Also, the redistribution method attempts to obtain the most load balanced block scheduling based on the measured execution times of the iterations from the previous stage.

Fig. 4(d) shows the state of the shadow arrays after the second LRPD test when the remainder of the work is redistributed on all processors. Note, that work redistribution resulted in newly uncovered dependences. Fig. 4(e) shows the final state of the shadow arrays after the second (and successful) LRPD test with work redistribution (RD).

C. The Sliding Window (SW) Strategy

The performance of the R-LRPD test is very dependent on the distribution and type of data dependences. For codes with long distance data dependences, there is the sliding window strategy. Instead of distributing the *entire* iteration space over all the available processors, the entire speculative execution process can be strip-mined and the R-LRPD test is applied on each strip of contiguous iterations.

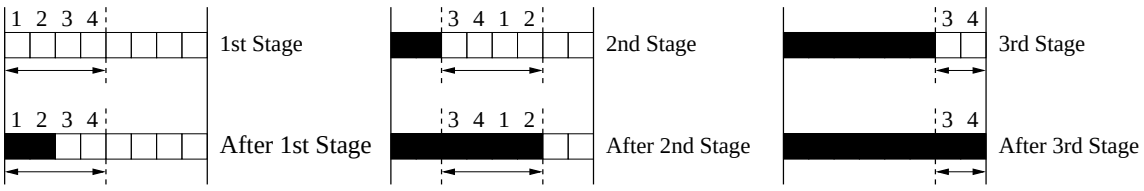


Fig. 5. Sliding window example.

In Fig. 5 we illustrate a few execution stages when this technique is applied to a loop. Assuming a total of 4 processors we schedule the first 4 contiguous iterations (1–4) and speculatively execute them in parallel. The subsequent analysis phase will commit iterations 1 and 2 and re-schedules iterations 3 and 4 because of a dependence between processor two and three. The *commit point* is advanced to iteration 3 and higher iterations (5 and 6) are scheduled. A new speculative R-LRPD test is performed and all 4 iterations can be committed (3-6) because no dependences have been uncovered. Finally the last two iterations are speculatively executed on processors 3 and 4.

To increase memory reference locality we organize the sliding window in a circular manner such that iterations are re-executed (if necessary) on their originally assigned processor. There are trade-offs to be made between the Sliding Window

(**SW**) and the previously presented strategies. For a fully parallel loop (N)RD methods execute all iterations in one stage, i.e., with one global synchronization, while SW will have one synchronization per strip. If dependences are present, it is possible that (N)RD techniques need to re-execute many more iterations than SW. The SW strategy has potentially more analysis overhead because it may have to go over the shadows of the memory elements that are reused in every iteration. So far we have not devised a strategy to choose between the two techniques except through the use of history based predictions.

The window size, i.e., the size of the block of contiguous iterations (super-iteration) assigned to one processor affects the number of global synchronizations (a larger window needs fewer global synchronizations) and the number of uncovered dependences. So far we have been able to tune our technique only experimentally (empirically). The scheduled block sizes can be dynamically adjusted by applying history based prediction. When many close dependences are encountered, the block size is increased. Alternatively, we can start with a very large block, equivalent to (N)RD and, if dependences are uncovered, reduce it until no re-executions are needed.

D. Extracting Data Dependence Graphs

Although the R-LRPD test can extract parallelism from loops for which a proper inspector does not exist, it cannot always extract the maximum available parallelism with the NRD, RD, or SW strategies. For some loops with complex dependence graphs but significant intrinsic parallelism, the R-LRPD test may generate an almost sequential execution schedule. In such cases, it would be beneficial to extract the (iteration) data dependence graph (DDG) and generate an optimized sched-

ule. Somewhat similar techniques have been previously presented in the literature [8, 10, 15, 19, 20], but apply only to loops from which a proper inspector can be extracted.

The Sliding Window R-LRPD test can be used to detect, window by window, the edges of the DDG and store them as (Read, Write) pairs. The shadow arrays are organized as n-level mark list where n is the number of iterations assigned to each processor. A distributed last reference table maintains the last valid write for each memory address. This is used to detect cross-window flow dependences between a successfully completed iteration and an iteration inside the current window. After each stage (a doall), we perform the SW R-LRPD test. Every cross-processor flow dependence between multiply referenced memory elements is logged into the inverted edge table. We also log each intra-processor flow dependence and each cross-window flow dependence into the inverted edge table. We save the last valid write reference of each memory address into the distributed last reference table in order to record the latest write reference occurring before the current window of iterations. At the next stage, we record the next set of references into the shadow array. After execution is completed, we obtain the DDG by inverting the edges in the graph stored in the inverted edge table.

Fig. 6 illustrates an application of this method for two processors and one iteration assigned to each processor. `InitWindow` and `AdvanceWindow` control the iteration window. `Analysis` applies the LRPD test for iterations within the window and generates the DD graph edges. In Step 1, after iterations 1 and 2 are executed in parallel, one cross-processor flow dependence is found and recorded in the inverted edge table. In Step 2, no cross-processor flow dependences between processors are found. In Step 3, two *cross-window* flow dependences are detected and recorded in

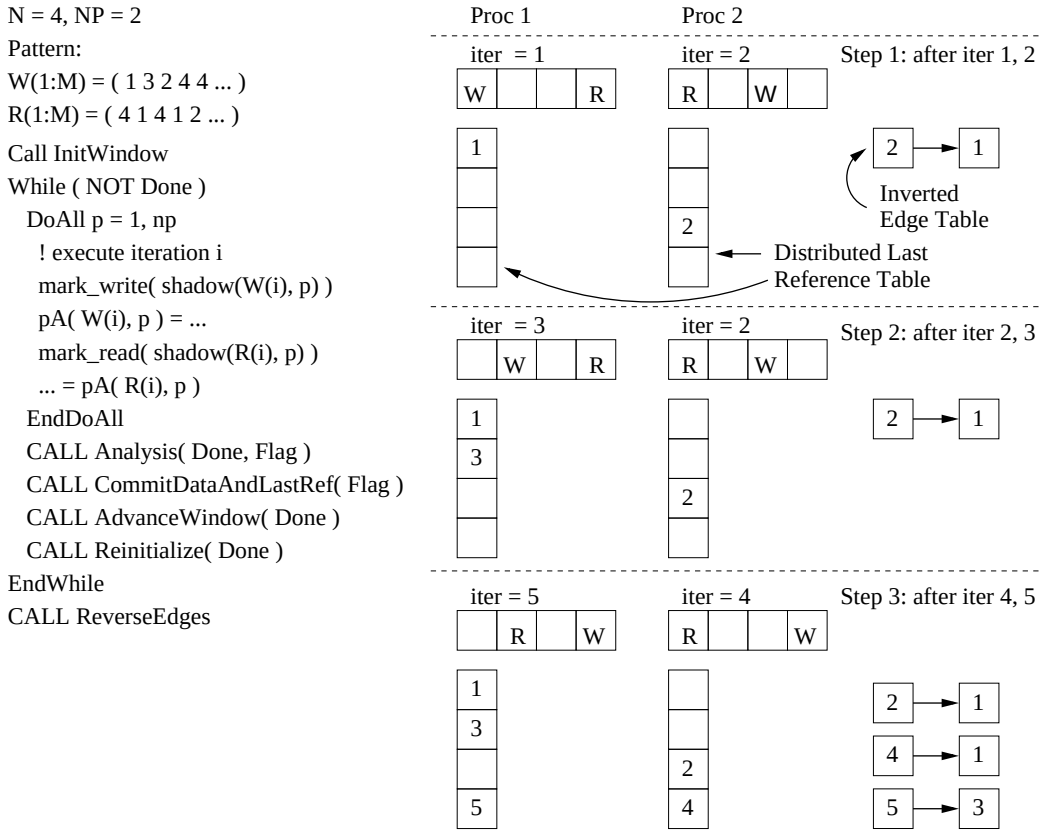


Fig. 6. Example for DD graph extraction using the sliding window R-LRPD test.

the inverted edge table.

The so extracted DDG is then reduced by removing edges representing non-essential dependences (through privatization, reduction recognition, etc). We can also insert synchronizations in the DDG followed by copy-out or accumulate and copy-out operations. Finally, the iteration space is scheduled using the DDG information. These last optimizations have not yet been implemented. Instead we have used the 'un-processed' DDG to obtain a wavefront schedule (sets of independent iterations separated by global synchronizations) and used them to parallelize some important loops in SPICE 2G6 (Section VII). The wavefront schedule of the DDG

from the example in Fig. 6 is shown in Fig. 7.

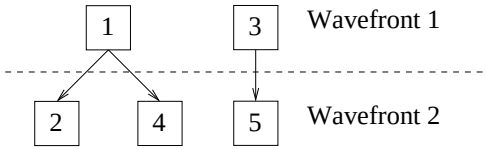


Fig. 7. Wavefront scheduling of the DDG from Fig. 6.

The complexity of the DDG collection is essentially the same as that of the SW R-LRPD test with some additional, constant overhead. It is important to note that in the case of sparse reference patterns (e.g. in SPICE) we have to use shadow hash-tables instead of shadow arrays. The result is an increased time per logging operation but a much more compact representation which allows faster analysis.

Tuning the algorithm means finding the right number of iterations per scheduled block (during the R-LRPD test). A larger block size results in fewer steps but possibly poorer parallelism (less possible overlap). If coarser granularity is desired, then the mark list size must be increased. We believe that, due to the input sensitivity of this method, more experiments are necessary before we can produce a good performance model.

CHAPTER V

MODELING THE RD AND NRD STRATEGIES

In [7], if the LRPD test passes (fully parallel loop), then the obtained speedups range from nearly 100% to *at least* 25% of the ideal. The overhead spent performing the single stage (original) LRPD test scales well with the number of processors and data set size of the parallelized loop. We can break down the time spent testing and running a loop with the LRPD (single stage) test in the following *fully parallel* phases:

The *initialization of shadow structures* is proportional to their dimension. For dense access patterns the shadow arrays are conformable to the tested arrays.

The work associated with *checkpointing* the state of the program before entering speculation is proportional to the number of distinct shared data structures that may be modified by the loop. For dense access patterns it is proportional to the dimension of all shared arrays that may be modified. The time spent saving the state of the loop at every stage depends on the chosen checkpointing implementation: as a step before loop execution or 'on-the-fly', before the modification of a shared variable.

The overhead associated with the execution of the *speculative loop* is proportional to the cost of marking relevant data references. For dense access patterns it can be approximated by the number of distinct references under test.

The final *analysis of the marked shadow structures* will be, in the worst case, proportional to the number of distinct memory references marked on each processor and to the (logarithm of the) number of processors that have participated in the speculative parallel execution. For dense access patterns this phase may involve the merge operation of p (number of processors) shadow arrays.

The recursive application of the LRPD test adds some additional overhead components which depend on the fraction of the successfully completed work which in turn depends on the data dependence structure of the loop. If cross-processor dependences are detected then a *Data Restoration* phase will restore the state of the shared arrays that were modified by the processors whose work cannot be committed. Its time is proportional to the number of elements of the shared arrays that need to be copied from their checkpointed values. If dependences are detected and re-execution is needed, then the shadow arrays will be *re-initialized*. The *Commit* phase transfers the last data computed (last value) by the earlier processors from private to shared memory. Its cost is proportional to the number of written array elements. Each of these steps is *fully parallel* and scales with the number of processors and data size. Furthermore, the commit, re-initialization of shadow arrays and restoration of modified arrays can be done concurrently as two tasks on the two disjoint groups of processors, i.e., those that performed a successful computation and those that have to restart.

The number of times re-execution is performed, as well as the work performed during each of them, depends on the strategy adopted: with or without work redistribution. When we do not redistribute work (NRD), the time complexity equals the cost of a sequential execution (worst case). We will have at most p steps performing n/p work, where p is the number of processors and n is the number of iterations. In the redistribution case (RD), each step will take progressively less time because we execute in p processors a decreasing amount of work. Completion is guaranteed in a finite number of steps because the first processor always executes correctly. Let us now model more carefully the tradeoff between these two strategies.

Initially, there are n iterations equally distributed among the processors. The

computation time for each iteration is ω, yielding a total amount of (useful) work in the loop as ωn. In the following discussion we assume that we know ω, the cost of useful computation in an iteration, ℓ, the cost of redistributing the data for one iteration to another processor, and s, the cost of a barrier synchronization.

For the purpose of an efficient speculative parallelization we classify loop types based on their dependence distribution in the following two classes: (a) **geometric (α)** loops where a constant fraction $(1-\alpha)$ of the current *remaining* iterations are completed during each speculative parallelization (step), and (b) **Linear (β)** loops where a constant fraction $(1-\beta)$ of the *original* iterations are completed during each speculative parallelization (step).

A. No Data Redistribution (NRD)

If $\omega \leq \ell + s$, then it does not pay to redistribute the remaining iterations among the p processors after a dependence is detected during a speculative parallelization attempt. That is, the overhead of the redistribution (per iteration) is larger than work of the iteration. In this case, the total time required by the parallel execution is simply

$$T_{\text{static}}(n) = \sum_{i=0}^{k_s} \left(\frac{n\omega}{p} + s \right) = \frac{n\omega k_s}{p} + k_s s \tag{5.1}$$

where $k_s \leq p$ is the number of steps required to complete the speculative parallelization. Thus, to determine the time $T_{\text{static}}(n)$ we need to compute the number of steps k_s (the number of speculative parallelization attempts needed to execute the loop). We consider two cases (the α and β loops) and determine the value of k_s for each.

For the α loops, we assume a constant fraction $(1-\alpha)$ of the *remaining work* is completed during each speculative parallelization step. In this case, $n\omega\alpha^i$ work

remains to be completed after i steps. Thus, the final (k_s-th) step will occur when $n\omega\alpha^{k_s} = \frac{n\omega}{p}$ (since then all remaining iterations reside on one processor because we do not redistribute). So, solving for k_s, we get $k_s = \log_{\frac{1}{\alpha}} p$. For example, if $\alpha = \frac{1}{c}$, then $k_s = \log_c p$, for constant c.

For the β loops, we assume a constant fraction $(1 - \beta)$ of the *original work* is completed successfully in each speculative parallelization step (i.e., a constant number of processors successfully complete their assigned iterations). In this case, $n\omega(1 - \beta)i$ work is completed after i steps. Thus, all the work will be completed when $n\omega(1 - \beta)k_s = n\omega$, or when $k_s = \frac{1}{(1-\beta)}$. For example, for a fully parallel loop, $\beta = 0$ and so $k_s = 1$ and $T_{\text{static}}(n) = \frac{n\omega}{p} + s$, and for a sequential loop, $\beta = \frac{p-1}{p}$ and so $k_s = p$ and $T_{\text{static}}(n) = n\omega + ps$.

B. Data Redistribution (RD)

If $\omega > \ell + s$, then it may pay to redistribute the remaining iterations among the p processors after a dependence is detected during a speculative parallelization attempt. In this case, as opposed to the **NRD** case, in each subsequent step the processors will have a smaller number of iterations assigned to them. Thus the total time required by the parallel execution is

$$T_{\text{dyn}}(n) = \sum_{i=0}^{k_d} \left(\frac{n_i\omega}{p} + \frac{n_i\ell}{p} + s \right) \tag{5.2}$$

$$= \frac{(\omega + \ell)}{p} \left(\sum_{i=0}^{k_d} n_i \right) + k_d s \tag{5.3}$$

where n_i is the number of iterations remaining to be completed at the start of the i-th step, and k_d is the number of steps completed to this point using redistribution.

Even if redistribution is initially useful, there comes a point when it should

be discontinued. In particular, it should occur only as long as the time spent (per processor) on useful computation is larger than the overhead of redistribution and synchronization. That is, redistribution should occur as long as the first term in the first sum in Eq. 5.2 is larger than the sum of the last two terms, i.e., as long as

$$n_{k_d} \geq \frac{ps}{\omega - \ell}. \tag{5.4}$$

Note that this condition can be tested at run-time since it only involves the number of uncompleted iterations which is known at run-time and p, s, ω, and ℓ, which we assume are known *a priori*, and can be estimated through both static analysis and experimental measurements.

In summary, for the first k_d steps, the remaining iterations should be redistributed among the processors. After that, no redistribution should occur. From this point on, we are in the case described as T_{static} above, but starting from $n' = n_{k_d}$ instead of n. Thus, the total time required will be

$$T(n) = T_{\text{dyn}}(n) + T_{\text{static}}(n_{k_d}) \tag{5.5}$$

$$= \frac{(\omega+\ell)}{p}\left(\sum_{i=0}^{k_d} n_i\right) + \frac{n_{k_d}\omega k_s}{p} + (k_d + k_s)s \tag{5.6}$$

where n_i, k_d and k_s are as defined above.

To compute an actual value for $T(n)$, we need to determine n_i, k_d, and k_s, and substitute them in Eq. 5.6. For example, consider the geometric loops in which a constant fraction $(1-\alpha)$ of the current work is completed during each speculative parallelization attempt.[1] In this case, $n_i = n\alpha^i$, and $\sum_{i=0}^{k_d} n_i = \sum_{i=0}^{k_d} n\alpha^i = n\left(\frac{\alpha^{k_d+1}-1}{1-\alpha}\right)$.

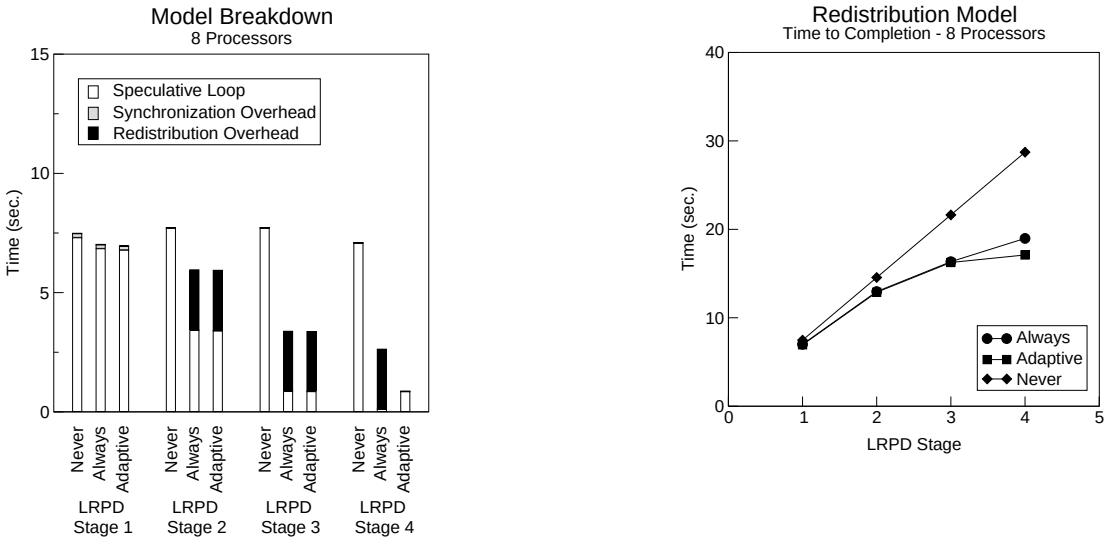
[1]The case in which a constant fraction of the original work is completed during each speculative parallelization is not realistic here since the number of iterations each processor is assigned varies from one speculative parallelization to another.

Using $n_{k_d} = n\alpha^{k_d}$ in Eq. 5.4, and solving for k_d we obtain $k_d = \log_\alpha \left[\left(\frac{s}{\omega-\ell}\right)\frac{p}{n}\right]$. Finally, $k_s = \log_{\frac{1}{\alpha}} p$ as described above. Thus, the total time required will be

$$T(n) = \frac{n}{p}(\omega+\ell)\left(\frac{\alpha^{k_d+1}-1}{1-\alpha}\right) + \frac{n\alpha^{k_d}\omega k_s}{p} + (k_d+k_s)s$$

where k_d and k_s are computed as defined above based on the known values of n, ω, ℓ, s, and α. In general, one may not know α exactly, however, in many cases reasonable estimates can be made in advance, and recomputed during execution (e.g., as an average of the α values observed so far).

C. Experimental Model Validation



(a) (b)

Fig. 8. Selection strategy between RD and NRD re-execution technique.

The graph in Fig. 8 illustrates the loop, testing overhead, and redistribution overhead time (mostly due to remote cache misses) for each restart of the R-LRPD test of a synthetic loop executed on 8 processors of an HP-V2200 system. We assume

that the fraction of remaining iterations is 1/2. The initial speculative run is assumed not to incur a redistribution overhead. We have performed three experiments to illustrate the performance of the following three strategies: The *never* case means that we use the NRD strategy (never redistribute the remaining work). *Adaptive* redistribution means that redistribution is done as long as the previous speculative loop time is greater than the sum of the overhead and incurred delay times of the previous run. *Always* redistribution means 'always' redistribute. Fig. 8(a) shows the execution time breakdown of our experiment. At each stage of the R-LRPD test we measure the time spent in the actual loop and the synchronization and redistribution overhead. In Fig. 8(b) we show the cumulative times spent by the test during its four stages. The "adaptive" redistribution method begins to have shorter overall execution times compared to the "always" redistribution method after the failure on processor 8. The NRD method performs the worst, by a wide margin. It should be noted that our synthetic loop assumes, for simplicity, that α and β are constant. In practice we would have to adjust the model parameters at every stage of the R-LRPD test.

CHAPTER VI

IMPLEMENTATION AND OPTIMIZATIONS

We have implemented the Recursive LRPD test in the NRD, RD, and SW flavors. We have also applied several optimization techniques to reduce the run-time overhead of checkpointing and the load imbalance caused by the required block scheduling of the parallelized irregular loops. As previously mentioned block scheduling is a requirement of the R-LRPD test and thus load balancing is an important issue. The base implementation (code transformations) is mostly done by our run-time pass in Polaris (it can automatically apply the simple LRPD test) and additional manually inserted code for the commit phase and execution of the `while` loop shown in Fig. 4(b).

A. On-demand Checkpointing and Commit

In Section V we have already mentioned the need to optimize checkpointing because its work is approximately proportional to the working set of the loop. At every stage of the test we find a contiguous number of processors (processors executing a contiguous block of iterations) that have executed without uncovering any dependences between them and a remainder block of processors which have to re-execute their work. The data residing in the shared arrays needs to be saved before it is modified by the speculative execution. There are two types of shared variables: Variables that are under test because the compiler cannot analyze them and variables proven by the compiler to be either independent (accessed in only one iteration (processor) or read-only) or privatizable. Saving state or preserving a safe state can be done in two ways:

- Write into un-committed private storage which we later either commit by copying it out to the shared area or delete.

- Copy the data that *may* be modified by the speculative loop to another, safe, memory storage and then either delete it (if we commit the results of the speculation) or copy back from the original variables (in case we have to restore state).

Both the copy-in/copy-out mechanism and the copying to a safe area can be done in two ways: before the speculative loop the *entire* working set of the loop is saved or copied-in, or *On-demand*, during loop execution. Performing this activity before the loop always adds to the critical path-length of the program and, in the case of sparse reference patterns, generates more work and consumes more memory than necessary. It is, however, fully parallel and the per operation cost is small (block copy). The on-demand strategy has many advantages: It performs the copy operations only when and if they are needed, which, for sparse codes, can be orders of magnitude less than a 'wholesale' approach. Moreover, because it is done during loop execution, it may not actually add to the critical path of the program due to the exploitation of low level parallelism. However, each operation has to be initiated separately and may have to be guarded. We need to save data (or copy-in) only at the first write (or read) reference. To accomplish this 'first access' filter we have to distinguish between *variables under test*, i.e., those variables that cannot be analyzed by the compiler and which are shadowed during execution and shared variables that have been analyzed statically. From these variables only the independent ones need attention (read-only and privatized variables don't modify state and don't need to be restored). An independent variable references its location in only one iteration

(or processor) and its location can be extracted by the compiler. The 'referenced first' filter can be generated also by the compiler either through peeling it off (in case of nested loops) or using a guard and a very simple shadow (or tag). If the code is such that there is only one statement per distinct reference in an iteration then the filter becomes trivial.

The commit and restoration phase needed after the analysis region of each stage of the R-LRPD test depends on the strategy used for checkpointing. For Committing data we need to copy out the last value written (in the sequential semantics). For independent arrays (not under test) this is either accomplished by a compiler generated loop (in case we used copy-in) or by simply deleting the corresponding saved data (if the wholesale copy before the loop strategy is used).

In the experiments shown in Section VII we have implemented on demand copy-in, last-value-out for the arrays under test and on-demand checkpointing with release of back-up storage at commit phase because it proved to be the most cost-effective for the application studied.

B. Data Dependence Analysis

Since the LRPD test does not have any scheduling restrictions, the LRPD test allows for processors p_1 and p_2 to execute iterations i_1 and i_2 respectively where $i_1 > i_2$ and $p_1 < p_2$. This requires a comparison of the shadowed references in the shadow arrays for all processors to determine the existence of a flow dependence as shown in Fig. 9 (a). The horizontal arrows represent the cross-processor memory accesses for the shadow arrays. For this horizontal analysis, the examination of the rows of each shadow array are distributed among the processors. Since each process needs to access the shadow arrays of the other processors, there could be many remote cache

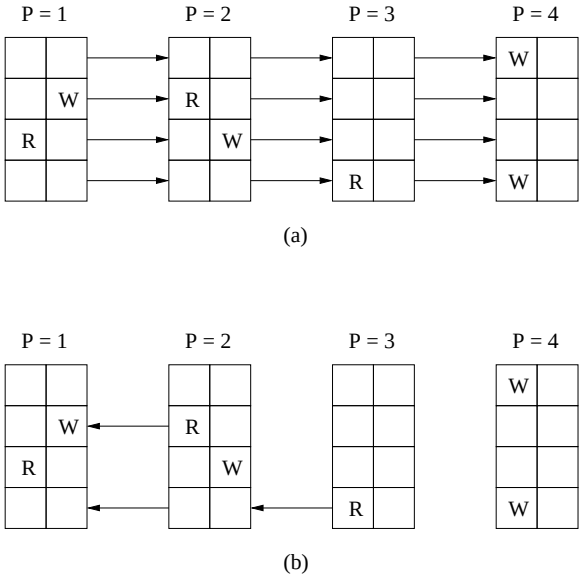


Fig. 9. Dependence analysis with different traversal methods.

misses with this analysis method.

Since the R-LRPD test uses static blocked scheduling, for any read first mark on processor p_i, a flow dependence can only occur if there is a write first mark on any processor 1 to p_{i-1}. During the dependence analysis, each processor, upon finding a read first mark for an element in its shadow array, needs only to search for a write first mark in the corresponding array index in the shadow arrays of processors that are logically to its left as shown in Fig. 9 (b). With this vertical analysis method, processors examine the shadow arrays of other processors only when necessary so the remote cache misses are minimized.

C. Shadow Data Structures

1. Sparse Shadow Structures

If the access pattern is dense, then the shadows of the arrays under test are implemented as shadow arrays because they are by far the most cost effective solution. However, in the case of sparse applications, e.g., SPICE, the use of shadow arrays is not viable. Because SPICE effectively does its own memory management using a very large static array, the use of a shadow array would imply that the analysis phase would have to traverse the entire work space of the program. Moreover the shadows would use up a very large amount of memory. Our solution to this problem has been the use of shadow hash tables. We have implemented private (per processor) conformable hash tables using arrays and a hash function that hashes the array index with the modulus operator. Each row of the hash table is implemented as a singly-linked list. The first element in the linked list is stored in the test section of the array. Any additional elements in the linked list of the hash table row are stored in the overflow section. Fig. 10 illustrates the implementation of the hash table using arrays. Thus we can compact the entire access pattern and keep the algorithm scalable with data size and number of processors. The cost of the scalability is that every shadowed data access will be more expensive. This hash table implementation, has also been used in [21, 22] for reduction optimizations.

2. Parallel Shadow Processing

For array-based shadow structures, the array elements are implemented as one-byte integers. Using padding to ensure that the array size is a multiple of eight, these one-byte shadow arrays in the R-LRPD library subroutines as formal arguments

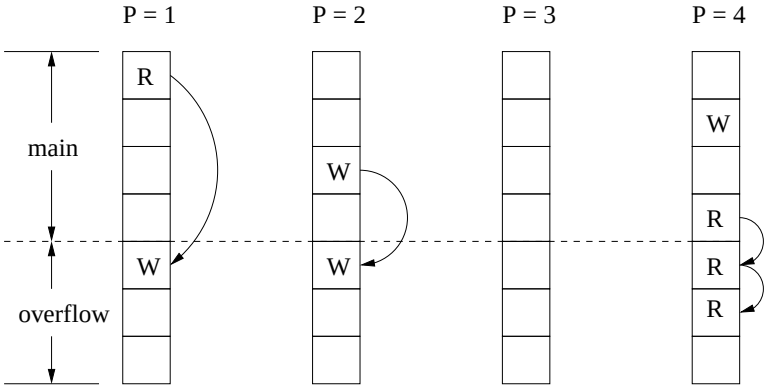


Fig. 10. Shadow hash table.

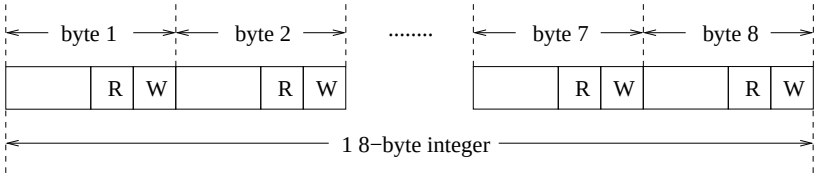


Fig. 11. Eight-byte shadow structure.

and can be redefined as arrays of eight-byte integers. An example of eight one-byte shadow elements redefined as a single eight-byte integer is shown Fig. 11. This allows the library to process eight one-byte shadow structure elements simultaneously. This method is used for initialization of the shadow structures and during data dependence analysis. The vertical analysis can test simultaneously eight shadow elements using eight-byte bitwise AND operations and bit-masks.

D. Feedback-Guided Load Balancing

One of the drawbacks of the R-LRPD test is the requirement that the speculative loop needs to be statically block scheduled in order to commit partial work. Since our techniques are intended for irregular codes, load balancing does indeed pose some performance problems. We have independently developed and implemented a new

technique similar to [23] that adapts the size of the blocks of iterations assigned to a processor such that load imbalance is minimal at every stage of the R-LRPD test.

At every instantiation of the loop, we measure the execution time of each iteration. These execution times are saved for the next instantiation of the loop. At the next instantiation of the loop, we compute the prefix sums of the total execution time of the previous instantiation of the loop as well as the 'ideal', perfectly balanced, execution time per processor, i.e., the average execution time per processor ($\frac{total\ execution\ time}{number\ of\ processors}$). Using the prefix sums we can then compute a block distribution of iterations that would have achieved perfect load balance. We use this result as a first order predictor for the current instantiation of the loop. When the iteration space changes from one instantiation to another, we scale the block distribution accordingly. The implementation is rather simple: We instrument the loop with low overhead timers and then use a parallel prefix routine to compute the iteration assignments to the processors. A possible improvement for this technique is to use higher order derivatives to better predict trends in the distribution of the execution time of the iterations. The overhead of the technique is relatively small and can be further decreased. Another advantage of the method is its tendency to preserve locality.

CHAPTER VII

EXPERIMENTAL RESULTS

Our experimental test-bed is a 16 processor ccUMA HP-V2200 system running HPUX11. It has 4GB of main memory and 4MB single level caches. We have applied our techniques to the most important loops in TRACK, a PERFECT code, SPICE 2G6, a PERFECT and SPEC code, and FMA3D, a SPEC 2000 code. The codes (with the exception of Loop 15 in SPICE_DCDCMP have been instrumented with our run-time pass which was (and is) developed in the Polaris infra-structure [24].

To better gauge the obtained speedups, we define a measure of the parallelism available in a loop over the life of the program as the *parallelism ratio* ($\frac{total\ instantiations}{total\ restarts\ +\ total\ instantiations}$). For example, a fully parallel loop has a $PR = 1$ and a partially parallel loop has a $PR < 1$. In the case of the NRD strategy, a fully sequential loop has a $PR = 1/p$, while the RD strategy can have a much lower PR.

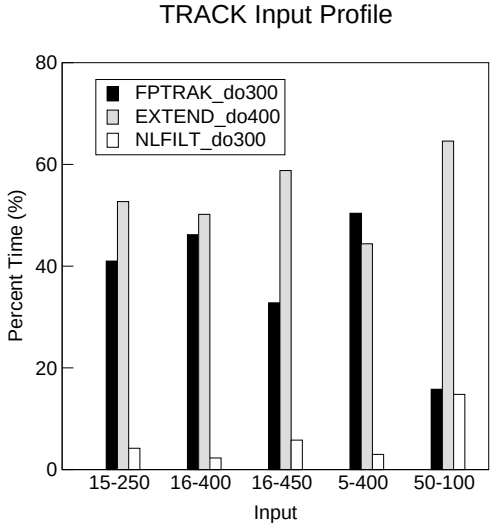
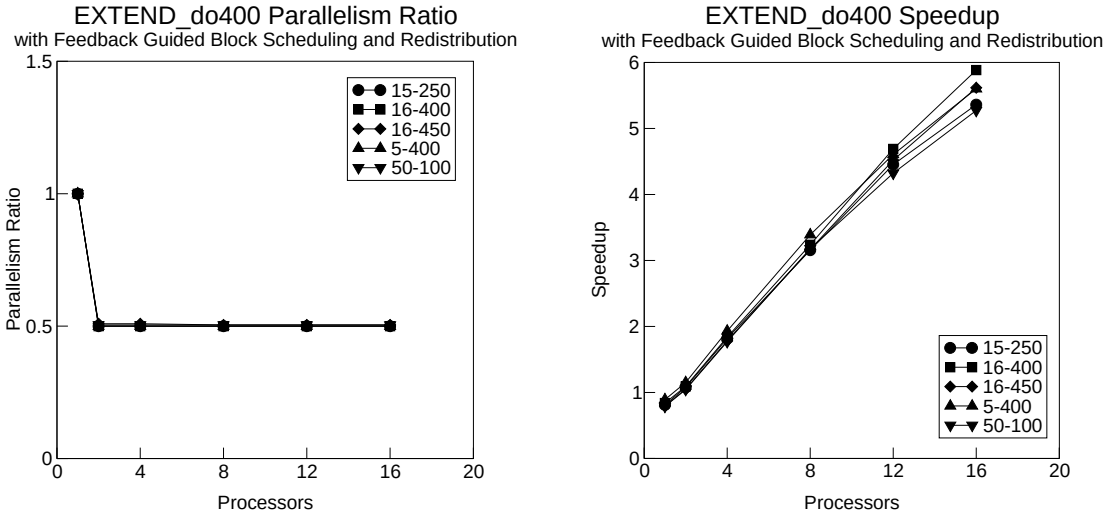


Fig. 12. TRACK execution profile for entire program.

TRACK is a missile tracking code that simulates the capability of tracking many boosters from several sites simultaneously. Its main loops, 400 in subroutine EXTEND, 300 in NLFILT and 300 in FPTRAK, account for $\approx 95\%$ of sequential execution time. We have modified the original inputs which were too small for any meaningful measurement. The execution profile of the **entire TRACK code** for different input sets given in Fig. 12 shows how input sensitive this program is. We have also created several input files to vary the degree of parallelism of some of its loops.



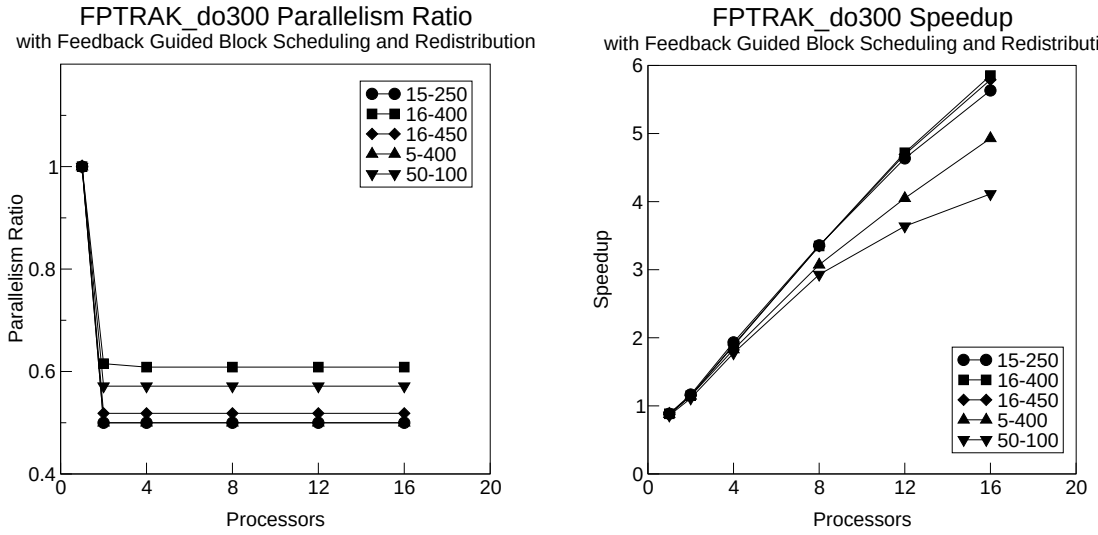
(a) (b)

Fig. 13. EXTEND 400: (a) Parallelism ratio and (b) speedup.

EXTEND 400. This loop reads data from a read-only section of an array and always writes at the end of the same arrays that are being extended at every iteration. It first extends them in a temporary manner by one slot. If some loop variant condition does not materialize then the newly created slot (track) is re-used (overwritten) in the next iteration. This implies that at most one element of the track

arrays needs to be privatized. These arrays are indexed by a counter (LSTTRK) that is incremented conditionally and whose values cannot be precomputed.

We have all processors speculatively compute LSTTRK from an offset based on its pre-loop value and collect the array reference ranges [21]. After the first parallel execution we obtain the per processor offsets of the induction variable (the prefix sums of LSTTRK) and show that all read references to the array do not intersect any writes, i.e., maximum read index $<$ minimum write. In the second `doall` we repeat the execution using the correct offsets for LSTTRK. Last value assignment commits the arrays to their shared storage. Figs. 13(a) and (b) show the **PR** and the best obtained speedup for these inputs, which represents about 60% of the speedup obtainable through hand-parallelization.

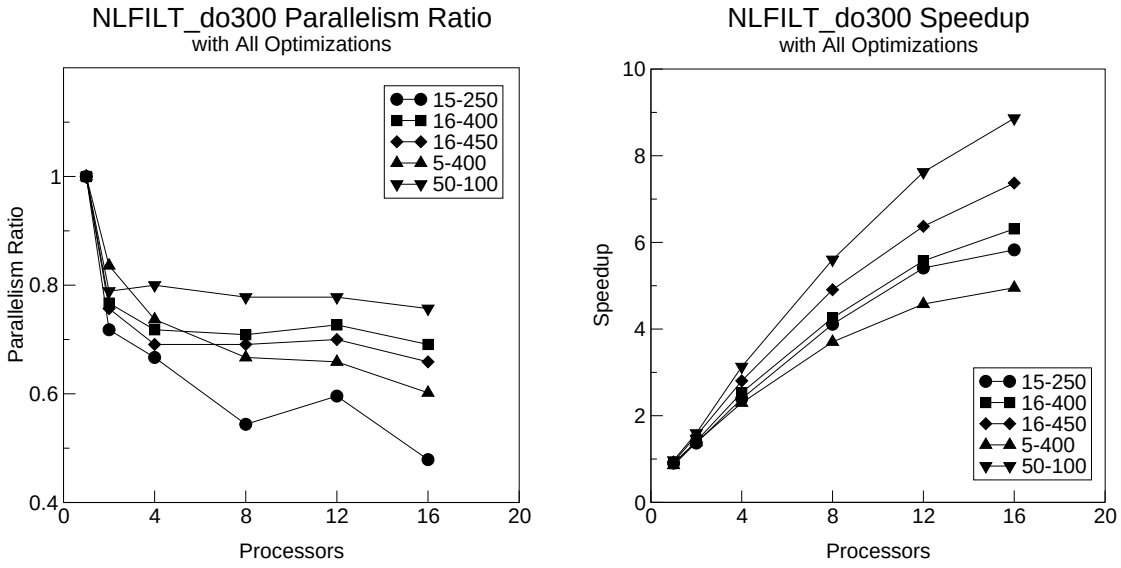


(a) (b)

Fig. 14. FPTRAK 300: (a) Parallelism ratio and (b) speedup.

FPTRAK 300. This loop is very similar to, yet simpler than, EXTEND 400. The array under test has a read-only front section which is conditionally extended by

appending a new element. The array under test is privatized with the copy-in/last-value out method and shadowed. The same two stage approach as in EXTEND is employed here. Figs. 14(a) and (b) show the PR and the best obtained speedup for these inputs.

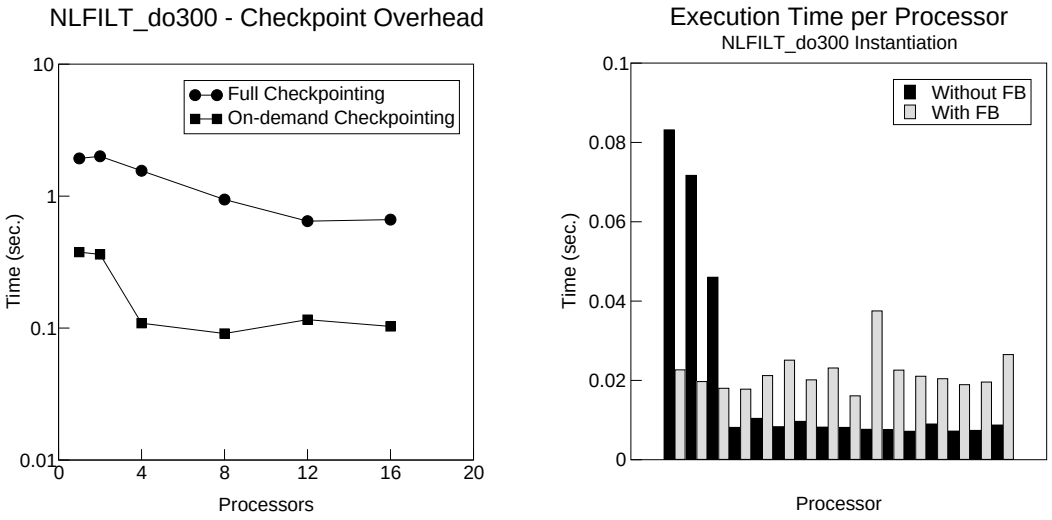


(a) (b)

Fig. 15. NLFILT 300: (a) Parallelism ratio and (b) speedup.

NLFILT 300. The PF parameter in the input file has been set to 3 to make the loop DO 300 in NLFILT 75% parallel (75% of the total of 56 instantiations are parallel, while the rest have to restart once per instantiation). The chosen value is realistic. The input file in the Perfect suite has PF = 6 which makes the loop always parallel (there are a few dependences that are masked by the processor-wise LRPD test). The compiler un-analyzable array that can cause dependences (mostly short distances) is NUSED. Its write reference is guarded by a loop variant condition so that a proper inspector cannot be obtained. Fig. 15(a) presents the effect of

the input sets on **PR** for different number of processors. **PR** is dependent on the number of processors because only interprocessor dependences affect the number of restarts (stages) of the R-LRPD test. With feedback guided scheduling, the length of iteration blocks assigned to processors is variable which can lead to a variable **PR**. Fig. 15(b) shows the best obtained speedups (all optimizations turned on) for the tested input sets. The speedup numbers include all associated overhead.



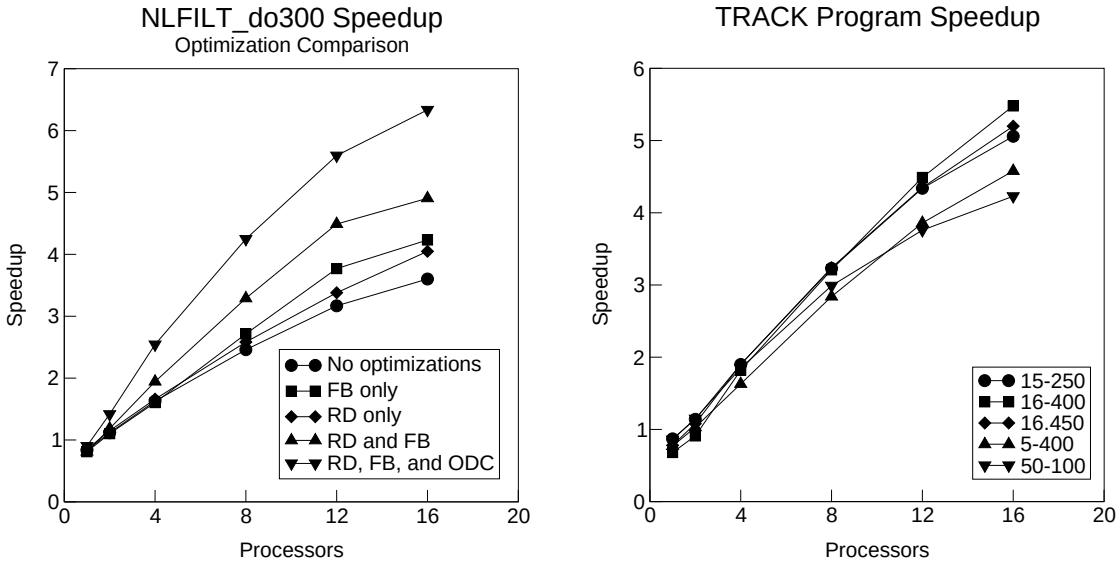
(a) (b)

Fig. 16. NLFILT 300: Feedback guided load balancing.

The next figures, present the importance of our optimizations to the quality of our parallelization. Fig. 16(a) illustrates the reduction in overhead when the checkpointing is done before the speculative loop and on-demand, i.e., during the speculative loop. It is quite obvious that the on-demand strategy generates much less overhead and drastically reduces the overall execution time. Fig. 16(b) compares the execution time per processor when the iteration space is equally distributed to the processors with the time per processor when feedback guided scheduling is employed.

We can clearly see that our loop balancing technique 'flattens' the distribution of execution time and thus balances the irregular loop.

Fig. 17(a) compares the effectiveness of the various optimization techniques. The input set is 16-400, i.e., a moderate number of dependences are uncovered almost independent of the number of processors used in the experiment. Clearly, due to the large state of the loop and its conditional modification the on-demand-checkpointing is the most important optimization. The load balancing technique is very important when redistribution (RD) is used. RD vs. NRD strategy has here a lesser impact because we use only 16 processors.

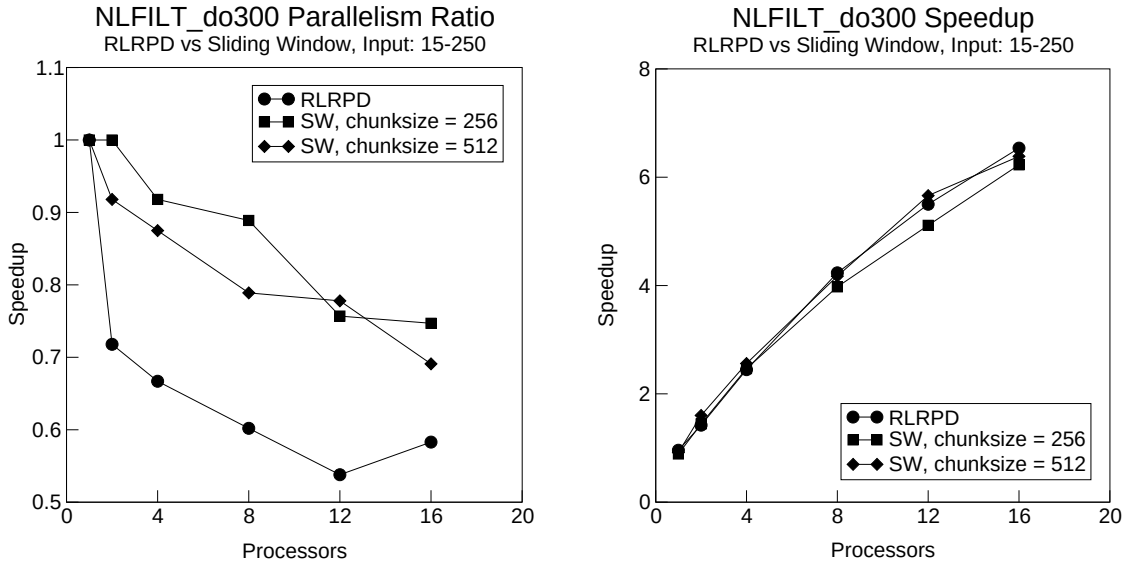


(a) (b)

Fig. 17. (a) NLFILT 300: Optimization contributions and (b) TRACK program speedup.

In Figs. 18 and 19, we show that both SW and (N)RD strategies are viable, depending on the actual dependence structure of the loop. The resulting **PR** is also strategy dependent. The effect on the obtained speedup is a bit skewed due to the

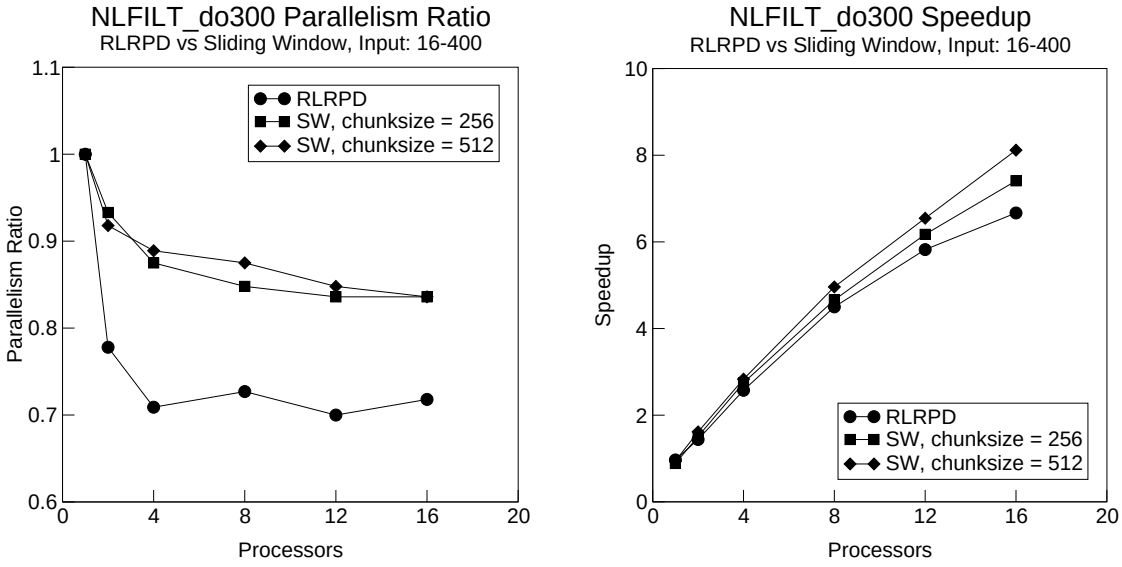
different levels of optimizations that could be applied. Scaling the analysis phase for the SW technique is not always possible. The graphs also show how the **PR** and speedup varies with the window size. Ideally, we want the largest window size for which there is a minimum number of failures (restarts); this size can be adapted based on previous loop instantiations. The overall program speedup of the **entire TRACK code** shown in Fig. 17(b) is scalable and is quite impressive.



(a) (b)

Fig. 18. NLFILT 300: Sliding window vs (N)RD strategy. Input: 15-250.

SPICE 2G6 is a circuit simulator that spends most of its time in two distinct loops: A loop in subroutine DCDCMP which implements a sparse solver (the decomposition part) and several similar loops in subroutine LOAD, BJT, MOSFET, etc., which update the Y matrix of a circuit with the current evaluation of the device models. None of the arrays can be compiler analyzed because they are all equivalenced to one large array (VALUE), the working space of the program and all references have multiple levels of indirection. It is a 'total' workspace aliasing problem.



(a) (b)

Fig. 19. NLFILT 300: Sliding window vs (N)RD strategy. Input: 16-400.

To make parallelization profitable we have chosen larger input decks than the ones available in the original PERFECT and SPEC codes, i.e., the circuit of a 128 bit adder in BJT technology and a scaled up input deck from PERFECT codes. We have parallelized the three most important loops in SPICE: Loops 70 and 15 in subroutine DCDCMP and the main loop in BJT (which is similar to all the loops called from the model evaluation routine LOAD). The technique to parallelize the main loop in BJT (speculative linked list traversal distribution, sparse LRPD test on the remainder couple with sparse reduction optimization) has been presented in [21, 22]. Loop 70 in DCDCMP is fully parallel with a premature exit and has been parallelized with our techniques described in [9, 25]. Loop 15 in DCDCMP (LU decomposition) is partially parallel due to the sparse nature of the circuit topology. We employ a sparse version of the R-LRPD test that can extract the Data Dependence Graph (DDG). Based on the DDG, we schedule in wavefronts. This schedule can be reused

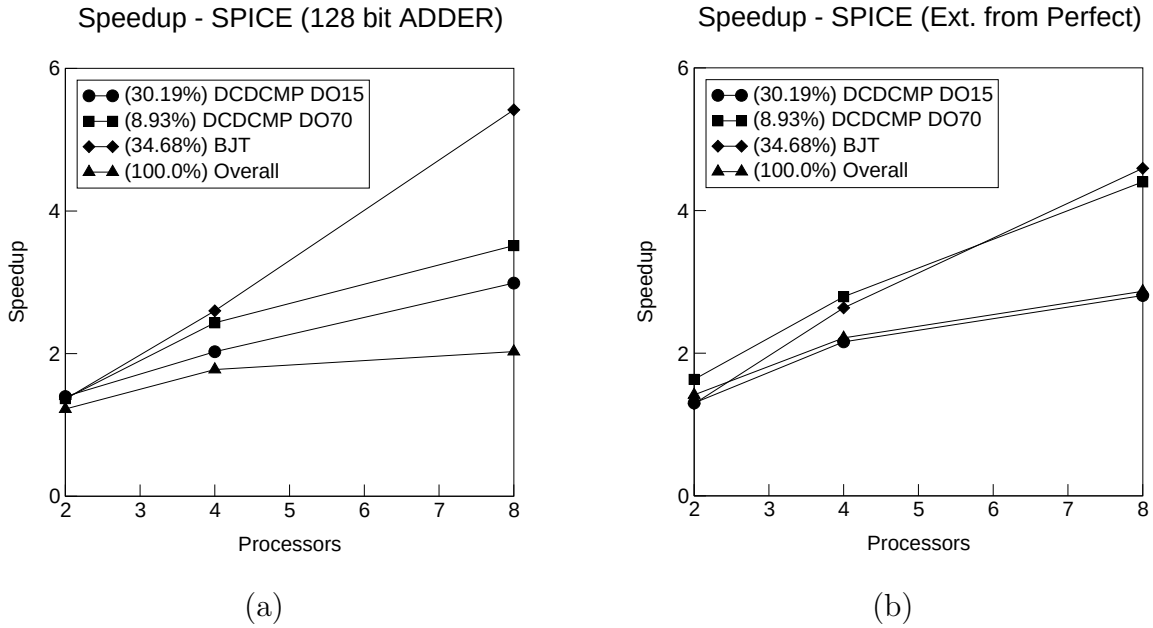


Fig. 20. SPICE 2G6 speedup for important loops and entire program for (a) adder128, and (b) extended PERFECT input decks.

throughout the execution of the program because the access pattern does not change, and thus fully amortizes the initial cost of generating the dependence graph.

For the `adder.128` input deck the parallelized loop in DCDCMP has 14337 iterations with a critical path length of 334 (number of wavefronts). We believe we will improve our speedup numbers by employing a better scheduling technique than the use of wavefronts. Other input decks we have studied have similar characteristics. Figs. 20 show the obtained speedups for each loop and the for the *entire code* for the studied input decks.

FMA3D is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. Its most important loop, (accounting for 56% of the sequential execution time), contains array references (to `stress` and `state` arrays) using indirection and its call graph is several levels deep.

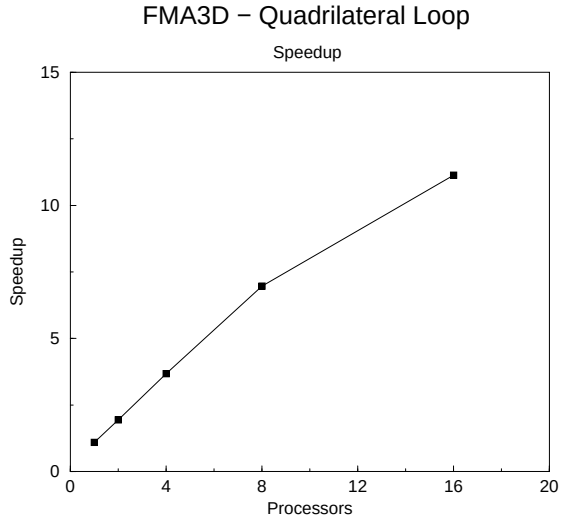


Fig. 21. FMA3D speedup of quadrilateral loop.

This complexity makes the 'Quadrilateral' loop statically un-analyzable (Theoretically this loop can be statically parallelized because it is input independent). As it turns out the loop is fully parallel and thus the R-LRPD test has only one stage. Fig. 21 shows the overall speedup of this loop.

CHAPTER VIII

CONCLUSION

A. Thesis Research

In this thesis, we have increased the applicability of speculative run-time parallelization by adapting the LRPD test for partially parallel loops. This method can exploit the performance of partially parallel loops which could not be parallelized previously with compiler analysis, inspector/executor methods, and/or the LRPD test. We've addressed limitations in the R-LRPD test by reducing the speculation overhead with optimizations such as on-demand checkpointing and reducing the load imbalance from blocked scheduling with feedback guided scheduling. The R-LRPD test limits the potential slowdown of speculation to the test overhead. Previously, the performance slowdown of the LRPD test for failed speculation is proportional to the sequential re-execution and the test overhead. We presented a speculative execution approach to build the DDG for a loop where inspector/executors could not (i.e. loops that have dependence cycles between address and data computation)

We presented different strategies capable of handling different dependence distributions with speculation: short dependences (R-LRPD test), long dependences (SW R-LRPD test), and complex dependence distributions (DDG extraction using the SW R-LRPD test). By applying the R-LRPD test to the entire loop, short dependences between two iterations can be hidden if run on the same processor. The SW R-LRPD strategy is suitable for long dependences since it is only affected by inter-processor dependences. Any cross-window dependences are handled by an on-demand copy-in/commit approach as in the R-LRPD test. For complex dependence distributions, we've shown that the SW R-LRPD test is capable of extracting the

DDG during speculative execution from loops where a proper inspector cannot be obtained. Then a schedule can be generated and possibly reused for future instantiations.

There is recent work in developing simpler run-time tests for parallelization by using comprehensive compile time techniques that combine control flow and data flow analysis to generate efficient run-time tests to decide whether a loop is parallel or not by evaluating a small set of run-time values. This approach could reduce the need for an exhaustive memory reference analysis. However, if these simpler run-time tests cannot prove a loop is parallel, we believe that the R-LRPD test will be needed for more exhaustive data dependence analysis.

B. Future Directions

This thesis presented different strategies in applying the R-LRPD test. However, the strategy selection was based on empirical experiments. We believe that heuristics could be developed for choosing between the different R-LRPD (NRD, RD, and SW) strategies based on the dependence distributions of prior loop instantiations. Also, heuristics could be developed for selecting, adapting and/or predicting the best possible window size for the SW R-LRPD test based on historical information at run-time.

For the feedback guided scheduling, more accurate predictors of the execution time distribution for future loop instantiations could reduce the load imbalance and possibly avoid the overhead in measuring the execution time of each iteration.

With the DDG generated by the R-LRPD test, there may be more efficient schedules that can avoid or reduce the global synchronizations inherent to using wavefront schedules. However, the scheduling heuristics will need to reduce syn-

chronizations while not incurring large additional overhead in the scheduling itself compared to wavefront scheduling.

We believe there will be cases where loops may have slowly changing dependence distributions. If reuse of an exact schedule is not possible, there is a chance that the difference in the dependence distribution may be slight so that the previous schedule could be adjusted rather than extracting a new data dependence graph and computing a new schedule.

REFERENCES

[1] D. A. Padua and M. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986.

[2] M. Wolfe, *Optimizing Compilers for Supercomputers*, The MIT Press, Cambridge, MA, 1989.

[3] L. Nagel, "Spice2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, University of California at Berkeley, Berkeley, California, May 1975.

[4] R. G. Whirley and B. Engelmann, *DYNA3D: A Nonlinear, Explicit, Three-Dimensional Finite Element Code for Solid and Structural Mechanics*, Lawrence Livermore National Laboratory, Nov. 1993.

[5] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, et. al, *Gaussian 98, Revision A.11*, Gaussian, Inc., Pittsburgh PA, 2001.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "Charmm: A program for macromolecular energy, minimization, and dynamics calculations," *Journal of Computational Chemistry*, vol. 4, no. 6, pp. 187, 1983.

[7] L. Rauchwerger and D. A.. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 160, 1999.

[8] L. Rauchwerger, N. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *Int. J. Paral. Prog.*, vol. 26, no. 6, pp. 537–576, Jul. 1995.

[9] L. Rauchwerger and D. A.. Padua, "Parallelizing WHILE Loops for Multiprocessor Systems," in *Proc. of 9th International Parallel Processing Symposium*, pp. 347–356, Apr. 1995.

[10] J. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Trans. Comput.*, vol. 40, no. 5, pp. 603–612, May 1991.

[11] M. Gupta and R. Nim, "Techniques for Speculative Run-Time Parallelization of Loops," in *Proc. of Supercomputing 1998*, pp. 1–12, 1998.

[12] S. Rus and L. Rauchwerger, "Hybrid analysis: static & dynamic memory reference analysis," *International Journal of Parallel Programming*, vol. 31, no. 3, pp. 251–283, 2003.

[13] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization in multiprocessors," *2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, pp. 13–24.

[14] P. Rundberg and P. Stenstrom, "Low-cost thread-level data dependence speculation on multiprocessors," *4th Workshop on Multithreaded Execution, Architecture and Compilation*, Dec. 2000.

[15] I.H. Kazi and D. Lilja, "Coarse-grained speculative execution in shared-memory multiprocessors," in *Proc. of the 12th ACM International Conference on Supercomputing*, Jul. 1998, pp. 93–100.

[16] P. Tu and D. A. Padua, "Automatic array privatization," *1993 Workshop on Languages and Compilers for Parallel Computing*, Portland, Ore., Aug. 1993, number 768, Lecture Notes in Computer Science, pp. 500–521, Berlin: Springer Verlag.

[17] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Data dependence and data-flow analysis of arrays," *1992 Workshop on Languages and Compilers for Parallel Computing*, New Haven, Conn., Aug. 1992, number 757, Lecture Notes in Computer Science, pp. 434–448, Berlin: Springer Verlag.

[18] Z. Li, "Array privatization for parallel execution of loops," *1992 ACM International Conference on Supercomputing*, Washington, D.C., Jul. 1992, pp. 313–322.

[19] S. Leung and J. Zahorjan, "Improving the performance of runtime parallelization," *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993, pp. 83–91.

[20] C. Zhu and P. C. Yew, "A scheme to enforce data dependence on large multiprocessor systems," *IEEE Trans. Softw. Eng.*, vol. 13, no. 6, pp. 726–739, Jun. 1987.

[21] H. Yu and L. Rauchwerger, "Run-time parallelization overhead reduction techniques," *Proc. of the 9th International Conference on Compiler Construction (CC2000), Berlin, Germany*, pp. 232–248, Mar. 2000.

[22] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization," *Proceedings of the 14th ACM International Conference on Supercomputing, Santa Fe, NM*, pp. 66–77, May 2000.

[23] J. M. Bull, "Feedback guided dynamic loop scheduling: Algorithms and experiments," in *EUROPAR98*, pp. 377–382, Sep. 1998.

[24] 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*, vol. 29, no. 12, pp. 78–82, Dec. 1996.

[25] J. A. Carvallo de Ochoa, "Optimizations enabling tranformations and code generation for the HP V Class," M.S. thesis, Texas A&M University, Department of Computer Science, Aug. 2000.

APPENDIX A

GENERAL PURPOSE ROUTINES FOR THE R-LRPD TEST

```
subroutine RT_STRIP(low, high, num, pos, slow, shigh)
integer*4 low, high, num, pos, slow, shigh
```

This routine partitions the given range of iteration numbers into contiguous sub-ranges based on the number of processors. The low and high variables are the original loop bounds. The num variable is the number of strips, usually the number of processors. The pos variable is the strip number which is usually the processor rank $(0..num-1)$. The slow and shigh variables are the calculated sub-range bounds for the given strip number.

```
subroutine RT_ZEROBYTES(array, numelems)
integer*4 numelems
integer*1 array(numelems)

subroutine RT_ZEROBYTES8(array, numelems)
integer*4 numelems
integer*8 array(numelems)
```

These routines initialize in parallel each element in the one dimensional array to zero. The first routine expects an array of one byte integers while the second routine expects an array of eight byte integers.

```
subroutine RT_COPYARRAY(source, dest, numelems)
integer*4 numelems
datatype source(numelems), dest(numelems)
```

These checkpointing routines save the entire state of an array in parallel before the loop is speculatively executed. The source and dest arrays are expected to be one dimensional arrays.

```
subroutine RT_RESTORE(array, check_array, numelems, istart, iend)
integer*4 numelems, istart, iend
datatype array(numelems), check_array(numelems)
```

This routine restores in parallel the selected range of elements in the original arrays to their state before the loop was speculatively executed. The element range is determined by the iteration assignments to each processor and the lowest ranked processor which encountered a dependence.

```
subroutine RT_COPYIN(global_array, private_array, numelems, numproc,
  padding)
integer*4 numelems, numproc, padding
datatype global_array(numelems + padding, numproc)
datatype private_array(numelems + padding, numproc)
```

In these routines, each processor in parallel copies all elements from the global data array to their section in the expanded data array. The padding ensures that the different processors do not access elements residing on the same cache line and thus prevents cache line contention.

APPENDIX B

ROUTINES FOR THE R-LRPD WITH SHADOW ARRAYS

```
subroutine RLRPD_MARK_WRITE(shadow, writecount)
integer*1 shadow
integer*4 writecount
```

In this write reference marking routine, if the write bit is not already set in the shadow element, the write bit is set in the shadow element and the write count is incremented. Both the shadow and writecount variables are expected to be private to the invoking processor.

```
subroutine RLRPD_MARK_READ(shadow, readfirst, private, global)
integer*1 shadow
integer*4 readfirst
datatype private, global
```

There is a marking routine for each data type (since it handles copy-in of data from the public data element to the processor's private data element) for recording read references during speculative execution of the loop. If the write bit is not already set in the shadow element, the read bit is set in the shadow element, the readfirst flag is set, and the data is copied from the global array element to the private array element. The shadow element, the readfirst flag, and the private element are expected to be private to the invoking processor. The global variable is expected to be an element of the original shared data array.

```
subroutine RT_A_RS_V(shadow_array, writecnt, numelems, numproc, nmf,
  pfail)
integer*4 numelems, numproc, pfail
integer*1 shadow(numelems, numproc)
integer*4 writecnt(16, numproc), nmf(16, numproc)

subroutine RT_A_RS_V8(shadow_array, writecnt, numelems, numproc, nmf,
```

```
  pfail)
integer*4 numelems, numproc, pfail
integer*8 shadow(numelems, numproc)
integer*4 writecnt(16, numproc), nmf(16, numproc)
```

The analysis is necessary only when both read and write references have been recorded during speculative execution. If only read or write references are recorded, the loop is fully parallel due to privatization with copy-in. The per-processor write counts and readfirst flags are checked to determine if the full dependence analysis is necessary. The first routine analyzes one shadow element as a time while the second routine analyzes eight shadow elements simultaneously.

```
subroutine RT_LV_ARRAY(global_array, private_array, shadow_array,
  write_counts, numelems, numproc, pstart, pfail)
integer*4 numproc, pstart, pfail, numelems
integer*1 shadow_array(numelems, numproc)
integer*4 write_counts(16, numproc)
datatype global_array(numelems), private_array(numelems, numproc)
```

This routine performs the last value assignment to determine the latest correctly written value for each array element. The commit routine uses the reference information from the shadow arrays and the list of processors that executed correctly as determined by the dependence analysis. The commit routine is skipped if there are no recorded write references during speculative execution.

APPENDIX C

ROUTINES FOR DDG EXTRACTION WITH HASH TABLES

The data dependence graph (DDG) is a directed graph that is implemented as an array of Fortran derived types. This derived type represents a single node (iteration) in the DDG. Each node maintains the number of incoming and outgoing edges, the node weight, a list of edges, and a list of edge weights for each node. The node weight of an iteration represents the total number of references recorded for that iteration. The edge weight represents the number of flow dependences between two iterations.

```
subroutine init_sw(wstart, wend, nstart, nend, iterperproc, np, its)
integer*4 wstart, wend, nstart, nend, iterperproc, np
integer*4 its(16, np)
```

This routine creates the initial window of iterations. The window size is determined by the desired number of iterations per processor (iterperproc) and the number of processors.

```
subroutine adv_sw(wstart, wend, nend, iterperproc, np, pfail, done,
  its)
integer*4 wstart, wend, nend, iterperproc, np, pfail
logical*4 done
integer*4 its(16, np)
```

This routine advances the window of iterations. Correctly executed iterations are removed from the window, incorrectly executed iterations remain in the window to be re-executed, and new iterations are added to the window if any unscheduled iterations remain. The window size is determined by the desired number of iterations per processor (iterperproc) and the number of processors.

```
subroutine rt_m_r_ihq(nentry, nhash, tsize, np, pid, addr, iter,
  numelems, overflow, of_ptr, hashidx, gdata, pdata, table, hlist)
integer*4 nentry, nhash, tsize, np, pid, addr, iter, numelems
integer*4 overflow, of_ptr, hashidx
logical*4 flag
integer*4 table(nentry, tsize, np), hlist(nhash+1, np)
datatype pdata(tsize, np), gdata(numelems)
```

This routine records the read reference into the processor's shadow hash table. If no reference data exists for the hashed address, reference type, and iteration, then the reference information is added. The data from the global array is copied into the processor's private storage which is a two-dimensional array whose size resembles the size of the shadow hash table.

```
subroutine rt_m_w_ihq(nentry, nhash, tsize, np, pid, addr, iter,
  numelems, overflow, of_ptr, hashidx, table, hlist)
integer*4 nentry, nhash, tsize, np, pid, addr, iter, numelems
integer*4 overflow, of_ptr, hashidx
logical*4 flag
integer*4 table(nentry, tsize, np), hlist(nhash+1, np)
```

This routine records the write reference into the processor's shadow hash table. If no reference data exists for the hashed address, reference type, and iteration, then the reference information is added.

```
subroutine rt_m_rw_ihq(nentry, nhash, tsize, np, pid, addr, iter,
  numelems, overflow, of_ptr, hashidx, gdata, pdata, table, hlist)
integer*4 nentry, nhash, tsize, np, pid, addr, iter, numelems
integer*4 overflow, of_ptr, hashidx
logical*4 flag
integer*4 table(nentry, tsize, np), hlist(nhash+1, np)
datatype pdata(tsize, np), gdata(numelems)
```

This routine records the read and write reference into the processor's shadow hash table. If no reference data exists for the hashed address, reference type, and iteration, then the reference information is added. The data from the global array is copied into the processor's private storage which is a two-dimensional array whose size resembles the size of the shadow hash table.

```
subroutine rt_a_ihqv(nentry, nhash, tsize, np, pfail, niter,
  iterperproc, wnentry, wnhash, wtsize, wstart,
  its, table, wtable, hlist, dtype_redges)
integer*4 nentry, nhash, tsize, np, pfail, iterperproc
integer*4 niter, wnentry, wnhash, wtsize
integer*4 table(nentry, tsize, np), its(16, np)
integer*4 wtable(wnentry, wtsize, np), hlist(nhash+1, np)
type (graph_node), dimension(niter) :: dtype_redges
```

The analysis checks for cross-iteration dependences between iterations within the current window. It also checks for dependences between successfully completed iterations and iterations inside the current window by examining the distributed table which contains the last write reference for each memory address. This routine also tentatively stores any detected dependence (both cross-window and cross-processor) edges. Any dependence edges occurring after the earliest cross-iteration dependence within the current window are removed in the edge commit routine.

```
subroutine rt_c_ihq_r8(nentry, nhash, tsize, np, pstart, pfail,
  numelems, wnentry, wnhash, wtsize,
  wt_overflow_ptr, wtable, its, table, parray, garray, hlist)
integer*4 nentry, nhash, tsize, np, pstart, pfail, numelems
integer*4 wnentry, wnhash, wtsize
integer*4 table(nentry, tsize, np), its(16, np)
integer*4 wtable(wnentry, wtsize, np), hlist(nhash+1, np)
integer*4 wt_overflow_ptr(16, np)
real*8 parray(tsize, np), garray(numelems)
```

This routine performs the last value assignment using information from the shadow hash tables. For any successfully executed iteration in the current window, this routine also commits the latest valid write reference for each memory address into a distributed hash table.

```
subroutine rt_c_e(iterperproc, niter, np, pfail, its, dtype_redges)
integer*4 iterperproc, niter, np, pfail
integer*4 its(16, np)
type (graph_node), dimension(niter) :: dtype_redges
```

Any cross-processor dependence within the current window of iterations can produce invalid computations and edges in the DDG. This routine removes any edges

in the DDG that were extracted during the analysis phase for iterations that were not completed successfully.

```
subroutine rt_zerotable_hl(nentry, nhash, tsize, np, hlist, table)
integer*4 nentry, nhash, tsize, np
integer*4 hlist(nhash+1, np), table(nentry, tsize, np)
```

This routine reinitializes the shadow hash table by zeroing only the hash indices that were modified by each processor during the previous speculative execution.

```
subroutine rt_reverse_edges(ibeg, iend, niter, np, dtype_redges,
    dtype_edges)
integer*4 ibeg, iend, niter, np
type (graph_node), dimension(niter) :: dtype_redges, dtype_edges
```

During the analysis, the edges in the DDG are collected as $(read, write)$ pairs. This routine converts the DDG into $(write, read)$ pairs and calculates the number of incoming and outgoing edges for each node in the DDG.

```
subroutine rt_sch_wf(ibeg, iend, niter, np, nlevels, maxiter,
  dtype_edges, iqueue, wf_its)
integer*4 ibeg, iend, niter, np, nlevels, maxiter
integer*4 iqueue(maxiter), wfits(2, maxiter)
type (graph_node), dimension(niter) :: dtype_edges
```

This routine schedules the DDG as a series of wavefronts which can be executed in parallel. The iterations in the same wavefront are stored contiguously in the *iqueue* array. The $wfits$ array maintains the starting and ending elements in the iqueue array for each wavefront.

VITA

Francis Hoai Dinh Dang received his Bachelor of Science degree in Computer Engineering from Texas A&M University at College Station, Texas in May 1999. He entered the Computer Science graduate program at Texas A&M University in September 1999. He has been working at the Texas A&M Supercomputing Facility since 2001.

Mr. Dang can be reached at the Texas A&M Supercomputing Facility, Computing and Information Services, 3363 TAMU, College Station, TX 77843-3363.