The LRPD Test: Speculative Run–Time Parallelization of Loops with Privatization and Reduction Parallelization †

Lawrence Rauchwerger and David Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801

Corresponding Author: Lawrence Rauchwerger telephone: (217) 244-0070
email: `rwerger@csrd.uiuc.edu` fax: (217) 244-1351

Abstract

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. As parallelizable loops arise frequently in practice, we advocate a novel framework for their identification: *speculatively execute the loop as a* `doall`*, and apply a fully parallel data dependence test to determine if it had any cross–iteration dependences*; if the test fails, then the loop is re–executed serially. Since, from our experience, a significant amount of the available parallelism in Fortran programs can be exploited by loops transformed through *privatization* and *reduction parallelization*, our methods can speculatively apply these transformations and then check their validity at run–time. Another important contribution of this paper is a *novel method for reduction recognition* which goes beyond syntactic pattern matching: it detects at run–time if the *values* stored in an array participate in a reduction operation, even if they are transferred through private variables and/or are affected by statically unpredictable control flow. We present experimental results on loops from the PERFECT Benchmarks which substantiate our claim that these techniques can yield significant speedups which are often superior to those obtainable by inspector/executor methods.

The methods presented in this paper differ from and extend *our* previous work on several important points. First, instead of distributing the loop into inspector and executor loops (the approach taken in *all* previous work on run–time parallelization) we advocate the use of run–time tests to validate the execution of a loop that is speculatively executed in parallel. Second, in addition to array privatization, the new techniques are capable of testing the validity of the powerful reduction parallelization transformation at run–time. Finally, the new algorithms consider only data dependences caused by actual cross–iteration data–flow (a flow of values) and thus potentially qualify more loops as parallel than possible with our previous run–time data dependence test.

†Research supported in part by Intel and NASA Graduate Fellowships, and Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

1 Introduction

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. Restructuring, or parallelizing, compilers address these problems by detecting and exploiting parallelism in sequential programs written in conventional languages. Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [24, 34]), 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. One major reason for this inability to statically parallelize some programs is that the most effective transformations, *privatization* and *reduction recognition*, cannot be applied to a large class of applications that have irregular domains and/or dynamically changing interactions. Typical examples are complex simulations such as SPICE for circuit simulation, DYNA–3D and PRONTO–3D for structural mechanics modeling, GAUSSIAN and DMOL for quantum mechanical simulation of molecules, CHARMM and DISCOVER for molecular dynamics simulation of organic systems, and FIDAP for modeling complex fluid flows [11].

Thus, in order to realize the full potential of parallel computing it has become clear that static (compile–time) analysis must be complemented by new methods capable of automatically extracting parallelism at *run–time* [9, 11, 13]. 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. In contrast, at compile–time the access pattern of some programs cannot be determined, sometimes due to limitations in the current analysis algorithms but often because the necessary information is just not available, i.e., the access pattern is a function of the input data. For example, most dependence analysis algorithms can only deal with subscript expressions that are linear in the loop indices. In the presence of non–linear expressions, a dependence is usually assumed. Also, generally compilers conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically–known values, nothing can be done at compile–time when the index arrays are a function of the input data [19, 29, 36].

Most previous approaches to run–time parallelization have concentrated on developing methods for constructing execution schedules for partially parallel loops, i.e., loops whose parallelization requires synchronization to ensure that the iterations are executed in the correct order.[1] These methods are centered around the extraction of an *inspector* loop that analyzes the data access pattern "off–line," i.e., without side effects [8, 19, 22, 26, 27, 28, 29, 35, 36]. The inspection phase of these schemes usually yields a partitioning of the set of iterations into subsets that can be executed in parallel. These subsets, sometimes called *wavefronts*, are scheduled sequentially by placing synchronization barriers between them.

Unfortunately the distribution of the original loop into an inspector and executor loop is often not advantageous: if the address computation of the array under test depends on the actual data computation, as exemplified by Fig. 1(a), then the inspector becomes both computationally expensive and has side–effects. This means that shared arrays would be modified during the execution of the inspector loop and saving the state of these variables would be required – making the inspector equivalent to the loop itself. In addition, the desirable goal of exploiting coarse–grain parallelization, i.e., at the level of large complex loops, makes it even less likely that an appropriate "inspector" loop can be extracted. Thus, the inspector/executor approach is not a generally applicable method, i.e., it is limited to special cases.

1.1 Our approach: speculative `doall` parallelization

In this paper we propose a novel framework for parallelizing `do` loops at run–time. The proposed framework differs conceptually from previous methods in two major points.

- Instead of finding a valid parallel execution schedule for the loop, we focus on the problem of simply deciding if the loop is fully parallel, that is, determining whether or not the loop has cross–iteration dependences. (This approach was also taken in [26].)

[1] The only exception of which we are aware is our inspector method for `doall` parallelization [26]. Run–time analysis techniques have also been used to detect *access anomalies* or *race conditions* in parallel programs (see, e.g., [12, 23, 30]). However, these methods are generally not appropriate for run–time loop parallelization since they are optimized for other purposes, e.g., for them minimizing memory requirements is more important than speed.

- Instead of distributing the loop into inspector and executor loops, we *speculatively* execute the loop as a `doall`, i.e., execute all its iterations concurrently, and apply a run–time test to check if there were any cross–iteration dependences. If the run–time test fails, then we will pay a penalty in that we need to backtrack and re–execute the loop serially.

Compilers often transform programs to optimize performance. The two most effective transformations for increasing the amount of parallelism in a loop (i.e., removing certain types of data dependences) are *array privatization* and *reduction parallelization*. Krothapalli and Sadayappan [14] proposed an inspector method for run–time privatization which relies heavily on synchronizaton, inserts an additional level of indirection into all memory accesses, and calls for dynamic shared memory allocation. In our previous work [26] we gave an inspector method without these drawbacks for determining whether a `do` loop can be executed as a `doall`, perhaps by privatizing some shared variables. No previous run–time methods have been proposed for parallelizing reduction operations.

The methods presented in this paper differ from our previous work [26] in several important ways. First, we advocate the use of run–time tests to validate the execution of a loop that is speculatively executed in parallel. The advantage of this approach is that the computation of the loop is performed concurrently with the tests, i.e., the memory access pattern does not need to be extracted and analyzed separately as in inspector/executor methods.[2] In Section 5 we present experimental results on loops from the PERFECT Benchmarks which substantiate our claim that speculative techniques can yield significant speedups which are often superior to those obtainable by inspector/executor methods. Second, in addition to array privatization, the new techniques are capable of testing at run–time the validity of the powerful reduction parallelization transformation. In particular, for an array element (or section), our run–time methods are able to detect whether it participated exclusively in a reduction operation, or if all its accesses were either read–only or privatizable. If all the memory references in a `do` loop fall under any of these categories then the speculative concurrent execution of the loop was valid, i.e., the loop was indeed parallel. Finally, the new algorithms consider only data dependences caused by actual cross–iteration data–flow (a flow of values). Thus, they may potentially qualify more loops as parallel than the method in [26] which conservatively considered the dependences due to every memory reference – even if no cross–iteration data–flow occurred at run–time. This situation could arise for example when a loop reads a shared variable, but then only uses it conditionally.

Another important contribution of this paper is a *novel method for reduction recognition*: in contrast to the static pattern matching techniques employed by compilers until now, our method detects if the *values* stored in an array participate in a reduction operation, even if they are transferred through private variables and/or are affected by statically unpredictable control flow.

Our methods for speculatively executing `do` loops in parallel are described in Sections 3 and 4. In Section 5, we present some experimental measurements of loops from the PERFECT Benchmarks executed on the Alliant FX/80 and 2800. These measurements show that the techniques presented in this paper are effective in producing scalable speedups even though the run–time analysis is done without the help of any special hardware devices. It is conceivable, and we believe desirable, that future machines would include special hardware devices to accelerate the run–time analysis and in this way widen the range of applicability of the techniques and increase potential speedups.

2 Preliminaries

A loop can be executed in fully parallel form, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the execution ordering of the data accesses from different iterations. In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the *data dependence* relations between the statements in the loop body must be analyzed [6, 17, 24, 34, 37]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write), *anti* (write after read), and *output* (write after write). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory–related dependences, are caused by the reuse of memory, e.g., program variables.

If there are flow dependences between accesses in different iterations of a loop, then the semantics of the loop cannot be guaranteed if the loop is executed in fully parallel form. For example, the iterations of the loop in Fig. 1(a) must be executed in order of iteration number because iteration $i+1$ needs the value that is produced in iteration i, for $1 \leq i < n$. In principle, if there are no flow dependences between the iterations of a loop, then the loop may be

[2] If desired, all of our run–time tests can be applied in inspector/executor mode.

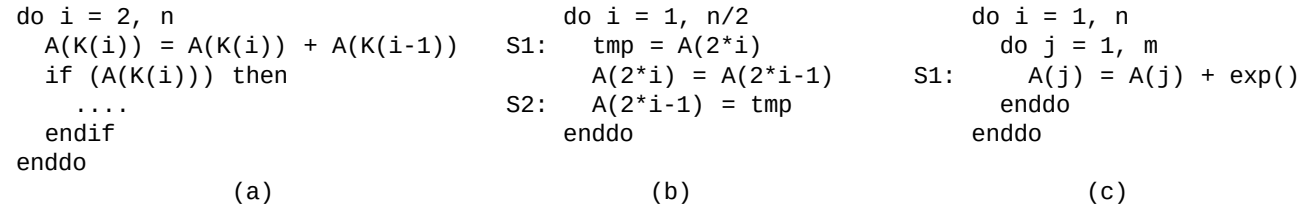


Figure 1:

executed in fully parallel form. The simplest situation occurs when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and the loop, as is, can be executed as a `doall` (i.e., a fully parallel execution). If there are no flow dependences, but there are anti or output dependences, then the loop must be modified to remove all these dependences before it can be executed in parallel. Not all such situations can be handled efficiently. In order to remove certain types of dependences and execute the loop as a `doall`, two important and effective transformations can be applied to the loop: *privatization* and *reduction parallelization*.

Privatization creates, for each processor cooperating on the execution of the loop, private copies of the program variables that give rise to anti or output dependences (see, e.g., [10, 20, 21, 31, 32]). The loop shown in Fig. 1(b), is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S2` of iteration i and statement `S1` of iteration $i+1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`. In this paper, the following criterion is used to determine whether a variable may be privatized.

Privatization Criterion: Let A be a shared array (or array section) that is referenced in a loop L. A can be *privatized* if and only if every read access to an element of A is preceded by a write access to that same element of A within the same iteration of L.

In general, dependences that are generated by accesses to variables that are only used as workspace (e.g., temporary variables) *within* an iteration can be eliminated by privatizing the workspace. However, according to the above criterion, if a shared variable is initialized by reading a value that is computed outside the loop, then that variable cannot be privatized. Such variables could be privatized if a *copy–in* mechanism for the external value is provided. The *last value assignment* problem is the conceptual analog of the copy–in problem. If a privatized variable is *live* after the termination of the loop, then the privatization technique must ensure that the correct value is copied out to the original (non privatized) version of that variable. It should be noted that the need for values to be copied into or out of private variables occurs infrequently in practice.

Reduction parallelization is another important technique for transforming certain types of data dependent loops for concurrent execution.

Definition: 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 associative and commutative operator and x does not occur in exp or anywhere else in the loop.

Reduction variables are therefore accessed in a certain specific pattern (which leads to a characteristic data dependence graph). A simple but typical example of a reduction is statement `S1` in Fig. 1(c). The operator $\otimes$ is exemplified by the $+$ operator, the access pattern of array $A(:)$ is *read, modify, write*, and 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* and occurs quite frequently in programs.

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.) Parallel methods are known for performing reduction operations. One typical method is to transform the `do` loop into a `doall` and enclose the access to the reduction variable in an unordered critical section [13, 37]. Drawbacks of this method are that it is not scalable and requires synchronizations which can be very expensive in large multiprocessor systems. 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 [15, 16, 18]. In this case the reduction variable is privatized in the transformed `doall`, and the final result of the reduction operation is computed in an interprocessor reduction phase following the `doall`, i.e., a scalar is produced using the partial results computed in each processor as operands for a reduction operation (with the same operator) across the processors. Thus, the difficulty encountered

by compilers in parallelizing loops with reductions arises not from finding a parallel algorithm but from recognizing 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 [37].

3 Speculative Parallel Execution of `do` Loops

Consider a `do` loop for which the compiler cannot statically determine the access pattern of a shared array A that is referenced in the loop. Instead of generating pessimistic, sequential code when it cannot unequivocally decide whether the loop is parallel, the compiler could decide to *speculatively* execute the loop as a `doall`, and produce code to determine at run–time whether the loop was in fact fully parallel. In addition, if it is suspected that some data dependences could be removed by privatization and/or reduction parallelization the compiler may further speculatively apply these transformations in order to increase the chances that the loop can be executed as a `doall`. If the subsequent run–time test finds that the loop was not fully parallel, then it will be re–executed sequentially. In order to speculatively parallelize a `do` loop as outlined above we need the following:

- *A mechanism of saving/restoring state:* to save the original values of the program variables for the possible sequential re-execution of the loop.
- *An error (hazard) detection method:* to test the validity of the speculative parallel execution.
- *An automatable strategy:* to decide when to use speculative parallel execution.

Saving/Restoring State. There are several ways to maintain backups of the program variables that may be altered by the speculative parallel execution. If the resources (time and space) needed to create a backup copy are not too big, then a practical solution is checkpointing prior to the speculative execution. It might be possible to reduce this cost by identifying and checkpointing a point of minimum state in the program prior to the speculative parallel execution. A more attractive solution is to privatize all shared variables, copy–in (on demand) any needed external values, and copy–out any live values if the test passes, thereby committing the results computed by the `doall` loop. This method could also yield better data locality and reduce the number of messages between processors (e.g., it would generate less coherency traffic in a cache coherent distributed shared–memory machine). Note that privatized arrays need not be backed up because the original version of the array will not be altered during the parallel execution.

Hazard Detection. There are essentially two types of errors (hazards) that could occur during the speculative parallel execution: (i) exceptions and (ii) the presence of cross–iteration dependences in the loop. A simple way to deal with exceptions is to treat them as an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, clear the exception flag, restore the values of any altered program variables, and execute the loop sequentially. Below, we present techniques that can be used to detect the presence of cross–iteration dependences in the loop and to test the validity of any privatization and/or reduction parallelization transformations that were applied.

An Automatable Strategy. The main factors that the compiler should consider when deciding whether to speculatively parallelize a loop are: the probability that the loop is a `doall`, the speedup obtained if the loop is a `doall`, and the slowdown incurred if the loop is not a `doall`. For example, the compiler might base its decision on a ratio of the estimated run–time cost of an erroneous parallel execution to the estimated run–time cost of a sequential execution. If this ratio is small, then significant performance gains would result from a successful (valid) parallelization of the loop, at the risk of increasing the sequential execution time by only a small amount. In order to perform a cost/benefit analysis and to predict the parallelism of the loop, the compiler should use static analysis and run–time statistics (collected on previous executions of the loop or from different codes); in addition, directives about the parallelism of the loop might prove useful. In Section 4.1 a complexity analysis of our run–time tests is presented that can be used to statically predict the minimum obtainable speedup and the maximum potential slowdown for a loop parallelized using our techniques.

3.1 Run–time data dependence analysis

In this section we describe an efficient run–time technique that can be used to detect the presence of cross–iteration dependences in a loop that has been speculatively executed in parallel. If there are any such dependences, then this test

will not identify them, it will only flag their existence. We note that the test need only be applied to those scalars and arrays that cannot be analyzed at compile–time. In addition, if any shared variables were privatized for the speculative parallel execution, then this test can determine whether those variables were in fact validly privatized.

An important source of ambiguity that cannot be analyzed statically and potentially generates overly conservative data dependence models is the run–time equivalent of *dead code*. A simple example is when a loop first reads a shared array element into a local variable but then only conditionally uses it in the computation of other shared variables. If the consumption of the read value does not materialize at run–time, then the read access did not in fact contribute to the data flow of the loop and therefore could not have caused a dependence. Since predicates seldom can be evaluated statically, the compiler must be conservative and conclude that the read access causes a dependence in every iteration of the loop. The test given here improves upon the Privatizing `doall` test described in [26] by checking only the dynamic data dependences caused by the actual cross–iteration flow of values stored in the shared arrays. This is accomplished using a technique we call *dynamic dead reference elimination* which is explained in detail following the description of the test.

The most general version of the test, as applied to a privatized shared array A, is given below, i.e., it tests for all types of dependences, and also whether the array is indeed privatizable. If some of these conditions do not need to be verified, then the test can be simplified in a straitforward manner, e.g., if the array was not privatized for the speculative parallel execution, then all steps pertaining to the privatization check are omitted.

The Lazy (value–based) Privatizing `doall` Test (LPD Test)

1. *Marking Phase.* (Performed during the speculative parallel execution of the loop.) For each shared array $A[1:s]$ whose dependences cannot be determined at compile time, we declare read and write shadow arrays, $A_r[1:s]$ and $A_w[1:s]$, respectively. In addition, we declare a shadow array $A_{np}[1:s]$ that will be used to flag array elements that *cannot* be validly privatized. Initially, the test assumes that all array elements *are* privatizable, and if it is found in any iteration that the value of an element is used (read) before it is redefined (written), then it will be marked as not privatizable. The shadow arrays A_r, A_w, and A_{np} are initialized to zero.

   During each iteration of the loop, all definitions or uses of the values stored in the shared array A are processed:

   (a) Definitions (done when the value is written): set the element in A_w corresponding to the array element that is modified (written).

   (b) Uses (done when the value that was read is used): if this array element is *never* modified (written) in this iteration, then set the corresponding element in A_r. If the value stored in this array element has not been written in this iteration before this use (read access), then set the corresponding element in A_{np}, i.e., mark it as *not* privatizable.

   (c) Count the total number of write accesses to A that are marked in this iteration, and store the result in $tw_i(A)$, where i is the iteration number.

2. *Analysis Phase.* (Performed after the speculative parallel execution.) For each shared array A under scrutiny:

   (a) Compute (i) $tw(A) = \sum tw_i(A)$, i.e., the total number of definitions (writes) that were marked by all iterations in the loop, and (ii) $tm(A) = sum(A_w[1:s])$, i.e., the total number of marks in $A_w[1:s]$.

   (b) If $any(A_w[:] \wedge A_r[:])$,[3] i.e., if the marked areas are common *anywhere*, then the loop *is not* a `doall` and the phase ends. (Since we define (write) and use (read, but do not define) values stored at the same location in different iterations, there is at least one flow or anti dependence.)

   (c) Else if $tw(A) = tm(A)$, then the loop *is* a `doall` (without privatizing the array A). (Since we never overwrite any memory location, there are no output dependences.)

   (d) Else if $any(A_w[:] \wedge A_{np}[:])$, then the array A *is not* privatizable. Thus, the loop, as executed, *is not* a `doall` and the phase ends. (There is at least one iteration in which some element of A was used (read) before it was been modified (written).)

   (e) Otherwise, the loop was made into a `doall` by privatizing the shared array A. (We remove all memory–related dependences by privatizing this array.)

```
do i = 1,n
   z = A(K(i))
   if B1(i) then
     A(L(i)) = z + C(i)
   endif
enddo

B1(1:4) = (1 0 1 0 1)
 K(1:4) = (1 2 3 4 1)
 L(1:4) = (2 2 4 4 2)
```

(a)

```
doall i = 1,n
   markread(K(i))
   z = A(K(i))
   if B1(i) then
     markwrite(L(i))
     A(L(i)) = z + C(i)
   endif
enddoall
```

(b)

```
doall i = 1,n
   z = A(K(i))
   if B1(i) then
     markread(K(i))
     markwrite (L(i))
     A(L(i)) = z + C(i)
   endif
enddoall
```

(c)

original PD test	shadow arrays 1	2	3	4	tw	tm
A_w	0	1	0	1	3	2
A_r	1	1	1	1		
A_{np}	1	1	1	1		
$A_w(:) \wedge A_r(:)$	0	1	0	1		
$A_w(:) \wedge A_{np}(:)$	0	1	0	1		

(d)

new LPD test	shadow arrays 1	2	3	4	tw	tm
A_w	0	1	0	1	3	2
A_r	1	0	1	0		
A_{np}	1	0	1	0		
$A_w(:) \wedge A_r(:)$	0	0	0	0		
$A_w(:) \wedge A_{np}(:)$	0	0	0	0		

(e)

Figure 2: The transformation of a `do` loop (a), using the original version of the PD test (b), and the lazy version (c). The `markwrite` (`markread`) operation marks the indicated element in the shadow array A_w (A_r and A_{np}) according to the criteria given in Step 1(a) (1(b)) of the LPD test. Since dynamic dead read references are not marked in the LPD test, the array A fails the PD test and passes the LPD test, as shown in (d) and (e), respectively.

Dynamic dead reference elimination. We now describe how the marking of the read and private shadow arrays, A_r and A_{np}, can be postponed until the value of the shared variable is actually used (Step 1(b)). More formally, the references we want to identify are defined as follows.

Definition A *dynamic dead read reference* is a read access of a shared variable that both
(a) does not contribute to the computation of any other shared variable, and
(b) does not control (predicate) the references to other shared variables.

The value obtained through a dynamic dead read does not contribute to the data flow of the loop. Ideally, such accesses should not introduce false dependences in either the static or the run–time dependence analysis. If it is possible to determine the dead references at compile time then we can just ignore them in our analysis. Since this is generally not possible (control flow could be input dependent) the compiler should identify the references that have the potential to be unused and insert code to solve this problem at run–time. In Fig. 2 we give an example where the compiler can identify such a situation by following the *def–use chain* built by using array names only. To avoid introducing false dependences, the marking of the read shadow array is postponed until the value that is read into the loop space is indeed used in the computation of other shared variables. In essence we are concerned with the flow of the values stored rather than with their storage (addresses). We note that if the search for the actual use of a read value becomes too complex then it can be stopped gracefully at a certain depth and a conservative marking of the shadow array can be inserted (on all the paths leading to a possible use).

As can be observed from the example in Fig. 2, this method allows the LPD test to qualify more loops for parallel execution then would be otherwise possible by just inspecting the memory references as in the original PD test [26]. In particular, after marking and counting we obtain the results depicted in the tables. The loop fails the PD test since $A_w(:) \wedge A_r(:)$ is not zero everywhere (Step 2(b)). However, the loop passes the LPD test as $A_w(:) \wedge A_r(:)$ is zero everywhere, but only after privatization, since $tw(A) \neq tm(A)$ and $A_w(:) \wedge A_{np}(:)$ is zero everywhere.

Private shadow structures. The LPD test can take advantage of the processors' private memories by using private shadow structures for the marking phase of the test. Then, at the conclusion of the private marking phase, the contents of the private shadow structures are merged into the global shadow structures. Note that since the order of the writes (marks) to an element of the shadow structure is not important, all processors can transfer their private shadow structures to the global structure without synchronization. In fact, using private shadow structures enables some additional optimization of the LPD test as follows. Since the shadow structures are private to each processor,

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

the iteration number can be used as the "mark." In this way, no re–initialization of the shadow structures is required between successive iterations, and checks such as "has this element been written in this iteration?" simply require checking if the corresponding element in A_w is marked with the iteration number. Another benefit of the iteration number "marks" is that they can double as time–stamps, which are needed for performing the last–value assignment to any shared variables that are live after loop termination.

A processor–wise version of the LPD test. The LPD Test determines whether a loop has any cross–*iteration* data dependences. It turns out that essentially the same method can be used to test whether the loop, as executed, has any cross–*processor* data dependences.[4] The only difference is that all checks in the test refer to processors rather than to iterations, i.e., replace "iteration" by "processor" in the description of the LPD test so that all iterations assigned to a processor are considered as one "super–iteration" by the test. Note that a loop that is not fully parallel could potentially pass the processor–wise version of the LPD test because data dependences among iterations assigned to the same processor will not be detected. This is acceptable (even desirable) as long each processor executes its assigned iterations in increasing order.

3.2 Run–time techniques for reduction parallelization

As mentioned in Section 2, there are two tasks required for reduction parallelization: *recognizing the reduction variable*, and *parallelizing the reduction operation*. Of these, we focus our attention on the former since, as previously noted, techniques are known for performing reduction operations in parallel. So far the problem of reduction variable recognition 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 validate it as a reduction variable [37]. There are two major shortcomings of such pattern matching identification methods.

1. The data dependence analysis necessary to qualify a statement as a reduction cannot be performed statically in the presence of input–dependent access patterns.

2. Syntactic pattern matching cannot identify all potential reduction variables (e.g., in the presence of subscripted subscripts).

In the next two sections we show how each of these two difficulties can be overcome with a combination of static and run–time methods.

3.2.1 The LRPD test: extending the LPD test for reduction validation

In this section we consider the problem of verifying that a statement is a reduction using run–time data dependence analysis. The potential reduction statement is assumed to syntactically pattern match the generic reduction template $x = x \otimes exp$; reduction statements that do not meet this criterion are treated in the next section. To verify that such a statement is a reduction we need to check that the reduction variable x satisfies the definition given in Section 2, i.e., that x is only accessed in the reduction statement, and that it does not appear in exp. In addition, if x is a reduction variable in several potential reduction statements in the loop, then it must also be verified that each of these reduction statements has the same operator.

Our basic strategy is to extend the LPD test to check all statically unverifiable reduction conditions. We first consider how the test would be augmented to check only that the reduction variable is not accessed outside the single reduction statement. This situation could arise if the reduction variable is an array element accessed through subscripted subscripts and the subscript expressions are not statically analyzable. For example, although statement S3 in the loop in Fig. 3(a) matches a reduction statement, it is still necessary to prove that the elements of array A referenced in S1 and S2 do not overlap with those accessed in statement S3, i.e., that: $K(i) \neq R(j)$ and $L(i) \neq R(j)$, for all $1 \leq i, j \leq n$. Thus, the LRPD test must check at run–time that there is no intersection between the references in S3 and those in S1 and/or S2; in addition it will be used to prove, as before, that any cross–iteration dependences in S1 and S2 are removed by privatization. To test this new condition we use another shadow array A_{nx} to flag the array elements that *are not* valid reduction variables. *Initially, all array elements are assumed to be valid reduction variables*, i.e., $A_{nx}[:] = false$. In the marking phase of the test, i.e., during the speculative parallel execution of the loop, *any array element defined or used outside the reduction statement is invalidated as a reduction variable*, i.e., its

[4] This fact was noted by Santosh Abraham[1].

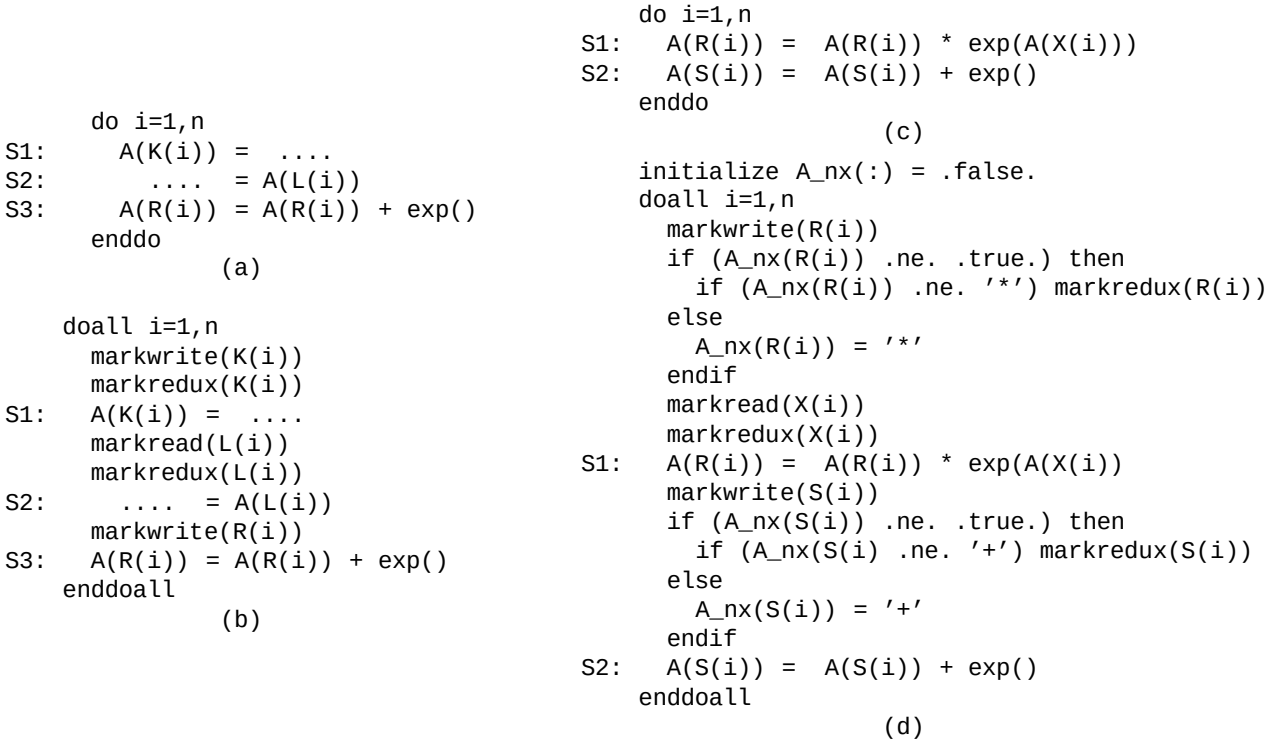


Figure 3: The transformation of the `do` loops in (a) and (c) is shown in (b) and (d), respectively. The `markwrite` (`markread`) operation marks the indicated element in the shadow array A_w (A_r and A_{np}) according to the criteria given in Step 1(a) (1(b)) of the LPD test. The `markredux` operation sets the shadow array element of `A_nx` to true. In (d), the type of the reduction is tested by storing the operator in `A_nx`.

corresponding element in A_{nx} is set to true. As before, after the speculative parallel execution, the analysis phase of the test is performed. *An element of A is a valid reduction variable if and only if it was not invalidated during the marking phase*, i.e., it was not marked in A_{nx} as not a reduction variable for any iteration. The other shadow arrays A_{np}, A_w and A_r are initialized, marked, and interpreted just as before.

The LRPD test can also solve the case when the *exp* part of the RHS of the reduction statement contains references to the array A that are different from the pattern matched LHS and cannot be statically analyzed. To validate such a statement as a reduction we must show that no reference in *exp* overlaps with those of the LHS. This is done during the marking phase by setting an element of A_{nx} to true if the corresponding element of A is referenced in *exp*.

In summary, the LRPD test is obtained by modifying the LPD test. The following step is added to the *Marking Phase.*

1(d) Definitions *and* uses: if a reference to A *is not* one of the two known references to the reduction variable (i.e., it is outside the reduction statement or it is contained in *exp*), then set the corresponding element of A_{nx} to true (to indicate that the element is *not* a reduction variable). (See Fig. 3(a) and (b).)

In the *Analysis Phase*, Steps 2(d) and 2(e) are replaced by the following.

2(d′) Else if $any(A_w[:] \wedge A_{np}[:] \wedge A_{nx}[:])$, then some element of A written in the loop is neither a reduction variable nor privatizable. Thus, the loop, as executed, *is not* a `doall` and the phase ends. (There exist iterations (perhaps different) in which an element of A is not a reduction variable, and in which it is used (read) and subsequently modified.)

2(e′) Otherwise, the loop was made into a `doall` by parallelizing reduction operations and privatizing the shared array A. (All data dependences are removed by these transformations.)

If the analysis phase validates (passes) the speculative parallel execution of the loop, then, as before, the last–value assignments are performed for any live shared variables, and *the scalar result of each reduction is computed using the processors' partial results in a reduction across the processors.* (See Fig. 7.) (If reductions are implemented by placing the reduction statements in unordered critical sections, then this last step is not necessary.)

Multiple potential reduction statements. A more complicated situation is when the loop contains several reduction statements that refer to the same array A. In this case the type of the reduction operation performed on each element must be the same throughout the loop execution, e.g., a variable cannot participate in both a multiplicative and an additive reduction since the resulting operation is not commutative and associative and is therefore not parallelizable. The solution to this problem is to mark the shadow array A_{nx} with the reduction type. Whenever a reference in a reduction statement is marked, the current reduction type (e.g., summation, multiplication) is checked with with previous one. If they are not the same, the corresponding shadow element of A_{nx} is set to true.

In Fig 3(c) and (d), we show how a loop containing two potential reduction statements with different operators and an *exp* operand that contains references to the array under test can be transformed to perform a run–time dependence and reduction test. The subsequent analysis of the shadow arrays will detect which elements were used in a reduction and which are privatizable or read–only. If any element is found not to belong to one of these categories, then the speculative parallelization was incorrect and a sequential re–execution must be initiated.

As a final remark, we note that a more aggressive implementation could promote the type of a reduction at run–time: if a memory element is first involved in a '+' reduction and then switches over to a '*' reduction and *stays that way for all the remaining references*, then the speculative parallel execution can still yield valid partial results on each processor. It is important to remember that a reduction type can be promoted in only one direction (it cannot be demoted back to its initial type) and only once per loop invocation. Of course, the reduction across processors must reflect the reduction operator promotion.

3.2.2 Static reduction recognition and run–time check

As mentioned at the beginning of this section, syntactic pattern matching is not a sufficiently powerful method to detect all the values that are "subject" to a reduction operation. In particular, syntactic pattern matching will fail to identify a reduction whenever all the references on the RHS of the assignment "look different" from the reference on the LHS. Thus, if a statement is in fact a reduction, but the references on the LHS and/or the RHS are indirect, then syntactic pattern matching will fail. This situation could arise naturally, e.g., through the use of temporary variables or subscripted subscripts. In the latter case, it can only be determined at run–time if any of the array elements are reduction variables.

In the following we show that a combination of static and run–time techniques can be used to successfully identify several types of potential reductions that could not be recognized with pattern matching techniques. The general strategy is to speculate that every assignment to the array of interest is a potential reduction, unless proven otherwise statically or by other heuristics. At run–time this assumption is then validated or invalidated on an element by element basis.

Single statement reduction recognition

We first consider a single statement in which the references on the RHS are either dependent on the array A (also referenced on the LHS) or are to values known to be independent of A, e.g., constants, loop invariants, or distinct global variables.

The simplest case is when the RHS contains exactly one reference to A. Consider the potential reduction statement $A(R(i)) = A(X(i)) + exp$. If $R(i) = X(i)$, for some values of i, then the probability that the surrounding loop is parallel is increased. In this case, the solution is simply to check this equality condition at run–time, and mark the shadow array A_{nx} accordingly.

The situation is a bit more complex when the RHS contains multiple references to the array A. Consider the statement $A(R(i)) = A(X_1(i)) + A(X_2(i)) + \ldots + A(X_k(i))$. This statement is a reduction if and only if $R(i) = X_j(i)$ for exactly *one* value of j (see Section 2). As the operation is commutative and associative, we cannot discount the possibility of a reduction. In this example, we must check for equality between $R(i)$ and *every* $X_j(i)$, $1 \leq j \leq k$. If this equality condition is not met exactly once, then $A_{nx}(R(i))$ is set to true (to indicate it was not a reduction). We note that a more aggressive strategy could be taken when there are multiple references to $A(R(i))$ on the RHS: promote the '+' reduction to a '*' reduction. However, as mentioned in Section 3.2.1, the reduction type can only be promoted once in the entire loop. Fig. 4 shows the code generated for run–time validation when the RHS contains multiple references to A. In the interest of clarity, reduction type promotion is not shown.

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

(a)

```
    proc checkequal(x,y,ct)
    {
      if (x .ne. y) then
        markredux(x)
      else
        ct = ct + 1
      endif
    }
```

(b)

```
    initialize A_nx(:) = .false.
    doall i=1,n
      private int count = 0
      markwrite(K(i))
      markredux(K(i))
S1:   A(K(i)) = ....
      markread(L(i))
      markredux(L(i))
S2:     ....  = A(L(i))
      markread(R(i))
      markread(T(i))
      markread(X(i))
      markwrite(S(i))
      checkequal(R(i),S(i),count)
      checkequal(T(i),S(i),count)
      checkequal(X(i),S(i),count)
  /*type could be promoted if count = 3*/
      if (count .ne. 1) markredux(S(i))
S3:   A(S(i)) =  A(R(i)) + A(T(i)) +  A(X(i))
    enddoall
```

(c)

Figure 4: The code generated for the `do` loop in (a) is shown in (c). In (c), the procedure in (b) is called, and the `mark-x` operations are as described in Fig. 3.

Multiple statement reduction recognition: Expanded Reduction Statements

We now relax all restrictions on the RHS and allow in it variables that are neither explicit functions of the array appearing on the LHS nor explicit loop invariants. Our goal is to uncover any possible link between the LHS and the RHS, if indeed one exists. The general strategy of our methods is a fairly straightforward demand driven forward substitution of all the variables on the RHS, a process by which all control flow dependences are substituted by data dependences as described in [2, 32]. Once this expression of the RHS is obtained it can be analyzed and validated by the methods described in the previous section. In the following we explain by way of example how our new method can identify reductions by performing in essence a *value–based* rather than a dependence–based analysis.

In Fig. 5(a) statement S3 is first labeled at compile time as a potential reduction. Then, by following the *def–use chains* of the variables on the RHS (i.e., z and y) *within the scope of the loop* we find that in statement S1 z may potentially carry the value of $A(R(i))$, while y is a constant with respect to A. The algorithm then *examines* statement S3 after forward substitution, but *does not actually replace* S3 *in the generated code*. The substitution is done only for compiler analysis purposes. This new version of S3, referred to as S33, is of the form: $S33 : A(R(i)) = A(K(i)) + constant$. Similarly, S5 becomes $S55 : A(L(i)) = A(K(i)) + constant$. Next, we label the statement pairs (S1, S3) and (S1, S5) in the original loop as *expanded reduction statements* (ERSs). If we treat each ERS as a single reduction statement, then this problem is reduced to one treated above.

The code generated for the run time marking of the ERS is inserted for both sides of the statement (RHS and LHS), *but only in the same basic block as the LHS*. As we will see in a later example, this rule insures that both sides are marked when and if there is an assignment, i.e., it insures that a value is actually passed from the RHS to LHS. Any uses of values participating in the reduction that occur outside the ERS invalidate the ERS, i.e., set the corresponding element of the shadow array A_{nx} to true. In the case of ERSs obtained through forward substitution, the value of the reduction reference may pass through several memory locations (intermediate variables) before reaching the statement of the LHS. As any use of an intermediate variable represents a use of a value that participates in the reduction, it invalidates the reduction for the corresponding element of A. The uses can be obtained by following the def–use chain within the scope of the loop. However, based on the dead reference elimination principle described in Section 3.1, only those uses that contribute to the actual data–flow of the loop (when the value is passed on to a shared variable or controls the access to a shared variable) are processed. If not all local variables carrying the reduction value end up being used in the global data–flow within the loop, then we have either to verify that they (the local variables) are indeed not live after loop exit, or, if that is not possible, make a conservative assumption (i.e., that all uses contribute to the data flow). In Fig. 5(a), statement S4 passes the value of $A(K(i))$ to the local variable t, which in turn passes it to $A(L(i))$ in S5. The same value is also passed to the shared variable $B(f(i))$ in S6. Both uses (in S5 and S6)

```
    do i = 1,n
S1:   z = A(K(i))
S2:   y = constant
S3:   A(R(i)) = z + y
S4:   t = z
S5:   A(L(i)) = t + y
S6:   if (exp) B(f(i)) = t
    enddo
```

(a)

```
    doall i = 1,n
S1:   z = A(K(i))
S2:   y = constant
      markread(K(i))
      markwrite(R(i))
      if (K(i) .ne. R(i)) then
        markredux(K(i))
        markredux(R(i))
      endif
S3:   A(R(i)) = z + y
S4:   t = z
      markwrite(L(i))
      if (K(i) .ne. L(i)) then
        markredux(K(i))
        markredux(L(i))
      endif
S5:   A(L(i)) = t + y
S6:   if (exp) then
        markredux(K(i))
        B(f(i)) = t
      endif
    enddo
```

(b)

Figure 5: The code generated for the `do` loop in (a) is shown in (b). The `mark-x` operations are as described in Fig. 3.

should, in principle, invalidate $A_{nx}(K(i))$. On the other hand, statement S5 is another potential reduction of the same type as in S3 and, thus only the use in S6 needs to invalidate $A_{nx}(K(i))$. The transformed code is shown in Fig. 5(b).

We note that if one of the intermediate variables is itself an array element addressed indirectly, then an additional run–time test must be performed. For example, if S1 and S3 in Fig. 5(a) were of the form: $S1 : X(N(i)) = A(K(i))$ and $S3 : A(R(i)) = X(P(i)) + y$, then a value would be passed from S1 to S3 only if $N(i) = P(i)$. However, if the array X is privatizable, and occurs *only* in these two statements, then the run–time test is not necessary, i.e., if $N(i) = P(i)$, then $A(K(i))$ would be processed with the read of $X(P(i))$ in S3, and otherwise no data flow would occur.

Taking control flow into account. The final situation we consider is when the forward substitution procedure must take into account conditional branches and carry information into the expression of the ERS (see Fig. 6). The additional difficulty presented by this case is the fact that the exact form of the RHS is not known statically. What is known, however, is the set of all possible RHS forms, which can be computed by following all potential paths in the control flow graph. A direct approach uses a *gated static single assignment* (GSSA) [5, 33] representation of the program. In such a representation, scalar variables are assigned only once. At the points of confluence of conditional branches a ϕ function of the form $\phi(B, X_1, X_2)$ is used (in the GSSA representation) to select one of the two possible definitions of a variable (X_1 or X_2), depending on the boolean expression B. By proceeding backwards through the def–use chains (which include the ϕ functions) it is easy to expand a scalar variable in terms of boolean expressions, other scalar variables, and array elements. In the example of Fig. 6, the variable w in statement S9 would be expanded as follows:

$$\begin{aligned} w &\Rightarrow \phi(B3, t, A(M(i))) \\ &\Rightarrow \phi(B3, \phi(B2, z, A(J(i)), A(M(i)))) \\ &\Rightarrow \phi(B3, \phi(B2, \phi(B1, A(K(i))), A(L(i)), A(J(i)), A(M(i)))) \end{aligned}$$

which means that the value of w is:

$$w = \begin{cases} A(K(i)) & \text{if } (B3 \wedge B2 \wedge B1) \text{ is true} \\ A(L(i)) & \text{if } (B3 \wedge B2 \wedge \neg B1) \text{ is true} \\ A(J(i)) & \text{if } (B3 \wedge \neg B2) \text{ is true} \\ A(M(i)) & \text{if } (\neg B3) \text{ is true} \end{cases} \quad (1)$$

This compound equation can then be used to generate a `markread` and a `markredux` operation at statement S9 where w is read. To save unnecessary work, we only expand those scalars that are on the RHS of assignments to

```
    do i = 1,n
S1:  w = A(M(i))
S2:  t = A(J(i))
S3:  if (B1) then
S4:   z = A(K(i))
     else
S5:   z = A(L(i))
     endif
S6:  if (B2) t = z
S7:  if (B3) w = t
S8:  if (B5) A(R(i)) = A(R(i)) + z
S9:  if (B6) Y(i) = w
    enddo
```

(a)

```
    doall i = 1,n
S1:  w = A(M(i))
S2:  t = A(J(i))
S3:  if (B1) then
S4:   z = A(K(i))
     else
S5:   z = A(L(i))
     endif
S6:  if (B2) t = z
S7:  if (B3) w = t
S8:  if (B5) then
      markread(B1*K(i) + notB1*L(i))
      markredux(B1*K(i) +  notB1*L(i))
      markwrite(R(i))
      A(R(i)) = A(R(i)) + z
     endif
S9:  if (B6) then
      markread(B3*B2*B1*K(i) + B3*B2*notB1*L(i)
               + B3*notB2*J(i) + notB3*M(i))
      markredux(B3*B2*B1*K(i) + B3*B2*notB1*L(i)
               + B3*notB2*J(i) + notB3*M(i))
      Y(i) = w
     endif
    enddoall
```

(b)

Figure 6: The code generated for the `do` loop in (a) is shown in (b). The `mark-x` operations are as described in Fig. 3. The expressions in the `markread` and `markredux` operations are abreviations of `if then else` statements representing the different assignments to `z` (`S8`) and `w` (`S9`) as in Equation 1. The operators "`*`", "`+`", and "`not`" represent logical "and", "or", and "complement" operators, respectively.

shared variables or in potential reduction statements (e.g., in the case of z in statement `S8`). All other scalar references can be safely ignored. Fig. 6(b) shows the program in Fig. 6(a) after the insertion of the `markread` and `markredux` operations, which are based on the expansion of the scalar variables. The possible drawback of this approach is that the number of potential reductions and the number of terms in the logic expressions generated may be quite large. If this happens, we can gracefully degrade to a more conservative approach: test only some of the expressions of the ERS and invalidate all the rest.

It is important to note that the loop in Fig. 6 exemplifies the type of loop found in the SPICE2G6 program (subroutine LOAD) which can account for 70% of the sequential execution time.

Finally we mention that reductions such as *min*, *max*, *etc.*, would first have to be syntactically pattern matched, and then substituted by the *min* and *max* functions. From this perspective, they are more difficult to recognize than simpler arithmetic reductions. However, after this transformation, our techniques can be applied as described above.

4 Putting it All Together

In the previous sections we described run–time techniques that can be used for the speculative parallelization of loops. These techniques are automatable and a good compiler could easily insert them in the original code. In this section, we give a brief outline of how a compiler might proceed when presented with a `do` loop whose access pattern cannot be statically determined.

1. *At Compile Time.*
   (a) A cost/benefit analysis is performed using both static analysis (based on the asymptotic complexity of the LPRD test given below) and run–time collected statistics to determine whether the loop should be:
      (i) speculatively executed in parallel using the LRPD test,
      (ii) first tested for full parallelism, and then executed appropriately (using an inspector/executor version of the LRPD Test), or
      (iii) executed sequentially.

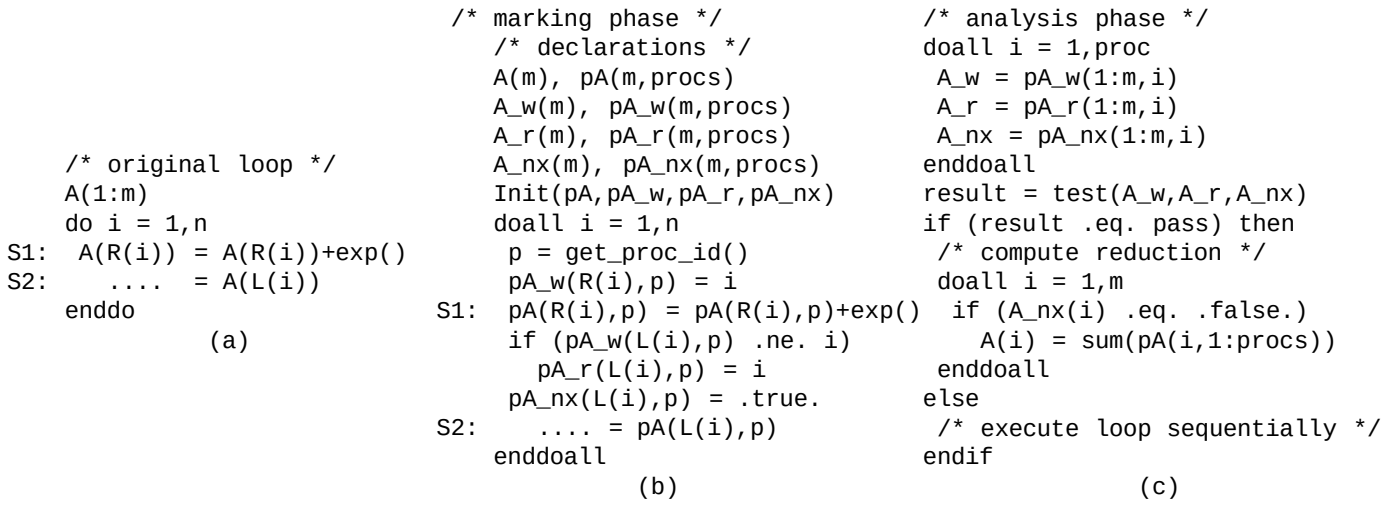


Figure 7: The simplified code generated for the `do` loop in (a) is shown in (b) and (c). Privatization is not being tested because there is a read before a write reference

(b) Generate the code needed for the speculative parallel execution. A parallel version of the original loop is augmented with the `markread`, `markwrite` and `markredux` operations for the LRPD test; if necessary to identify reduction variables, the loop is also augmented as described in Section 3.2.2. In addition, code is generated for: the analysis phase of the LRPD Test, the potential sequential re–execution of the loop, and any necessary checkpointing/restoration of program variables.

2. *At Run–Time.*

(a) Checkpoint if necessary, i.e., save the state of program variables.

(b) Execute the parallel version of the loop, which includes the marking phase of the test.

(c) Execute the analysis phase of the test, which gives the pass/fail result of the test.

(d) If the test passed, then compute the final results of all reduction operations (from the processors' partial results) and copy–out the values of any live private variables. If the test failed, then restore the values of any altered program variables and execute the sequential version of the loop.

(e) Collect statistics for use in future runs, and/or for schedule reuse in this run.

An example using iteration numbers as "marks" in private shadow arrays is shown in Fig. 7. If the speculative execution of the loop passes the analysis phase, then the scalar reduction results are computed by performing a reduction across the processors using the processors' partial results. Otherwise, if the test fails, the loop is re–executed sequentially.

4.1 Complexity of the LRPD test

The time required by the LRPD test is $T(n, s, a, p) = O(na/p + \log p)$, where p is the number of processors, n is the total iteration count of the loop, s is the number of elements in the shared array, and a is the (maximum) number of accesses to the shared array in a single iteration of the loop. We assume that the implementation of the test uses private shadow structures. The analysis below is valid for all variants of the LRPD test.

The marking phase (Step 1) takes $O(na/p + s + \log p)$ time, i.e., proportional to $\max(na/p, s, \log p)$ time. We record the read and write accesses, and the reduction and privatization flags in private shadow arrays using iteration number "marks". In order to check whether for a read of an element there is a write in the same iteration, we simply check that element in the shadow array – a constant time operation. All accesses can be processed in $O(na/p)$ time, since each processor will be responsible for $O(na/p)$ accesses. After all accesses have been marked in private storage, the private shadow arrays can be merged into the global shadow arrays in $O(s + \log p)$ time; the $\log p$ contribution arises from the possible write conflicts in global storage that could be resolved using software or hardware combining. The counting in Step 2(a) can be done in parallel by giving each processor s/p values to add within its private memory,

and then summing the p resulting values in global storage, which takes $O(s/p + \log p)$ time [18]. The comparisons in Step 2(b) (2(d)) of A_w with A_r (with A_{np} and A_{nx}) take $O(s/p + \log p)$ time.

If the loop passes the test, then the final result of each reduction must be computed (unless the reduction was parallelized using unordered critical sections) and last value assignments must be performed for the live private variables. If the reduction operation is parallelized using unordered critical sections, then no overhead is incurred, i.e., the original sequential reduction operation and its transformed parallel version require the same number of operations (within a small constant factor). However, if the reduction is parallelized using recursive doubling, then an overhead $O(s + \log p)$ is incurred when the processors' partial results are merged pair–wise into the scalar reduction results. Similarly, the private variables with the latest time stamps (iteration number "marks") can be selected for last value assignment in time $O(s + \log p)$.

Hash tables. If $s \gg na/p$, then the number of operations in the LRPD test does not scale since each processor must always inspect every element of its private shadow structure when transferring it to the global shadow structure (even though each processor is responsible for fewer accesses as the number of processors increases). Another related issue is that the resource consumption (memory) would not scale. However, if "shadow" hash tables are used, then each processor will only have private shadow copies of the array *elements* accessed in iterations assigned to it, which will increase the cost per access by a small constant factor. Thus, if hash tables of size $O(na/p)$ are used, then the complexity of the marking phase becomes $O(na/p + \log p)$. Similarly, using hash tables the analysis phase and any needed last value assignments and/or processor–wise reduction operations can be performed in time $O(na/p + \log p)$.

5 Experimental Results

In this section we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [3]) and 14 processors (Alliant FX/2800 [4]) using a Fortran implementation of our methods. However, we remark that our results scale with the number of processors and the data size and thus they should be extrapolated for massively parallel processors (MPPs), the actual target of our run–time methods.

We considered seven `do` loops from the PERFECT Benchmarks [7] that could not be parallelized by any compiler available to us. Our results are summarized in Table 1. For each loop, we note the type of test applied: *doall* indicates cross–iteration dependences were checked (Lazy Doall (LD) test), *privat* indicates privatization was checked (LPD test), *reduct* indicates reduction parallelization was checked (LRD test). For each method applied to a loop, we give the speedup that was obtained, and the potential slowdown that *would have been incurred* if, after applying the method, the loop had to be re–executed sequentially. If the inspector/executor version of the LRPD test was applied, the computation performed by the inspector is shown in the table: the notation *privatization* indicates the inspector verified that the shared array was privatizable and then dynamically privatized the array for the parallel execution, *branch predicate* and *subscript array* mean that the inspector computed these values, and *replicates loop* means that the inspector was work–equivalent to the original loop.

In addition to the summary of results given in Table 1, we show in Figures 8 through 14 the speedup and the *potential* slowdown measured for each loop as a function of the number of processors used. For reference, these graphs show the ideal speedup, which was calculated using an optimally parallelized (by hand) version of the loop. The potential slowdown reported is the percentage of the execution time that would be paid as a penalty if the test had failed, and the loop was then executed sequentially. In cases where extraction of a reduced inspector loop was impractical because of complex control flow and/or inter–procedural problems, we only applied the speculative methods.

Whenever necessary in the speculative executions, we performed a simple preventive backup of the variables potentially written in the loop. In some cases, the cost of saving/restoring might be significantly reduced by using another strategy. In order for our methods to scale with the number of processors, the shadow arrays must be distributed over the processor space, rather than replicated on each processor (Section 4.1). For this purpose, we tried using hash tables. Since we had at most 14 processors, the extra cost of the hash accesses dominated the benefit of reducing the size of the shadow arrays. This was particularly true for the loops from the OCEAN and TRFD Benchmarks. However, on a larger machine we would expect the use of hash tables to pay off. Due to this problem, the results reported do not reflect the use of hash tables.

The graphs show that in most cases the speedups scale with the number of processors and are a very significant percentage of the ideal speedup. When they do not scale, as mentioned above, we believe that the use of hash tables

[2] All benchmarks are from the PERFECT Benchmark Suite

[3] The final paper will include experimental results for all loops on both machines.

Benchmark[2] Subroutine Loop	Experimental Results			Tested	Description of Loop	Inspector (computation)
	Technique	Speedup	*potential* Slowdown			
MDG	14 processors					privatization
INTERF	speculative	11.55	1.09	doall	accesses to a privatizable vector guarded	data accesses
loop 1000	insp/exec	8.77	1.03	privat	by loop computed predicates	branch predicate
BDNA	14 processors					privatization
ACTFOR	speculative	10.65	1.09	doall	accesses privatizable array indexed by a	data accesses
loop 240	insp/exec	7.72	1.04	privat	subscript array computed inside loop	subscript array
	8 processors					
TRFD	speculative	.85	2.17	doall	small triangular loop accesses a vector	data accesses
INTGRL	sched reuse	1.93	2.17		indexed by a subscript array computed	replicates loop
loop 540	insp/exec	1.05	1.74		outside loop	
	sched reuse	2.10	1.74			
TRACK					accesses array indexed by subscript array	
NLFILT	8 processors			doall	computed outside loop, access pattern	not applicable
loop 300	speculative	4.21	1.01		guarded by loop computed predicates	
ADM					accesses privatizable array thru aliases,	
RUN	14 processors			doall	array repeatedly re–dimensioned, access	not applicable
loop 20	speculative	9.01	1.02	privat	pattern guarded by loop computed predicates	
OCEAN	8 processors				kernel–like loop accesses a vector with	data accesses
FTRVMT	speculative	2.23	1.45	doall	run–time determined strides	replicates loop
loop 109	insp/exec	2.14	1.30		26K invocations account for 40% T_{seq}	
SPICE					traverses linked list terminated by a NULL	
LOAD	8 processors			doall	pointer, all referenced arrays equivalenced	data accesses
loop 40	insp/exec	2.75	1.09	reduct	to a global work array	

Table 1: Summary of Experimental Results.

(for MPPs) will preserve the scalability of our methods. We note that with the exception of the TRFD loop (Fig. 10), the speculative strategy gives superior speedups versus the inspector/executor method. For both methods the potential slowdown is small, and decreases as the number of processors increases. As expected, the potential slowdown is smaller for the inspector/executor method.

We now make a few remarks about individual loops for which Table 1 does not give complete information.

The loop from TRACK is parallel for only 90% of its invocations. In the cases when the test failed, we restored state, and re–executed the loop sequentially. The speedup reported includes both the parallel and sequential instantiations (Fig. 11).

Loop 40 from SPICE is representative of the type of the loop contained in the LOAD subroutine, which accounts for 70% of the sequential execution time. Since all the arrays are equivalenced to a global work array, all accesses in the loop were shadowed in the LRD test, i.e., each array element was proven to be either a reduction variable, read–only, or independent (i.e., accessed in only one iteration). For this loop we used an inspector/executor version of the LRD test (instead of a speculative parallelization) because of complex memory management problems for the shadow arrays in the presence of highly irregular and sparse access patterns. The ideal speedup of loop 40 is not very large since the loop is small, imbalanced between iterations, and traverses a linked list. The linked list traversal was parallelized using techniques we developed for automatically parallelizing `while` loops [25]. Thus, although the obtained speedup is modest, it represents a significant fraction of the ideal speedup (see Fig. 14). Therefore, since loop 40 is one of the smallest loops in the LOAD subroutine, we expect to obtain better speedups on the larger loops (since they have larger ideal speedups). In the camera–ready version of the paper, we will report the speedups obtained on all loops in subroutine LOAD.

The speedups obtained for the loops from both OCEAN and TRFD are modest because they are kernels. In the case of the loop from TRFD we were able to reuse the schedule and improve our results significantly. Because of the large data set accessed, the loop from TRFD is the only case in which speculative execution proved to be inferior to the inspector/executor method (saving state was a significant portion of the execution time).

6 Conclusion

In this paper we have approached the problem of parallelizing statically intractable loops at run–time from a new perspective – instead of determining a valid parallel execution schedule for the loop, we speculate that the loop is fully parallelizable, a frequent occurrence in real programs. We proposed efficient, scalable run–time techniques for verifying the correctness of a speculative parallel execution, i.e., methods for checking that there were in fact no cross–iteration dependences in the loop. From our previous experience with static analysis and parallelization of Fortran programs, we have found that the two transformations most effective in removing data dependences are privatization and reduction parallelization. Thus, our new run–time techniques for checking the validity of speculative applications of these transformations increases our chance of extracting a significant fraction of the available parallelism in even the most complex program. The methods in this paper employ a dependence analysis based on the actual exchange (definition or use) of values rather than on the memory references themselves. This approach leads to the exploitation of more parallelism than was previously possibly, e.g., our general method for reduction recognition that does not rely on syntactic pattern matching.

Our experimental results show that the concept of run–time data dependence checking is a useful solution for loops that cannot be analyzed sufficiently by a compiler. Both speculative and inspector/executor strategies have been shown to be viable alternatives for even modestly parallel machines like the Alliant FX/80 and 2800. We would like to emphasize that our methods are applicable to all loops, without any restrictions on their data or control flow.

We believe that the significance of the methods presented here will only increase with the advent of massively parallel processors (MPPs) for which the penalty of not parallelizing a loop could be a massive performance degradation. As we have shown, our run–time tests are efficient and scalable, and thus if the target machine has many (hundreds) processors, then the cost of our techniques will become a very small fraction of the sequential execution time. In other words, speculating that the loop is fully parallel has the potential to offer large gains in performance (speedup), while at the same time risking only small losses. To bias the results even more in our favor, the decision on when to apply the methods should make use of run–time collected information about the fully parallel/not parallel nature of the loop. In addition, specialized hardware features could greatly reduce the overhead introduced by the methods.

Finally we believe that the true importance of this work is that it breaks the barrier at which automatic parallelization had stopped: regular, well–behaved programs. We think that the use of aggressive, dynamic techniques can extract most of the available parallelism from even the most complex programs, making parallel computing attractive.

Acknowledgment

We would like to thank Paul Petersen for his useful advice, and William Blume and Gung-Chung Yang for identifying and clarifying applications for our experiments. Special thanks go to Nancy Amato for her careful review of the manuscript and insightful comments.

References

[1] S. Abraham. Private communication, 1994.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] Alliant Computer Systems Corporation, 42 Nagog Park, Acton, Massachusetts 01720. *FX/Series Architecture Manual*, 1986. Part Number: 300-00001-B.

[4] Alliant Computers Systems Corporation. *Alliant FX/2800 Series System Description*, 1991.

[5] R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: a Representation Supporting Control- Data- and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[6] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA., 1988.

[7] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[8] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. Interim Report 90-13, ICASE, 1990.

[9] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks^{TM} Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[10] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[11] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Comm. ACM*, 37(4):31–41, April 1994.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proc. ACM ???*, pages 1–10, 1990.

[13] 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.

[14] V. Krothapalli and P. Sadayappan. An approach to synchronization of parallel computing. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 573–581, June 1988.

[15] C. Kruskal. Efficient parallel algorithms for graph problems. August 1985.

[16] C. Kruskal. Efficient parallel algorithms for graph problems. pages 869–876, August 1986.

[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

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

[19] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *4th PPOPP*, pages 83–91, May 1993.

[20] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 313–322, 1992.

[21] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proceedings 5th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[22] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. Comput.*, C-36(12):1485–1495, 1987.

[23] I. Nudler and L. Rudolph. Tools for the efficient developement of efficient parallel programs. In *Proc. 1st Israeli Conference on Computer System Engineering*, 1988.

[24] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, December 1986.

[25] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 33–43, July 1994.

[26] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 International Conference on Parallel Processing*, pages 174–178. CRC Press, Inc., 1991. Vol. II - Software.

[27] J. Saltz, R. Mirchandaney, and K. Crowley. The doconsider loop. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 29–40, June 1989.

[28] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5), May 1991.

[29] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 285–297, Portland, Oregon, 1989.

[30] P. Tu and D. Padua. Array privatization for shared and distributed memory machines. In *Proceedings 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, September 1992.

[31] P. Tu and D. Padua. Automatic array privatization. In *Proceedings 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

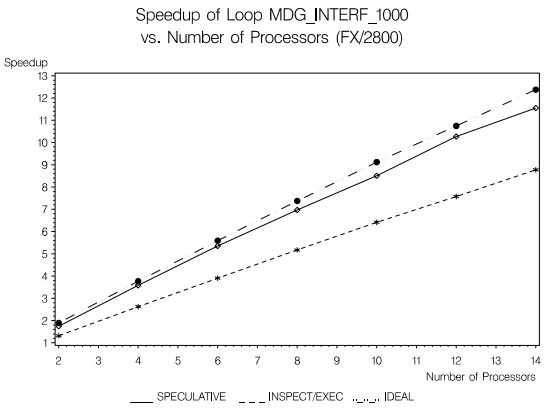
[32] Peng Tu and David Padua. GSA based demand-driven symbolic analysis. Technical Report 1339, University of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, February 1994.

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

[34] 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.

[35] C. Zhu and P. C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13(6):726–739, June 1987.

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



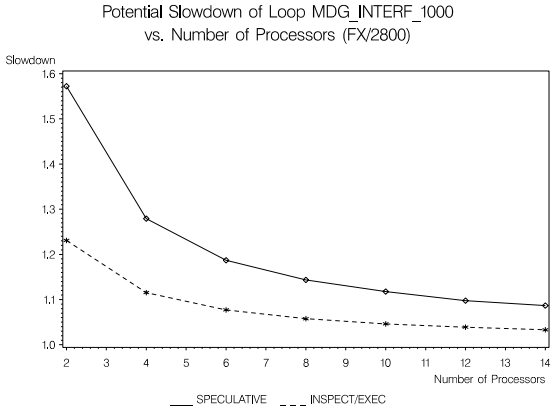
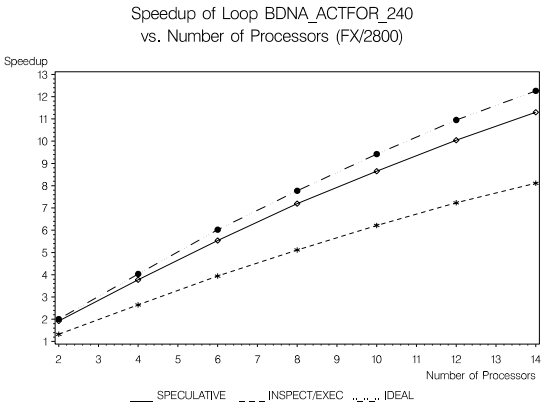


Figure 8:



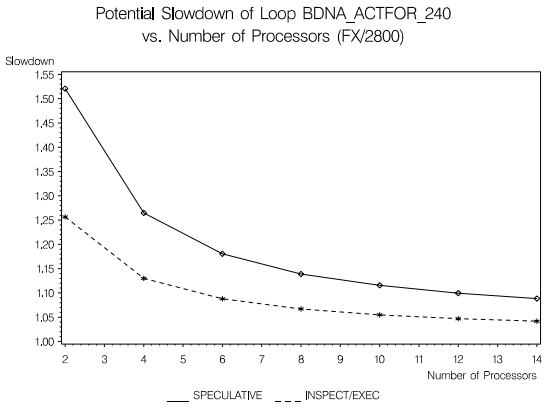
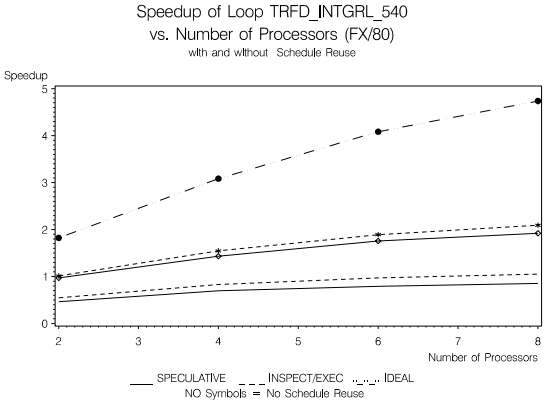


Figure 9:



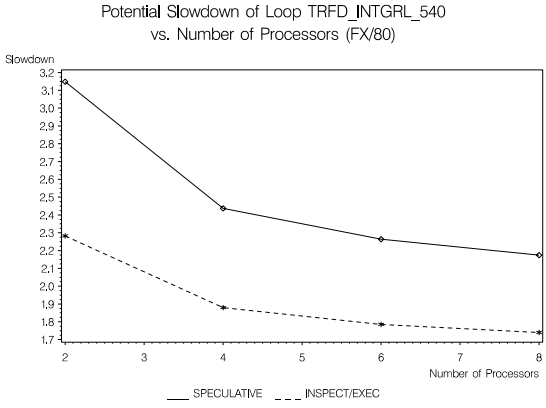
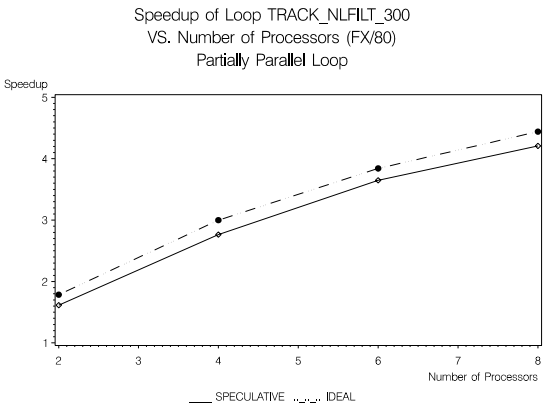


Figure 10:



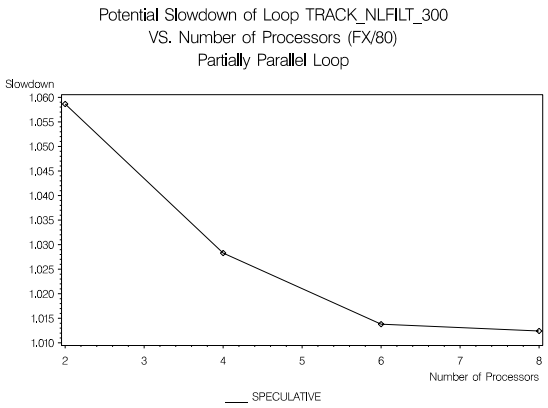
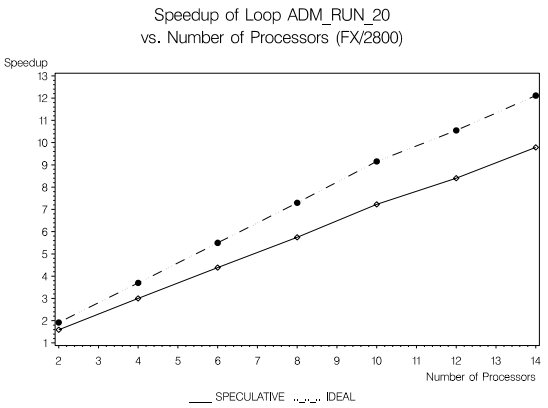


Figure 11:



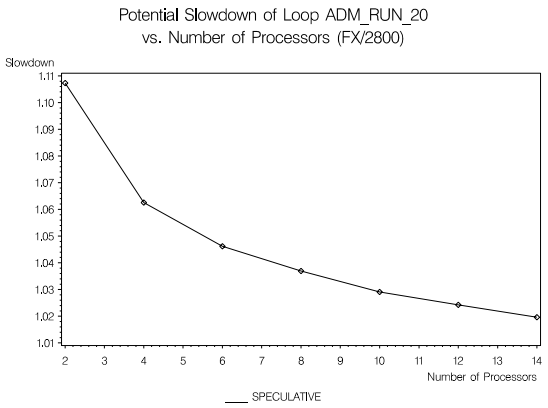
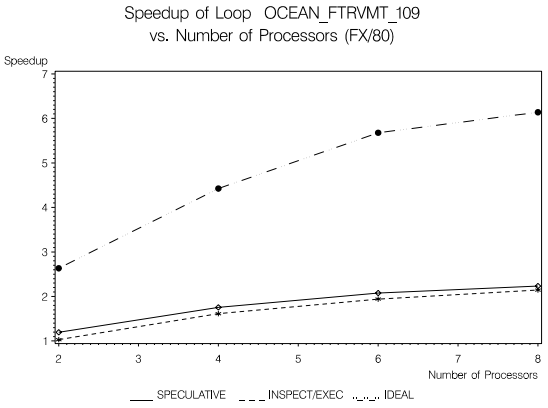


Figure 12:



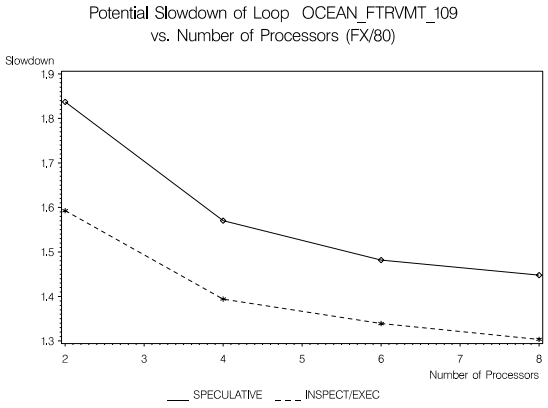
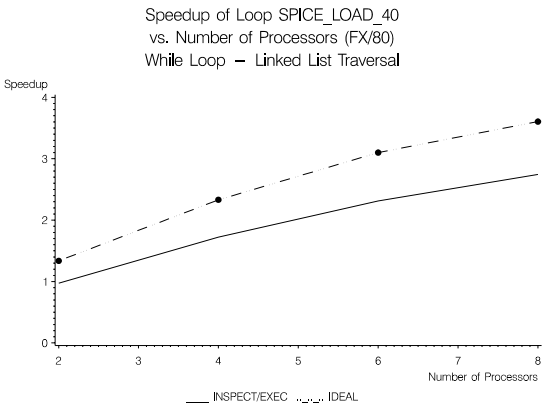


Figure 13:


Speedup of Loop SPICE_LOAD_40
vs. Number of Processors (FX/80)
While Loop – Linked List Traversal
Speedup
0
1
2
3
4
2
3
4
5
6
7
8
Number of Processors
INSPECT/EXEC
IDEAL

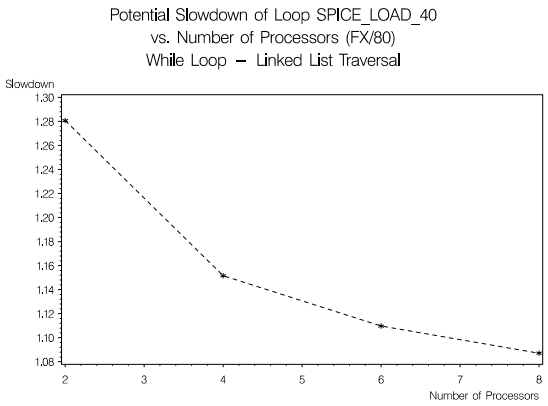

Potential Slowdown of Loop SPICE_LOAD_40
vs. Number of Processors (FX/80)
While Loop – Linked List Traversal
Slowdown
1.08
1.10
1.12
1.14
1.16
1.18
1.20
1.22
1.24
1.26
1.28
1.30
2
3
4
5
6
7
8
Number of Processors

Figure 14: