# Speculative Run-Time Parallelization of Loops[†]

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.
email: rwerger@csrd.uiuc.edu telephone: (217) 333-6578, fax: (217) 244-1351.

**Abstract**

Current parallelizing compilers cannot identify a significant fraction of fully parallel loops because they have complex or statically insufficiently defined access patterns. Since fully parallel loops arise frequently in practice, we have developed methods to speculatively execute loops concurrently. These methods can be applied to any loop, even if the iteration space of the loop is unknown, as in WHILE Loops or DO Loops with conditional exits. To verify the validity of our speculation, we have devised a fully parallel run-time technique for detecting the presence of cross-iteration data dependences in loops. This technique can also be used to eliminate some memory-related dependences by dynamically privatizing scalars and arrays. We outline a cost/performance analysis that can be performed to decide when the methods should be used. Our conclusion is that they should almost always be applied – because, as we show, the expected speedup for fully parallel loops is significant, and the cost of a failed speculation (a not fully parallel loop), is minimal. We present experimental results on loops from the PERFECT Benchmarks which substantiate our conclusion that these techniques can yield significant speedups.

[†]Research supported in part by 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

During the last two decades, compiler techniques for the automatic detection of parallelism have been studied extensively [29, 18, 7]. From this work it has become clear that, for a class of programs, compile-time analysis must be complemented with run-time techniques if a significant fraction of the implicit parallelism is to be detected. The main reason for this is that the access pattern of some programs cannot be determined statically, either because of limitations of the current analysis algorithms or because 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. Compilers usually also conservatively assume data dependences in the presence of subscripted subscripts. More powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically-known values. However, nothing can be done at compile-time when the index arrays are a function of the input data [31, 21, 13]. Even when the access pattern can be analyzed, if the the iteration space is statically unknown (WHILE loops), then compilers have so far not been able to generate parallel code.

Run-time techniques have been used practically from the beginning of parallel computing. Also, during the 1960s, relatively simple forms of run-time techniques, used to detect parallelism between scalar operations, were implemented in the hardware of the CDC 6600 and the IBM 360/91 [23, 25]. Some of today's parallelizing compilers postpone part of the analysis to run-time by generating two-version loops. These consist of an `if` statement that selects either the original serial loop or its parallel version. The boolean expression in the `if` statement typically tests the value of a scalar variable. During the last few years, new techniques have been developed for the run-time analysis and scheduling of loops with cross-iteration dependences [30, 19, 13, 20, 21, 31, 17, 5].

The behavior of programs is usually unknown at compile time. For example, incomplete foreknowledge of the outcome of branches and of the memory space access pattern prevent the compiler from adopting transformations such as locality enhancement. The strategy of speculatively making assumptions about the dynamic control flow or about data locality for prefetching instructions and/or data [9] has been used for some time with success. Recently, branch speculation has been used effectively in superscalar compilers [15, 22, 24].

In this paper we make assumptions about the parallelism of loops, i.e., we assume a loop is fully parallel, speculatively execute it concurrently, and then 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. Our interest in fully parallel loops is motivated by the fact that they arise frequently in real programs. As we show, the analysis needed to test whether a loop is fully parallel can be done very efficiently at run-time. The techniques presented are also capable of eliminating some memory-related dependences by dynamically privatizing scalars and arrays. When the iteration space of the loop is unknown, as in WHILE Loops or DO Loops with conditional exits, we show that the loops can be speculatively executed concurrently,

and then later, the effects of any iterations that overshot the termination condition can be undone. This technique can even be used to obtain significant speedups for loops involving linked list traversals. User directives or execution statistics can be used to identify the loops to which these methods should be applied to decrease the probability of unsuccessful speculation.

In Section 3, we describe methods for speculatively executing DO Loops in parallel. We also describe how these techniques can be used in an inspector/executor setup, i.e., code preceding the loop (the inspector) performs the run-time test for full parallelism, and decides if the the loop should be executed sequentially or in parallel. In Section 4, methods are given that can be used for the parallel execution of loops with unknown iteration space. In Section 5, we discuss some important compile-time issues. Finally, in Section 6, 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 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 [18, 11, 3, 29, 32]. 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.

```
S1:  DO i = 1, n
S2:     A[i] = 2*A[i]
S3:  ENDDO
```

(a)

```
S1:  DO i = 1, n/2
S2:     tmp = A[2*i]
S3:     A[2*i] = A[2*i-1]
S4:     A[2*i-1] = tmp
S5:  ENDDO
```

(b)

```
S1:  DO i = 2, n
S2:     A[i] = A[i] + A[i-1]
S3:  ENDDO
```

(c)

Figure 1:

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. The iterations of such a loop are not independent because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration of the loop. For example, the iterations of the loop in Fig. 1(c),

which computes the prefix sums for the array $A$, must be executed in order of iteration number because iteration $i+1$ needs the value that is produced in iteration $i$, for $2 \leq i \leq n$. In principle, if there are no flow dependences between the iterations of a loop, then the loop may be 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 in parallel. For example, there are no cross-iteration dependences in the loop shown in Fig. 1(a), since iteration $i$ only accesses the data in $A[i]$, for $1 \leq i \leq n$. 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. Unfortunately, not all such situations can be handled efficiently. In order to remove certain types of memory-related dependences a transformation called *privatization* can be applied to the loop. 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., [6, 14, 16, 26, 27]). The loop shown in Fig. 1(b), which, for even values of $i$, swaps $A[i]$ with $A[i-1]$, is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S4` 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 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.

# 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. The dependences between the statements referencing the shared array may be difficult and/or impossible for the compiler to analyze for a number of reasons: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, nonlinear subscript expressions (a fairly rare case)

and, most frequently, subscripted subscripts. Instead of executing the loop sequentially, the compiler could decide to speculatively execute the loop as a DOALL, and generate code to determine at run-time whether the loop was in fact fully parallel. In addition, if it is suspected that some memory-related dependences could be removed by privatization, then the compiler may further elect to privatize those arrays that need it in the speculatively executed DOALL loop. If the subsequent 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 checkpointing/restoration mechanism:* to save the original values of 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.

There are several ways to maintain backups of the values 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 the best solution will likely be to checkpoint prior to the speculative execution. It might be possible to reduce the cost of checkpointing by identifying and checkpointing a point of minimum state in the program prior to the speculative parallel execution. If it is known that the access pattern of the shared array is sparse, then it may be preferable to privatize the array, and copy-in any needed external values, and copy-out any live values if the test passes. Note that privatized arrays need not be backed up because the original version of the array will not be altered during the parallel execution.

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 like an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, restore the values of any altered program variables, and execute the loop sequentially. Below, we present a technique that can be used to detect the presence of cross-iteration dependences in the loop. Briefly, the test traverses shadow arrays using the access pattern of the original shared arrays, and performs some final analysis to determine whether there were cross-iteration dependences in the loop.

There are a number of factors that could influence the compiler's decision to use speculative parallelization. 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 propose a credible compiler strategy for determining when to use speculative parallel execution, we must first discuss the technique for detecting cross-iteration dependences in the loop. Therefore, in the remainder of this section we focus on this issue, and return to a discussion of possible compiler strategies in Section 5.

### 3.1 The PRIVATIZING DOALL Test

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 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.

The most general version of the test, as applied to a shared array $A$, is given below, i.e., it tests for all types of dependences, and assumes that the shared array $A$ was privatized for the speculative parallel execution. Later, we describe how the test can be simplified for instances in which we are only testing for certain types of dependences (e.g., output), or if the array $A$ was not privatized. In general, if it cannot be established that the shared array is *not* privatizable, then run-time privatization should probably be attempted.

**PRIVATIZING DOALL Test (PD 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 can NOT be validly privatized. Initially, the test assumes that all array elements *are* privatizable, and if it is found in any iteration that an element is read before it is written, then it will be marked as non privatizable. The shadow arrays $A_r$, $A_w$, and $A_{np}$ are initialized to zero.

   During each iteration of the loop, all accesses to the shared array $A$ are processed:

   (a) Writes: If this is the first write to this array element in this iteration, then set the corresponding element in $A_w$.

   (b) Reads: If this array element is never written in this iteration, then set the corresponding element in $A_r$. If this array element has not been written in this iteration before this 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 termination of the speculative parallel execution of the loop.) For each shared array $A$ under scrutiny:

   (a) Compute (i) $tw(A) = \sum tw_i(A)$, i.e., the total number of 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 $any(A_w[:] \wedge A_r[:])$,[1] i.e., if the marked areas are common *anywhere*, then the loop IS NOT a DOALL and the phase ends. (Since we read and write from 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 read before it was been 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.)

In order to illustrate the PRIVATIZING DOALL (PD) Test we consider the loop shown in Fig. 2, which contains memory-related dependences that can be removed by privatization. Assume the loop has 8 iterations, accesses a vector of dimension 12, and that the access pattern is given by the subscript arrays $R1, R2$ and $W$. After marking and counting we obtain the results depicted in Table 1. Since $A_w[:] \wedge A_r[:]$ and $A_w[:] \wedge A_{np}[:]$ are zero everywhere, the loop can be made into a DOALL, but only after privatization since $tw(A) \neq tm(A)$.

```
S1:  DO i = 1, n                          R1[1:8] = [ 2 2 2 10 8 8 8 10]
S2:     ...       = A[R1[i]]               W[1:8] = [ 1 3 5  4 7 3 6 12]
S3:     A[W[i]]   = ...                   R2[1:8] = [ 1 3 2 10 7 3 8 12]
S4:     ...       = A[R2[i]]
S5:  ENDDO
```

Figure 2:

| | Position in shadow arrays | | | | | | | | | | | | Written | Counted |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | $tw(A)$ | $tm(A)$ |
| $A_w[1:12]$ | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 8 | 7 |
| $A_r[1:12]$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | |
| $A_{np}[1:12]$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | |
| $A_w[:] \wedge A_r[:]$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| $A_w[:] \wedge A_{np}[:]$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |

Table 1: PRIVATIZING DOALL Test - PASSED

### 3.1.1 Simplified Versions of the Test

Under certain circumstances, the PD Test can be simplified. For example, if the shared array $A$ under test is not privatized for the speculative parallel execution, then all operations relating to the shadow array $A_{np}$ can be omitted. In this case, parts (d) and (e) of Step 2 would be omitted, and if the loop was not classified by Step 2 (b) or (c), then there must exist some output dependences between the iterations of the loop (i.e.,

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

$tw(A) \neq tm(A)$) and the loop is NOT a DOALL. Similarly, if it is known that the only possible dependences between iterations of the loop are output dependences, then only the operations relating to $A_w$ would be required, and the determination of whether or not the loop was a DOALL would be made by Step 2(c).

### 3.1.2 Complexity of the PRIVATIZING DOALL Test

Let $p$ be the number of processors, $n$ the total iteration count of the loop, $s$ the number of elements in the shared array, and $a$ the (maximum) number of accesses to the shared array in a single iteration of the loop. As explained below, the time required by the PD Test is $T(n, s, a, p) = O(na/p + \log p)$.

The marking phase (Step 1) takes time $O(na/p + \log p)$, i.e., time proportional to $\max(na/p, \log p)$. We record the read and write accesses, and the privatization flags in private shadow arrays. 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. The private shadow arrays can be merged in the global shadow arrays in $O(na/p + \log p)$ time; the $\log p$ contribution arises from the possible write conflicts in global storage that could be resolved using software or hardware combining. If $s > na/p$, then the time required to merge the private shadow arrays into the global shadow arrays may dominate the time required for the actual marking. This can be avoided by using private hash tables of size $O(na/p)$ instead of the private shadow arrays. Note that we minimize communication, since everything except the final merge step is done in private storage.

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; this takes $O(s/p + \log p)$ time [12]. The comparisons in Step 2(b) and 2(d) of the shadow arrays will take at most $O(s/p + \log p)$ time. If $s > na$, then the complexity can be reduced to $O(na/p + \log p)$ by using hash tables.

If a private variable is live after the loop terminates, then we will also need to perform a last value assignment. In this case, we can keep time stamps (iteration numbers) with the private variables, and after the termination of the loop, the private variable with the latest time stamp is copied to the original version of the variable. The private variables with the latest time stamp can be selected in time $O(na/p + \log p)$.

### 3.1.3 Schedule Reuse

Thus far, our analysis has assumed that the PRIVATIZING DOALL Test must be run *each* time a loop is executed in order to determine if that loop is parallel. However, if the loop is executed again, with the same data access pattern, the first test can be reused amortizing the cost of the test over all invocations. This is a simple illustration of the *schedule reuse* technique, in which a correct execution schedule is determined once, and subsequently reused if all of the defining conditions remain invariant (see, e.g., [21]). If it can be determined at compile time that the data access pattern is invariant across different executions of the same loop, then no additional computation is required. Otherwise, this condition must be checked, e.g., for

subscripted subscripts the old and the new subscript arrays can be compared.

### 3.1.4 Inspector/Executor Use of the PRIVATIZING DOALL Test

The PD Test presented above can also be used to detect the parallelism of a loop at run-time *before* executing it. In this scenario, an inspector loop would be formed by collecting the accesses to the shared variables under study into a sub-loop, and replacing them with accesses to the appropriate shadow variables. Then, the outcome of the analysis phase would determine whether the original loop should be executed sequentially or in parallel. The complexity analysis for the inspector/executor version of the test is the same as that given above for the speculative version. As will be discussed in Section 5, there are certain cases when the inspector/executor strategy may be preferable to speculatively executing the loop in parallel.

Note that if privatization is used for speculative parallel execution, then the entire shared array will be privatized. A feature of the inspector/executor strategy is that the PD Test can be used to identify the individual array *elements* that should be privatized, i.e., only the array elements that are written during the loop. It is possible that selectively privatizing array elements could lead to super-linear speedups due to a reduction in the size of the working set (relative to the current naive privatization techniques that duplicate the entire array in all processors). A detailed discussion of how the PD Test can be used to selectively privatize individual array elements can be found in [**?**].

# 4 Speculative Parallel Execution of WHILE Loops

WHILE Loops have often been treated by compilers as sequential constructs because their iteration space is unknown [28]. A related case which is generally also handled sequentially by compilers is the DO Loop with a conditional exit. In this section we propose techniques that can be used to execute such loops in parallel.

In order to focus on the problem of dealing with an unknown number of iterations, we first consider *WHILE Loops that are known to have no cross-iteration dependences* except for those necessary to control the loop. In this case, a WHILE Loop can be considered as a sequence of independent iterations ordered by some underlying recursion, which will be called the *dispatching recursion*, or simply the *dispatcher*. If the dispatcher has the simpler form of an an induction, then each point in the dispatcher's domain can be independently and concurrently evaluated using the closed form solution of the induction. In this case, all iterations of the WHILE Loop can be executed simultaneously since aside from the dispatching recursion we assumed no other dependences. An example of a recursive dispatcher is a pointer used to traverse a linked list; since the values of the dispatcher (the pointer) must be evaluated in sequential order, all the iterations of the loop cannot be initiated simultaneously. An example of a dispatcher with a closed form solution is a DO Loop; since all the values of the loop induction variable can be independently evaluated, all the iterations can be initiated simultaneously.

Another difficulty with parallelizing a WHILE Loop is that the termination condition of the loop may be

*overshot*, i.e., iterations could be executed that would not be executed by the sequential version of the loop. The termination condition is *loop invariant* if it is only dependent on the dispatcher and values that are computed outside the loop, and otherwise it is *loop variant*, i.e., it depends on some value computed in the loop. If the termination condition is loop variant, then iterations that larger the last valid iteration could be performed, i.e., iteration $i$ cannot decide if the termination condition is satisfied in the loop body of some iteration $i' < i$. Overshooting may also occur if the dispatcher is an induction and the termination condition is loop invariant. An exception in which overshooting would not occur is if the dispatcher is a monotonic function, and the termination condition is a threshold, e.g., $d(i) = i^2$, and $tc(i) : (d(i) < V)$, where $V$ is a constant, and $d(j)$ and $tc(j)$ denote the dispatcher and the termination condition, respectively, for the $j$th iteration. Another case in which overshooting can be avoided is when the dispatcher is a recurrence, and the termination condition is loop invariant. For example, the dispatcher *tmp* is a pointer used to traverse a linked list, and the termination condition is $(tmp = null)$.

We now discuss techniques that can be used to automatically transform WHILE Loops for speculative parallel execution. We propose two general methods: a fully parallel method for loops in which the dispatcher is an induction, and a method that is only partially parallel for loops in which the dispatcher is a more complex recurrence. For the case in which the dispatcher is not an induction, our methods assume that the dispatching recurrence is fully determined before loop entry (e.g., if the dispatcher is traversing a linked list, no elements may be inserted or deleted during loop execution). We first describe both methods without addressing the overshooting problem, and later discuss how they can be augmented to "undo" any iterations that overshot the termination condition. Initially, we assume no cross-iteration dependences other than those in the dispatcher, and then we remove this assumption and briefly mention how the methods described in this section can be combined with the run-time techniques for analyzing cross-iteration dependences discussed in Section 3.

## 4.1 The Dispatcher is an Induction

In this section we consider a WHILE Loop in which it is known that there are no cross-iteration dependences, and the dispatcher is an induction. To simplify our discussion, we assume that the dispatcher of the $i$th iteration is $i$, i.e, $d(i) = i$. In addition, we assume that some upper bound $u$ on the number of iterations of the WHILE Loop is known, e.g., a DO Loop with a conditional exit. If this is not the case, then some suitable number $u'$ can be selected and the following technique can be applied first to iterations 1 through $u'$, then to iterations $u' + 1$ through $2u'$, etc., until the termination condition is reached.

In this method, referred to as *Induction-1*, the loop is run as a DOALL and a test of the termination condition of the WHILE Loop is inserted into the loop body (see Fig. 3). During the parallel execution, each processor will keep track of the lowest iteration it executed that met the termination condition. Then, after the DOALL has terminated, the last iteration that would have been executed by the sequential version of the WHILE Loop is found by taking the minimum of the processor-wise minima; this iteration must be

```
*WHILE Loop: induction*
      integer i = 1
      WHILE (f(i))
        work(i)
        i = i + 1
      ENDWHILE
```

```
*WHILE Loop: recurrence*
ptr tmp = head(list)
WHILE ( tmp .ne. null)
     work(tmp)
     tmp = next(tmp)
ENDWHILE
```

```
*Induction-1*
integer It[0:nproc-1] = u
DOALL i = 1,u
  if (f(i)) then
    It[vpn] = min(It[vpn],i)
    QUIT
  endif
  work(i)
ENDDO
LastIter = min(It[1:nproc])
```

```
*Recurrence-1*
ptr tmp = head(list)
DOALL i = 1,u
     ptr pt
     lock(list)
     pt = tmp
     tmp = next(tmp)
     unlock(list)
     if (pt .eq. null) QUIT
     work(pt)
ENDDOALL
```

```
*Recurrence-2*
DOALL i = 1, nproc
     integer j
     ptr pt = head(list)
     do j = 1,vpn
       pt = next(pt)
       if (pt .eq. null) goto 2
     enddo
1    work(tmp)
     do j = 1,nproc
       pt = next(pt)
       if (pt .eq. null) goto 2
     enddo
     goto 1
2    continue
ENDDOALL
```

Figure 3: Parallelizing WHILE Loops. In the DOALLs, *nproc* is the number of processors, $u$ is an upper bound on the number of iterations of the WHILE Loop, and *vpn* is the virtual processor number of the processor executing the iteration.

found so that the iterations that need to be undone can be identified. In order to terminate the parallel loop cleanly before all iterations have been executed, a *QUIT* operation similar to the one on Alliant computers [1] could be used. Once a QUIT command is issued by an iteration, all iterations with loop counters less than the that of the issuing iteration will be completed, but no iterations with larger loop counters will be begun. If multiple QUIT operations are issued, then the iteration with the smallest loop counter will control the exit of the loop. On Alliant computers iterations are issued in order. Therefore, the statement $It[vpn] = min(It[vpn], i)$ can be replaced with $It[vpn] = i$. The complexity of this method is $O(T_{seq}/p)$, where $T_{seq}$ is the sequential execution time of the WHILE Loop, and $p$ is the number of processors.

## 4.2 The Dispatcher is a Recurrence

We now consider a WHILE Loop in which it is known that there are no cross-iteration dependences, and the dispatcher is a recurrence. We propose two techniques that can be used to execute this type of WHILE Loop in parallel. Although these techniques are not fully parallel, they can yield very good speedups – especially if a significant amount of work is performed in the loop body. For simplicity, we describe the methods as applied to a WHILE Loop that traverses a linked list.

One obvious method, referred to as *Recurrence-1*, is to serialize the accesses to the *next*() operation. Another method, *Recurrence-2*, which avoids explicit serialization, is to compute the whole recurrence in each processor and assign to processor $i$ the values of the recurrence which are congruent to $i$ mod *nproc*, where *nproc* is the total number of processors. See Fig. 3 for examples of both methods. The complexity

of both these methods is $O(l + T_{seq}/l + T_{seq}/p)$, where $l$ is the number of iterations in the WHILE Loop, $T_{seq}$ is its sequential execution time, and $p$ is the number of processors. Note that in both *Recurrence-1* and *Recurrence-2*, the entire recurrence is traversed (sequentially). In addition to the fact that *Recurrence-1* explicitly serializes accesses to *next*(), and no such serialization is used in *Recurrence-2*, there are some other differences between the two methods. First, in *Recurrence-1* the recurrence is traversed just once by all processors cooperatively, but in *Recurrence-2* each processor will traverse the entire recurrence. Second, in *Recurrence-1* the values of the recurrence are dynamically allocated to the processors, but in *Recurrence-2*, processor $i$, $0 \leq i < nproc$, is statically assigned values congruent to $i$ mod *nproc*.

In the example of Fig. 3, no overshooting occurs because the termination condition is loop invariant. However, if the termination condition had been loop variant, then overshooting might have occurred and in order to determine which iterations needed to be undone, we would have also needed to find the last valid iteration.

### 4.3 Undoing Iterations that Overshot the Termination Condition

Perhaps the easiest method for "undoing" iterations that overshot the termination condition is to checkpoint prior to executing the DOALL, and to maintain a record of *when* (i.e., iteration number) a memory location is written during the loop. Note that since all iterations of the WHILE Loop are independent, each memory location will be written during at most one iteration of the loop. Then, after the DOALL has terminated and the last valid iteration is known, the work of iterations that have overshot can be undone by restoring the values that were overwritten during these iterations. This solution may require as much as three times the actual memory needed by the original WHILE Loop: one copy for checkpointing, one for the actual loop data, and one for the time-stamps. This increase in memory requirements could degrade the performance of the parallel execution of the WHILE Loop.

Checkpointing could be avoided by privatizing all variables in the loop, copying in any needed values, and copying out only those values that are live after the loop and have time-stamps less than or equal to the last valid iteration.

One simple way to reduce the memory requirements is to *strip mine* the loop. This method would introduce global synchronization points and potentially reduce significantly the amount of obtainable parallelism. A way in which the memory requirements could be reduced without introducing rigid synchronization points is to maintain a *sliding window* of some predetermined size $w$: at any given time, the difference between the minimum iteration $l$ that has not been completely executed and the maximum iteration $h$ that has been, or is currently being, executed is at most $w$. Also, the window size could be dynamically determined: the window size is increased if more memory can be used without degrading performance, and is decreased if less memory should be used to improve performance.

### 4.4 WHILE Loops with Unknown Cross-Iteration Dependences

We now consider WHILE Loops in which the cross-iteration dependences cannot be analyzed at compile-time. The general strategy for such loops will be to combine the techniques described in Section 3 for detecting the presence of cross-iteration dependences with the techniques described above for WHILE Loops which do not contain any cross-iteration dependences.

If it is known that the parallel execution of the WHILE Loop will not overshoot, then the dependence tests can be inserted directly into the DOALL Loops as discussed in Section 3. When overshooting may occur, a simple solution is to assume that there are no cross-iteration dependences, and execute the loop twice. First, the loop is run in parallel to determine the number of iterations (using one of the methods discussed above), and once the number of iterations is known the resulting DO Loop can be speculatively parallelized as described in Section 3.

In order to avoid executing the parallel version of the WHILE Loop twice, the PD Test can be incorporated directly into the WHILE Loop methods. Suppose that some shared array in the WHILE Loop will be privatized and tested using the PD Test, and assume that it is known that the shared array is not live after the loop. In this case, all writes to the shadow arrays used for the PD Test will be time-stamped (just like all other variables), and for each shadow element we will maintain the minimum iteration that marked it. Everything proceeds as before, except that in the analysis phase of the PD Test, those marks in the shadow arrays with minimum time-stamps greater than the last valid iteration will be ignored.

If the shared array under test is live after the loop, then the backup method for the privatized array must be more sophisticated. The reason for this is that it is possible for a private variable to be written in more than one iteration of a valid parallel loop. In order to handle this problem, we can keep a time-stamped trail of all write accesses to the privatized array. If the test passes, the live values need to be copied out; the appropriate value would be the value with the latest time-stamp that was not larger than the last valid iteration number, and could be found in the time-stamped trail of write accesses. In order to reduce the memory requirements, strip mining or the sliding window method discussed in Section 4.3 could be used.

If the termination condition of the WHILE Loop is dependent (data or control) upon a variable with unknown dependences, then special care must be taken. In this situation, the best solution is probably to strip mine the loop, and to run the PD Test on each strip.

## 5 Automatic Application of Speculative Parallel Execution

In the previous sections we have discussed 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 consider some issues involving the compile-time strategy for applying these techniques.

## 5.1 Strategy

The following is a brief outline of how a compiler might proceed when presented with a DO Loop whose access pattern cannot be statically determined; a WHILE Loop would be handled similarly.

1. *At Compile Time.*
   (a) A cost/performance analysis determines whether the loop should be:
      (i) speculatively executed in parallel,
      (ii) first tested for full parallelism, and then executed appropriately (using the inspector/executor version of the PD Test), or
      (iii) executed sequentially.
   (b) Generate the code needed for the speculative parallel execution. In addition to the augmented parallel version of the original loop (see Fig. 4 for an example), this includes code for: the analysis phase of the PD Test, the potential sequential re-execution of the loop, and any necessary checkpointing/restoration of program variables. The analysis can be done by calls to a run-time library.
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 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.

## 5.2 Determining When to Attempt Speculative Parallel Execution

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of techniques for parallelizing loops at run-time will be enhanced if the overhead of these techniques is avoided when it is likely that the loop is not fully parallel. The main factors that the compiler should consider when deciding whether to attempt to parallelize a loop at run-time 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. When selecting the best method for the given situation, the compiler should perform a cost/benefit analysis of the three possible options: speculative parallel execution, the inspector/executor method (see Section 3.1.4), or sequential execution. In order to perform this analysis and to predict the parallelism of the loop, the compiler should use both static analysis and run-time statistics (collected on previous executions of the loop); in addition, directives about the parallelism of the loop might prove useful.

Given a loop $L$, the *ideal speedup*, $Sp_{id}$, of $L$ is the ratio between its sequential execution time $T_{seq}$ and its ideal parallel execution time, $T_{doall}$ (under the hypothesis that it is a fully independent loop) However, if our run-time techniques are applied, the *attainable speedup* will be reduced by the overhead of the methods to $Sp_{spec}$ and $Sp_{i/e}$ for the speculative and inspector/executor strategies, respectively.

$$Sp_{id} = \frac{T_{seq}}{T_{doall}} \qquad Sp_{spec} = \frac{T_{seq}}{T_{mark} + T_{analysis} + T_{save} + T_{doall}} \qquad Sp_{i/e} = \frac{T_{seq}}{T_{imark} + T_{analysis} + T_{doall}}$$

```
      **Original Version**
      DO 540 I=1,NP
            DO 530 J=1,I
               IJ=IA(I)+J
      ....
               DO 520 K=1,I
                  MAXL=K
                  IF(K.EQ.I) MAXL=J
                  DO 510 L=1,MAXL
                     KL=IA(K)+L
            .....
C **DOALL Test checks the writes to X**
                     X(IJ,KL)=....
 510              CONTINUE
 520           CONTINUE
 530        CONTINUE
 540        CONTINUE
```

```
          **Augmented Loop for Marking Phase**
      DOALL  I = 1,NP
C **private variables -- do once per processor**
         integer X_w(:,:), tw_i, IJ, MAXL, J, K, L, KL
      LOOP ** do once per iteration (I) **
         tw_i = 0
         DO J=1,I
            IJ=IA(I)+J
            DO K=1,I
               MAXL=K
               IF(K.EQ.I) MAXL=J
               DO L=1,MAXL
                  KL=IA(K)+L
                  X(IJ,KL)=....
C **mark shadow array X_w (if not already marked)**
                  IF (X_w(IJ ,KL) .NE. I) THEN
                     X_w(IJ,KL) = I
                     tw_i = tw_i + 1
                  ENDIF
               ENDDO
            ENDDO
         ENDDO
         tw(X) = tw_i + tw(X)
      ENDDOALL
```

Figure 4: An example of how the original data accesses are processed using shadow variables during the marking phase of the PD Test. This loop is extracted from loop 540 in subroutine INTGRL of TRFD from the PERFECT Benchmarks. The PD Test is applied to the shared array $X$, and checks for output dependences. In the transformed loop (right), writes are marked in the shadow array `X_w` by iteration number $I$, so that the shadow array can be reused. The number of writes marked is recorded in the private variable `tw_i`.

In the relations above, $T_{mark}$ and $T_{analysis}$ denote the time for the marking and analysis phases of the PD Test, and $T_{save}$ is the time to save state. $T_{imark}$ represents the marking phase of the inspector/executor method. In the worst possible case (e.g., a kernel), all components contributing to the overhead of the methods may be comparable to $T_{doall}$, so that $Sp_{spec} \approx \frac{1}{4} Sp_{id}$ and $Sp_{i/e} \approx \frac{1}{3} Sp_{id}$. Note that for massively parallel processors (MPPs), 25% of the ideal speedup would still be an excellent performance, especially when compared to the alternative of sequential execution.

When selecting between speculative parallel execution and the inspector/executor method, the compiler should to compare the estimated costs of $\delta_{spec} = T_{save} + T_{mark}$ and $\delta_{i/e} = T_{imark}$. If $\delta_{spec} \leq \delta_{i/e}$, then speculative parallel execution should be attempted, and otherwise the inspector/executor method should be used. Note that it is possible that $T_{imark} \gg T_{mark}$ since the marking phase of the inspector/executor must traverse the access pattern of the shared variable under study prior to the parallel execution of the loop. In other words, $T_{mark}$ includes only the operations needed to mark the shadow arrays, but $T_{imark}$ must also include any computation needed to determine the access pattern itself. Another factor that should be considered is that the working set for speculative parallel execution may be larger than for the inspector/executor method because of the potential need to save/restore state.

It is also instructive to examine the *slowdown* incurred by a failed speculative parallel execution and a failed test for the inspector/executor method, i.e., when the loop must be executed sequentially. In these cases, the sequential execution time will be increased by the execution time of the failed parallelization attempt, and a restore operation in the case of the speculative method. Note that since all of these components

are fully parallel, in the worst case they are all proportional to $\frac{1}{p}T_{seq}$, yielding a slowdown for both methods that is also proportional to $\frac{1}{p}T_{seq}$:

$$T_{seq}T_{mark}+T_{analysis}+T_{s/r}+T_{doall} \approx T_{seq}+\frac{5}{p}T_{seq} \qquad T_{seq}+T_{imark}+T_{analysis}+T_{doall} \approx T_{seq}+\frac{3}{p}T_{seq}$$

Therefore, unless it is known a priori with a high degree of confidence that the loop is not parallel, a run-time method of parallelizing the loop should probably be applied, i.e., the potential payoff is worth the risk of slightly increasing the sequential execution time.

For WHILE Loops we must also consider the cost of executing and "undoing" any iterations that overshot the termination condition, so that the attainable speedup would be slightly reduced. However, in the case of a failed PD Test, the slowdown might be reduced by using the number of iterations (found by the test), to transform the WHILE Loop into a DO Loop (which is easier to optimize).

### 5.3 The One Processor/$(n-1)$ Processor Solution

As a final remark, we note a method that can be used to minimize the risks of speculative execution: one processor executes the loop sequentially, and the rest of the processors, speculatively, execute the loop in parallel. Of course, the sequential and the parallel executions would need separate copies of the output data for the loop. As long as the cost of creating these copies is not too great, this technique should maximize the potential gains attainable from parallel execution, while, at the same time, minimizing the costs incurred by failed speculations, i.e., speculations on loops that are, in fact, not parallel.

## 6 Experimental Results

In this section we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [1]) and 14 processors (Alliant FX/2800 [2]) using a Fortran implementation of our methods. It should be pointed out that our results scale with the number of processors and the data size and that they should be extrapolated for MPPs, the actual target of our run-time methods.

We considered six DO Loops and two WHILE Loops that could not be parallelized by any compiler available to us; seven loops are from the PERFECT Benchmarks [4], and one of the WHILE Loops is extracted from MCSPARSE, a parallel version of a sparse matrix solver [8]. Our results are summarized in Table 2. 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. In the graphs, we also show the ideal speedup, which was calculated using an optimally parallelized (by hand) version of the loop. If the inspector/executor version of the PD 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

| Benchmark[2] Subroutine Loop | Experimental Results: Technique | Speedup | *potential* Slowdown | Description of Loop | Inspector (computation) |
|---|---|---|---|---|---|
| **DO Loops** | | | | | |
| MDG | 14 processors | | | | privatization |
| INTERF | speculative | 11.55 | 1.09 | accesses to a privatizable vector guarded | data accesses |
| Loop 1000 | insp/exec | 8.77 | 1.03 | by loop computed predicates | branch predicate |
| BDNA | 14 processors | | | | privatization |
| ACTFOR | speculative | 10.65 | 1.09 | accesses privatizable array indexed by a | data accesses |
| Loop 240 | insp/exec | 7.72 | 1.04 | subscript array computed inside loop | subscript array |
| TRFD | 8 processors | | | | data accesses |
| INTGRL | speculative | .85 | 2.17 | small triangular loop accesses a vector | data accesses |
| Loop 540 | sched reuse | 1.93 | 2.17 | indexed by a subscript array computed | replicates loop |
| | insp/exec | 1.05 | 1.74 | outside loop | |
| | sched reuse | 2.10 | 1.74 | | |
| TRACK | 8 processors | | | accesses array indexed by subscript array | |
| NLFILT | | | | computed outside loop, access pattern | not applicable |
| Loop 300 | speculative | 4.21 | 1.01 | guarded by loop computed predicates | |
| ADM | 14 processors | | | accesses privatizable array thru aliases, | |
| RUN | | | | array repeatedly redimensioned, access | not applicable |
| Loop 20 | speculative | 9.01 | 1.02 | pattern guarded by loop computed predicates | |
| OCEAN | 8 processors | | | kernel-like loop accesses a vector with | data accesses |
| FTRVMT | speculative | 2.23 | 1.45 | run-time determined strides | replicates loop |
| Loop 109 | insp/exec | 2.14 | 1.30 | 26K invocations account for 40% $T_{seq}$ | |
| **WHILE Loops** | | | | | |
| SPICE | 8 processors | | | traverses linked list terminated by a | |
| LOAD | Recurrence-1 (locks) | 4.20 | N/A | NULL pointer, loop counter: recurrence, termination condition: loop invariant | not applicable |
| Loop 40 | Recurrence-2 (no locks) | 4.91 | N/A | | |
| MCSPARSE | 8 processors | | | processes an array, | |
| DFACT | | | | loop counter: induction, | not applicable |
| Loop 500 | Induction-1 | 5.60 | N/A | termination condition: loop variant | |

Table 2: Summary of Experimental Results.

*replicates loop* means that the inspector was work-equivalent to the original loop.

Whenever necessary in the speculative executions, we performed a simple preventive backup of the variables potentially written in the loop as follows: 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 3.1.2). 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, and for this reason in some cases the speedups shown in Figures 5 through 12 do not appear to scale.

[2] All benchmarks are from the PERFECT Benchmark Suite, with the exception of MCSPARSE.
[3] The final paper will include experimental results for all loops on both machines.

For all of the loops studied we have obtained a very good fit of our experimental data to the speedups predicted by the modeling described in Section 5. Our estimates were made using a simple instruction counting model: for loops (vectors), we used the product of the number of instructions and the number of iterations (elements). In addition to the summary of results given in Table 2, we show in Figures 5 through 12 the speedup and the *potential* slowdown measured for each loop as a function of the number of processors used. 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 interprocedural problems, we only applied the speculative methods.

These 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 (for MPPs) will preserve the scalability of our methods. We note that with the exception of the TRFD loop (Fig. 7), 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.

Our results also show that significant speedups can be obtained by parallelizing WHILE Loops using our methods. In particular, even though the loop body from SPICE does little work, we obtained a very good speedup (Fig. 11). Note that although each processor traversed the entire linked list, the Recurrence-2 method outperformed the Recurrence-1 method, in which the processors cooperatively traversed the list (by placing the *next*() operation in a critical section).

We now make a few remarks about individual loops for which Table 2 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. 8. The speedups obtained for the loops from both OCEAN and TRFD are modest, just as predicted by our model, 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).

# 7 Conclusion

In this paper we have approached the problem of parallelizing 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 actually fully parallel, a frequent occurrence in real programs. We have developed efficient run-time techniques for verifying a speculative parallel execution, i.e., to check that there were in fact no cross-iteration dependences in the loop. These techniques can also determine whether privatization of scalars and arrays eliminated all

memory-related dependences. We have also shown that loops with unknown iteration space can be executed in parallel, even for such cases as a loop involving a linked list traversal.

Our experimental results prove that the concept of run-time data dependence checking is a useful solution for loops that cannot be sufficiently analyzed by a compiler. Moreover, lack of knowledge about the iteration space of a loop does not in itself preclude parallelization, or the application of our run-time methods. We would like to re-emphasize that our methods are applicable to all loops, without any restrictions on their data or control flow. 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.

However, we believe that the true significance of these methods will be the increase in real speedup obtainable on massively parallel processors (MPPs). As we have shown, the cost associated with these run-time tests is generally proportional to $\frac{1}{p}T_{seq}$, where $p$ is the number of processors available, and $T_{seq}$ is the sequential execution time. If the target architecture is an MPP with *hundreds* or, in the future *thousands*, of processors, then this cost will become a very small fraction of sequential execution time. When applying our run-time data dependence tests to a loop, our performance gain/loss will range from at least 1/4 of the ideal speedup (which can reach into the hundreds for MPP's) when the test passes, to an additional few percentage points of the sequential execution time if the test fails. 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.

# References


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

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

[3] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer. Boston, MA., 1988.

[4] 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.

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

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

[7] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar Fortran and its Restructuring Compiler. In A. Nicolau D. Gelernter, T. Gross and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 1–23. MIT Press, 1991.

[8] K. Gallivan, B. Marsolf, and H. Wijshoff. A large-grain parallel sparse system solver. In *Proc. Fourth SIAM Conf. on Parallel Proc. for Scient. Comp.*, pages 23–28, Chicago, IL, 1989.

[9] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 354–368, July 1990.

[10] M. Guzzi, D. Padua, J. Hoeflinger, and D. Lawrie. Cedar fortran and other vector and parallel fortran dialects. *Journal of Supercomputing*, 4(1):37–62, March 1990.

[11] 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.

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

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

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

[15] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schl ansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[16] 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.

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

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

[19] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 Int'l. Conf. on Parallel Processing, St. Charles, Illinois, August 12-16*, pages 174–178. CRC Press, Inc., 1991. Vol. II - Software.

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

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

[22] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–354, May 1990.

[23] J. E. Thornton. *Design of a Computer:The Control Data 6600*. Scott, Foresman, Glenview, Illinois, 1971.

[24] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipelined architectures. In *Supercomputing*, November 1990.

[25] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[26] 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.

[27] P. Tu and D. Padua. Automatic array privatization. In *Proceedings 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1992.

[28] III W. Ludwell Harrison. Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., Mar. 20, 1986. CSRD Rpt. No. 565.

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

[30] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 Int'l. Conf. on Parallel Processing, St. Charles, Illinois, August 12-16*, pages 26–30. CRC Press, Inc., 1991. Vol. II - Software.

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

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

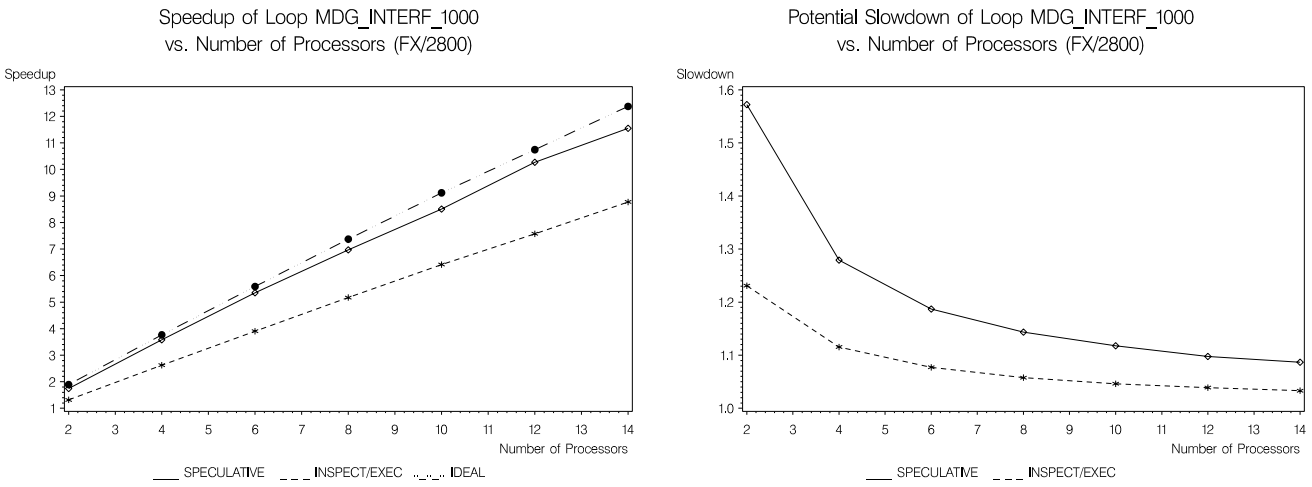



Figure 5:

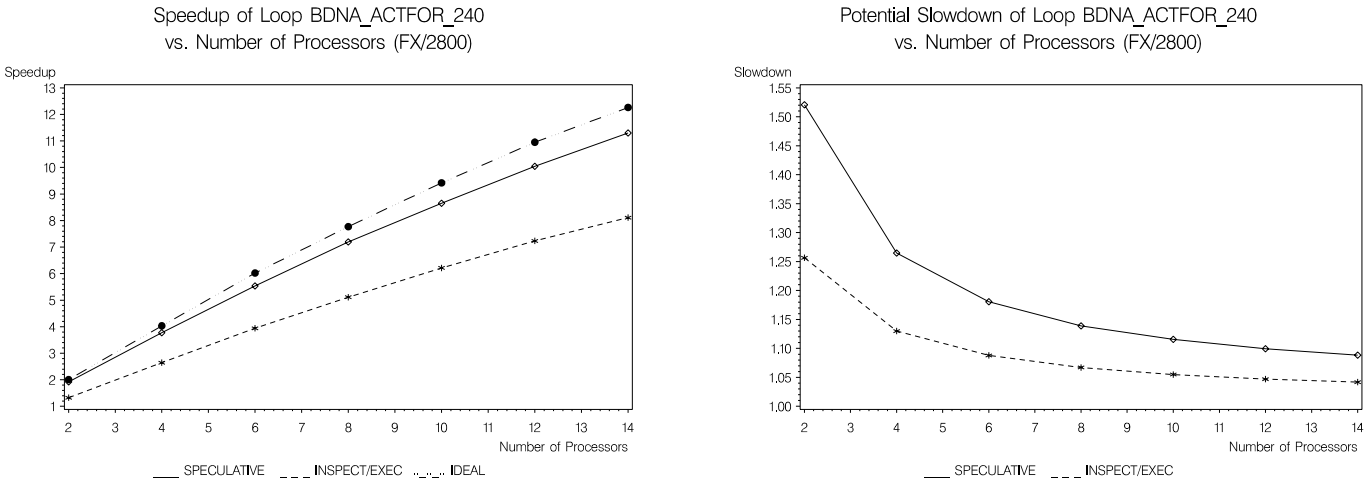



Figure 6:

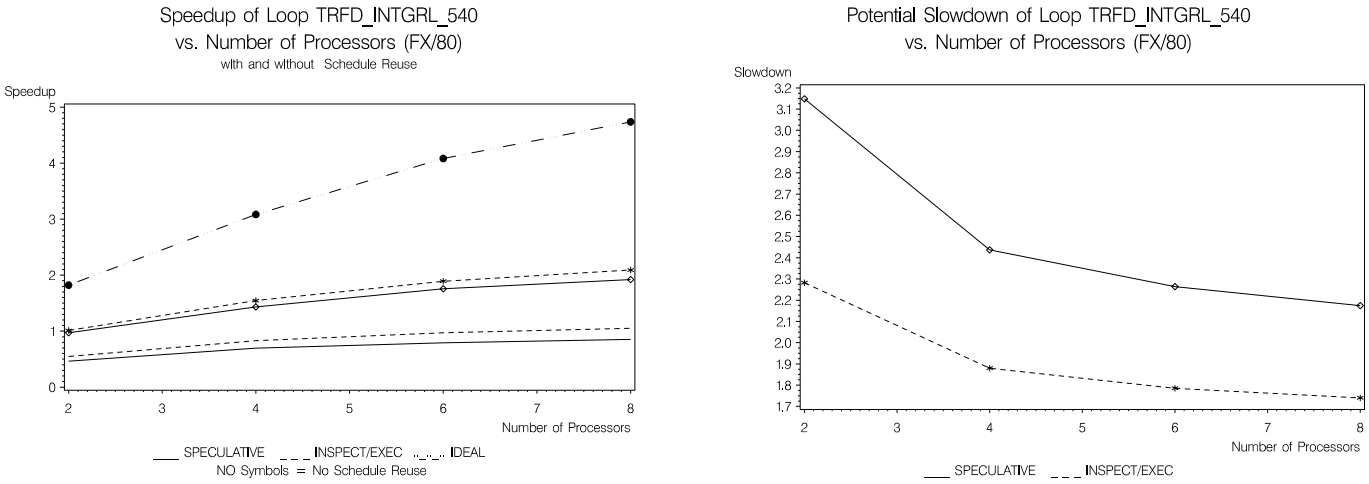
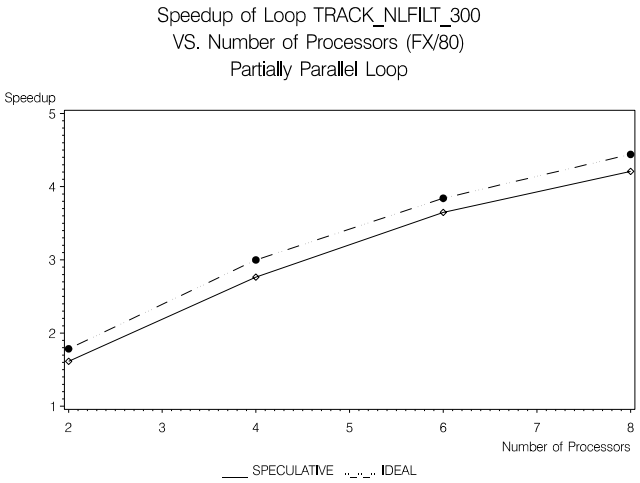



Figure 7:



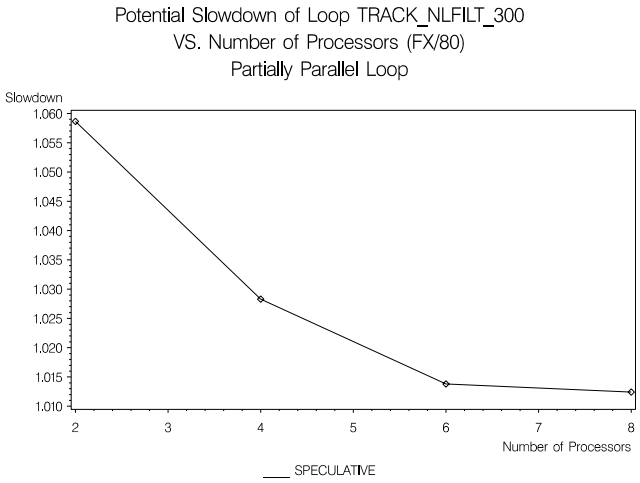
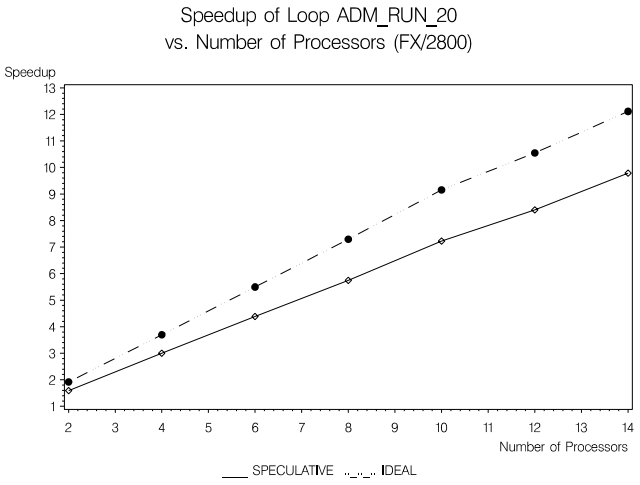




Figure 8:



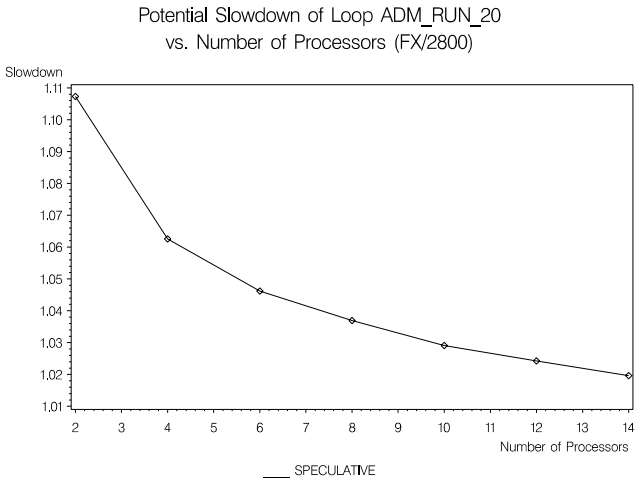
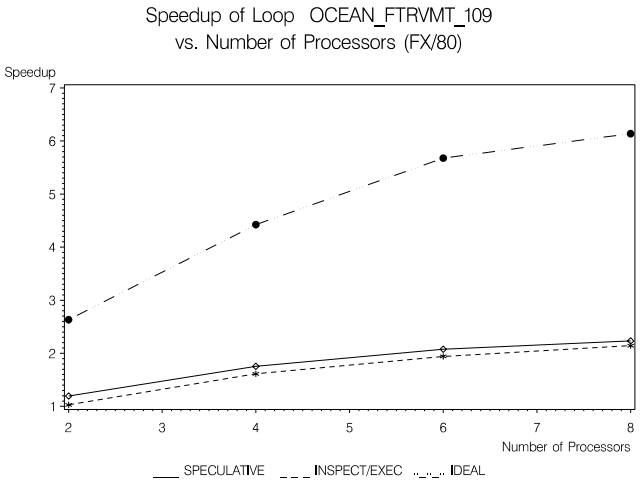




Figure 9:



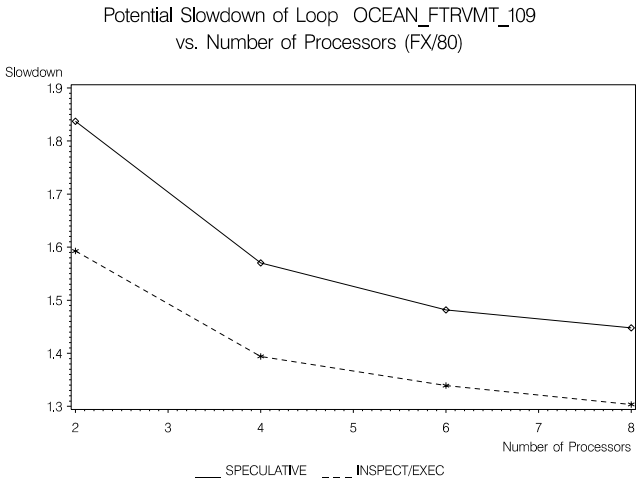




Figure 10:

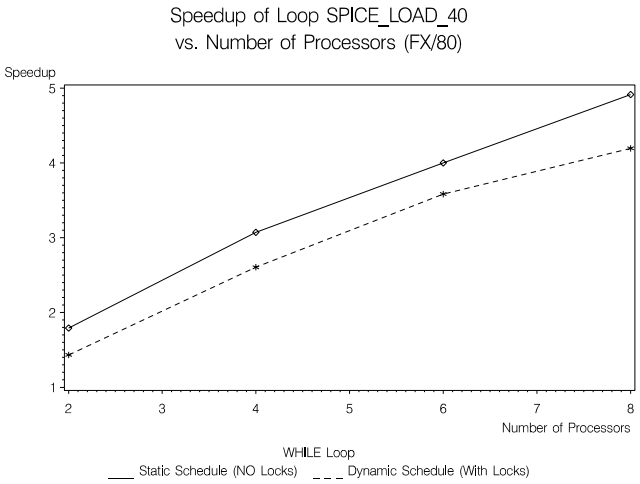



Figure 11:

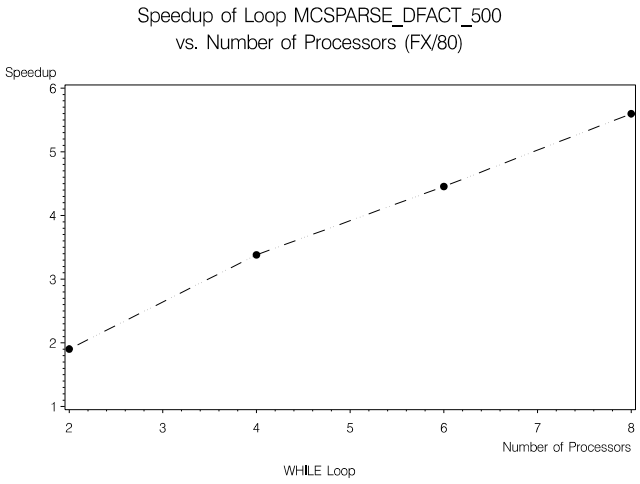



Figure 12: