The Privatizing DOALL Test:
A Run-Time Technique for DOALL Loop Identification and Array Privatization*

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

Abstract

Current parallelizing compilers cannot identify a significant fraction of fully parallel loops because they have complex or statically insufficiently defined access patterns. For this reason, we have developed the Privatizing DOALL test – a technique for identifying fully parallel loops at run-time, and dynamically privatizing scalars and arrays. The test itself is fully parallel, and can be applied to any loop, regardless of the structure of its data and/or control flow. The technique can be utilized in two modes: (i) the test is performed before executing the loop and indicates whether the loop can be executed as a DOALL; (ii) speculatively – the loop and the test are executed simultaneously, and it is determined later if the loop was in fact parallel. The test can also be used for debugging parallel programs. We discuss how the test can be inserted automatically by the compiler and outline a cost/performance analysis that can be performed to decide when to use the test. Our conclusion is that the test should almost always be applied – because, as we show, the expected speedup for fully parallel loops is significant, and the cost of a failed test (a not fully parallel loop), is minimal. We present some experimental results on loops from the PERFECT Benchmarks which confirm our conclusion that this test can lead to significant speedups.

1 Introduction

During the last two decades, compiler techniques for the automatic detection of parallelism have been studied extensively [17, 27]. From this work it has become clear that, for a class of programs, compile-time analysis has to be complemented with run-time techniques if a significant fraction of the implicit parallelism is to be detected [6, 8]. 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 [13, 21, 29].

Run-time techniques have been used practically from the beginning of parallel computing. During the 1960s, relatively simple 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, 24]. 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 [5, 13, 16, 19, 20, 21, 28, 29]. Most of this work has focussed on developing run-time methods for constructing parallel schedules for DOACROSS loops. Unfortunately, these methods have significant sequential components, rely heavily on global synchronizations (communication), or do not extract the maximum available parallelism (they make conservative assumptions). As a result, these methods have not produced scalable speedups and have not succeeded in gaining wide acceptance.

In this paper we approach the problem of determining the parallelism of loops at run-time from a different viewpoint – 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. Our interest in identifying fully parallel loops is motivated by the fact that they arise frequently in real programs. As we show, the analysis needed to test whether or not a loop is fully parallel can be done very efficiently at run-time and produces scalable speedups. User directives or execution statistics can be used to identify the loops to which this test is to be applied. The techniques presented are also capable of eliminating some memory-related dependences by dynamically privatizing scalars and arrays.

In Section 2, we discuss the analysis techniques. In Section 3, we describe how these techniques can also be used for

*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.

debugging parallel programs and for speculative parallelization. In Section 4, we discuss some important compile-time issues. Finally, 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 speedups even though the 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 Detecting Parallelism at Run-Time

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 [3, 11, 17, 27, 30]. 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 are data producer and consumer dependences, i.e., they 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, have to 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. In order to remove certain types of anti and output 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., [7, 14, 15, 25, 26]). 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 `S2` of iteration $i+1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`.

In this section we describe run-time techniques that can be used to determine if a loop can be executed in parallel. We first describe the *DOALL test* which determines if the loop, in its original form, can be executed in parallel, i.e., it decides if there are any cross-iteration dependences in the loop. We then explain how to augment the DOALL test to also determine at run-time whether all existing memory-related dependences can be removed by privatization. If it is found that all dependences can be eliminated, then the augmented DOALL test will transform the loop by privatizing the variables which give rise to the anti and/or output dependences. We will call the augmented version of the DOALL test the *Privatizing DOALL test*.

In order to identify the dependence relations among the iterations of the loop, both tests inspect the accesses to the variables that cannot be analyzed at compile time. Briefly, the inspection is done by using shadow versions of the variables under scrutiny to follow (keep a record of) the data access pattern of the original loop. After processing all the accesses contained in the original loop, some additional computation determines whether all iterations of the loop can be performed in parallel while guaranteeing the semantics of the loop. For the Privatizing DOALL test, an additional phase may be required to actually allocate the private variables.

It must be emphasized that both DOALL tests are designed to be used only at run-time on loops for which the compiler could not evaluate with certainty the data dependence relations. We recall that there are several possible situations in which it is either difficult or impossible to determine the data dependence relationships at compile time: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, non-linear subscript expressions (a fairly rare case) and, most often, subscripted subscripts.

Another very important point is that these run-time tests must be fully parallel. If the tests cannot be executed in parallel, then they would not scale with the number of processors or the data size, and the overhead associated with the tests could become a sequential bottleneck of the loop.

2.1 The DOALL test

The DOALL test described below is a pass/fail test for full parallelism of a loop, i.e., it detects if there are any cross-iteration dependences in the loop. If there are any depen-

```
S1: DO i = 1, 8                          W[1:8] = [ 1 3 2  3 7 5 6 12]
S2:  A[W[i]] = ...                       R[1:8] = [ 1 9 2  2 7 8 8 12]
S2:      ... = A[R[i]]
S3: ENDDO                              W'[1:8] = [ 1 3 2  4 7 5 6 12]
                                       R'[1:8] = [ 1 9 2 10 8 8 8 12]
```

	Position in shadow arrays												Written	Counted
	1	2	3	4	5	6	7	8	9	10	11	12	$tm(A)$	$tw(A)$
$A_w[1:12]$	1	1	1	0	1	1	1	0	0	0	0	1	7	8
$A_r[1:12]$	0	1	0	0	0	0	0	1	1	0	0	0		
$A_w[:] \wedge A_r[:]$	0	1	0	0	0	0	0	0	0	0	0	0		
$A'_w[1:12]$	1	1	1	1	1	1	1	0	0	0	0	1	8	8
$A'_r[1:12]$	0	0	0	0	0	0	0	1	1	1	0	0		
$A'_w[:] \wedge A'_r[:]$	0	0	0	0	0	0	0	0	0	0	0	0		

Figure 2: Results of the DOALL test.

dences, then this test will not identify them, it will only flag their existence. We first describe the DOALL test, as applied to a shared array A, and then give a few examples illustrating its use.

DOALL Test

1. *Marking Phase.* 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; the shadow arrays are initialized to zero, and are marked as follows.

   In parallel, for each iteration i of the loop, process all accesses to the shared array A:

   (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.

   (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.* 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 and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)

   (d) Otherwise, it IS NOT a DOALL. (There are output dependences since we overwrite at least one memory location.)

We now give a few examples of the DOALL test. Consider the loop shown in Fig. 2. Assume that the shared array $A[1:12]$ is accessed in a manner such that the dependences cannot be determined at compile time. In the first example, the reference pattern of A within the loop is given by the subscript arrays $W[1:8]$ and $R[1:8]$. The DOALL test allocates, and initializes to zero, the write and read shadow arrays, $A_w[1:12]$ and $A_r[1:12]$, respectively. We obtain the results depicted in the table. Because $A_w[2] = A_r[2] = 1$, we know there exists at least one flow or anti dependence. Since the number marked does not equal the number written, we know that there are output dependences. Therefore, the loop cannot be executed in parallel. In the second example, we use the subscript arrays $W'[1:8]$ and $R'[1:8]$, and the shadow arrays $A'_w[1:12]$ and $A'_r[1:12]$. Because the number of write accesses marked equals the number written, and since $A_w[:] \wedge A_r[:]$ is zero everywhere, we conclude that there are no cross-iteration dependences, i.e., the loop can be executed in parallel.

It should be noted that an implementation of the DOALL test need not adhere exactly to the above description. For example, communication costs may be reduced by strip-mining the loop and marking in private storage, and then merging the private arrays into a global shadow array. The complexity of the DOALL test is analyzed in Section 2.3.

2.2 The Privatizing DOALL Test

The DOALL test described above is only able to detect the presence of dependences among the iterations of the loop. In this section we explain how to augment the DOALL test so that it can determine if all dependences can be removed by privatization. If it is found that all the dependences can be eliminated, then this Privatizing DOALL test will transform the loop by allocating the private variables.

We now define a private variable, and state the criterion that must be satisfied in order for a variable to be determined as privatizable by the Privatizing DOALL test.

Definition. A *private variable* can only be accessed by the loop iteration to which it belongs.

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

```
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
```

	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		

Figure 3: Results of the Privatizing DOALL test.

Privatization Criterion. Let A be a shared array that is referenced in a loop L. A can be *privatized* by the Privatizing DOALL test 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, there are some types of dependences that the Privatizing DOALL test does not handle. Specifically, if a shared variable is initialized by reading a value that is computed outside the loop, then we will not privatize that variable. Such variables could be privatized if a *copy-in* mechanism for the external value is provided. Since this situation occurs infrequently in practice we have not addressed it in our tests.

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. One way in which this problem can be handled by our run-time tests is to associate a time stamp (iteration number) with each private variable, which is updated at every write. After the loop has been executed, the value of the private variable with the latest time stamp is copied to the original version of the variable. It should be noted that private loop variables are seldom live after the loop.

In order to simplify the description of the Privatizing DOALL test given below, we do not address the last value assignment problem. (The additions to the DOALL test are italicized.)

Privatizing DOALL Test

1. *Marking Phase.* For each shared array $A[1:s]$ whose dependences cannot be determined at compile time, in addition to the shadow arrays, $A_r[1:s]$ and $A_w[1:s]$, *we declare a shadow array* $A_{np}[1:s]$ *that will be used to flag array elements that can NOT be privatized.* As before, the shadow arrays are initialized to zero. *Initially, we assume 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.*

   In parallel, for each iteration i of the loop, process all accesses to the shared array A:

   (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.* 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 $any(A_w[:] \wedge A_r[:])$, 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 and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)

   (d) *Else if* $any(A_w[:] \wedge A_{np}[:])$*, then the loop IS NOT a DOALL and the phase ends. (There is at least one dependence that cannot be removed by privatization).*

   (e) *Otherwise, the loop can be made into a DOALL by privatizing all elements of the shared array that are written in the loop. (We remove all memory-related dependences by privatizing these array elements.)*

In order to illustrate the differences between the DOALL test and the Privatizing DOALL test, we consider the loop shown in Fig. 3, which contains memory-related dependences

index	1	2	3	4	5	6	7	8	9	10	11	12
$A_w[1:12]$	1	0	1	1	1	1	1	0	0	0	0	1
Prefix Sums	1	1	2	3	4	5	6	6	6	6	6	7
$PA[1:7]$	$A[1]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[12]$					
$R1[1:8]$	2	2	2	10	8	8	8	10				
$PR1[1:8]$	2	2	2	10	8	8	8	10				
$W[1:8]$	1	3	5	4	7	3	6	12				
$PW[1:8]$	1+d	2+d	4+d	3+d	6+d	2+d	5+d	7+d				
$R2[1:8]$	1	3	2	10	7	3	8	12				
$PR2[1:8]$	1+d	2+d	2	10	6+d	2+d	8	7+d				

Table 1: Allocating the private variables, and creating private subscript arrays.

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 the table. 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 $tw(A) \neq tm(A)$.

2.2.1 Allocating the Private Variables

If the Privatizing DOALL test determines that privatization can be used to transform the loop into a DOALL, then there are two choices: privatize the entire shared variable, or only privatize the shared memory locations (e.g., elements of the array) that are *written* in the loop. If only the elements that are written are privatized, it is possible that this technique might yield performance gains over traditional compile time privatization techniques, which usually privatize the entire array [14, 15, 25, 26]. In fact, if the data access pattern is sparse enough, it is even conceivable that the reduction in the size of the working set could lead to superlinear speedups due to cache effects.

If it is decided to privatize only the elements that are written, then the private variables can be allocated as follows. Consider the array A from the previous example. The elements of A that will be privatized are exactly those elements k for which $A_w[k] = 1$. Since $tm(A) = 7$, we will allocate enough space for seven elements of A; denote this space by $PA[1:7]$. We can determine the positions of the privatized elements of A in PA from the prefix sums of $A_w[1:12]$, e.g., the private version of $A[5]$ is contained in $PA[4]$ since the prefix sum value of $A_w[5] = 4$ (see Table 1).

In general, on each access to a shared array element $A[k]$, it must be determined whether or not $A[k]$ has been privatized, e.g., by checking $A_w[k]$. However, in the case of subscripted subscripts, we can remove the need for this check from the execution as follows. Each iteration is provided with a private subscript array (of the same dimension as the original subscript array), and all references to the subscript array will use the private version. If $A[k]$ is not privatized, then references to it in the subscript array will remain the same. If $A[k]$ is privatized, then occurrences of k in the subscript array will be replaced by the prefix sum value in $A_w[k]$ plus the offset between the starting addresses of A and PA. The private versions $PR1, PW$, and $PR2$, of the subscript arrays $R1, W$, and $R2$, respectively, are shown in Table 1, where d is the offset between the starting addresses of PA and A, i.e., $d = \&PA[0] - \&A[0]$. Actually, we can handle other cases besides subscripted subscripts in a similar manner by constructing a private subscript array $PS[1:s]$, and and transforming references such as $A[i]$ into $A[PS[i]]$. The values of $PS[:]$ are set as follows: if $A[k]$ is not private, then $PS[k] = k$, and if $A[k]$ is private, then $PS[k] = s_k + d$, where s_k is the prefix sum value in $A_w[k]$, and d is again the offset between the starting addresses of A and PA. Since this technique requires indexing out of bounds, it may cause problems on certain architectures and would be best implemented in machine language.

2.3 Complexity of the DOALL Tests

We now examine the complexity of the DOALL and the Privatizing DOALL tests. Let p denote the number of processors, let n denote the total iteration count of the loop, let s denote the number of elements in shared array, and let a denote the (maximum) number of accesses to the shared array in a single iteration of the loop. As explained below, the time required by the Privatizing DOALL 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. After all accesses have been marked in private storage, 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. The hash tables can be transferred to the global shadow arrays in $O(na/p + \log p)$ time, and the check needed to avoid marking both a read and a write in the same iteration will remain a constant time operation (although slightly more expensive). 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 method takes $O(s/p + \log p)$ time [12]. The comparisons in Step 2(b) (2(d)) of the A_w and A_r (A_{np}) 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.

From the above analysis, we conclude that the DOALL and the Privatizing DOALL tests require little communication and should scale well with all of the parameters, i.e., the number of processors, the size of the shared variable, and the number of references to the shared variable (encompassing both the number of iterations, and the number of accesses within an iteration).

2.3.1 Complexity of Run-Time Privatization

If the Privatizing DOALL test determines that privatization is needed, then we either privatize the entire shared array A, or we privatize only the elements of the shared array that are *written* during the loop. If the entire array is privatized, then the allocation can be done in constant time.

When privatizing *only the elements that are written*, the information needed is $tm(A)$, the number of elements written, and the prefix sums of A_w. Since A_w is computed during the test itself, the only additional information needed is the prefix sums, which can be computed in time $O(s/p+\log p)$ by recursive doubling [12]. In fact, the prefix sums can be computed when determining $tm(A)$ without much extra work. If the entire array is not privatized, then in the special case of subscripted subscripts, private copies of the subscript arrays are also created. Given the original subscript array and A_w, each private subscript array can be created in time $O(m)$, where m is the size of the original subscript array.

If a private variable is live after the loop terminates, then we will also need to perform a last value assignment. In this case, as mentioned before, we can keep time stamps (iteration numbers) with the private variables. Then, 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 $O(s \log p)$.

2.3.2 Schedule Reuse

Thus far, our analysis has assumed that the DOALL test or 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, some additional computation must be included to check this condition, e.g., for subscripted subscripts the old and the new subscript arrays can be compared.

3 Verifying of Parallelized Loops

We have presented the DOALL and the Privatizing DOALL tests as run-time techniques that can be used to detect the parallelism of a loop before executing it. Another important area of application for these tests is to detect *race conditions* [9] in loops that the programmer identifies as parallel. In fact, the DOALL test can be used as an efficient on-the-fly test [22] for the cases when there are no synchronization operations between parallel loop iterations. When used for this purpose, the marking phase could be run in parallel by incorporating it in the body of the parallel loop, and the analysis phase could be done after the completion of the loop. It is important to note that since the DOALL test itself is fully parallel, the outcome of the test will be valid for any loop, regardless of its data dependence structure.

There are several situations in which it may be useful to have a run-time test that can determine, for a particular input set, whether or not a loop has been validly parallelized. For example, such a test can be used to *check manually parallelized* loops. Writing and especially debugging parallel programs is a very complex task, and there are many cases in which it may be difficult to verify a program's correctness by only analyzing its output. For example, in loops where access to a variable is guarded by a branch condition the DOALL test can be used to verify that no illegal concurrent accesses to that variable are made. In fact, using the DOALL test, we have found an instance in which the programmer incorrectly identified a loop as a DOALL for this reason.

3.1 Speculative Execution

Thus far, we have advocated the DOALL test as a method which should be used at run-time to determine whether a loop should be be executed sequentially or in parallel. There are, however, certain circumstances under which it may be preferable to go even further and actually execute the loop in parallel, and determine later if, for the given input, the loop was in fact parallel. If it was not, the loop is re-executed sequentially. One example of such a situation is when it is known (from, e.g., static analysis, run-time statistics, or compiler directives) that the loop is *usually* fully parallel. Another case is when the work required to extract the data access pattern (for study by the tests) is comparable to the work performed by the loop itself; in this scenario, not too much extra work (besides the test itself) is performed, and, at the conclusion, both the status of the pass/fail test and the results of the, possibly invalid, parallel execution are known. Speculative use of the tests would be implemented in essentially the same way discussed in Section 3. In addition, the prior state of any variables modified by the parallel execution must be available for the sequential re-execution of the loop which will be required if the test fails. One alternative to saving/restoring the state of these variables is to privatize them, and to copy-in any needed external values. Then, if the test passes, only the usual copy-out of the live variables is necessary. In any case, the cost of the solution adopted for this problem must be factored into the decision of whether or not to use speculative execution. Another issue that must be dealt with is exception handling. A simple solution is to abandon the parallel execution if an exception occurs, and execute the loop sequentially.

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

```
      **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
```

```
      **Extracted Loop for Marking**
      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
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 replaced by accesses to the shadow variables for the marking phase of the DOALL test. This loop is extracted from loop 540 in subroutine INTGRL of TRFD from the PERFECT Benchmarks. The DOALL test is applied to the shared array X. In the inspector loop (right), writes are marked in the shadow array `X_w` by iteration number, so the shadow array can be reused. The number of writes marked is recorded in the private variable `tw_i`.

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 tests, i.e., from testing loops that are, in fact, not parallel.

4 Automatic Application of the Tests

In the previous sections we have discussed run-time data dependence and privatization techniques. These techniques are automatable and a good compiler could easily insert them in the original code. In this section, we address some of the issues that are involved with the automatic utilization of these tests. We begin with a brief outline of how a compiler might apply the DOALL test.

1. *At Compile Time.*
   (a) Generate an inspector loop for the marking phase of the test. This is done by collecting all accesses to the shared variables under study into a separate loop, where they are replaced by accesses to the appropriate shadow variables. An example is shown in Fig. 4, where there are only output dependences.
   (b) A cost/performance analysis determines whether the test should be applied. If not, the inspector will be discarded and a sequential version of the loop will be generated.
   (c) Generate a multi-version loop with options for sequential and parallel executions of the original loop. The run-time selection among the versions of the loop will depend upon the outcome of the analysis of the shadow variables marked by the inspector loop. The analysis can be done by calls to a run-time library.
2. *At Run-Time.*
   (a) Execute the marking phase of the test, i.e., run the inspector generated in Step 1(a).
   (b) Execute the analysis phase of the test, which gives the pass/fail result of the test.
   (c) Execute the selected version of the loop indicated by the result of Step 2(b).
   (d) Collect statistics for use in future runs, and/or for schedule reuse in this run.

Generating the Inspector Loop

An inspector loop can be formed from the original loop by replacing accesses to the shared variables with accesses to the appropriate shadow variables; also, to avoid side effects, all other program variables written in the loop must be privatized. However, since we want the inspector to run as fast as possible, we want to exclude any computation from the original loop that does not affect the pattern of access to the variables under study. In the best case, the access pattern of the shared variable is known before entering the loop, e.g., the subscripts are in a pre-determined subscript array. In this simple case, the access pattern of the array can be processed in complete isolation from the rest of the loop. However, there are cases in which it is not possible for the data access operations to be completely isolated from the other computation in the loop. Specifically, if the address (subscript) computation of an array element is performed in the same π-block (strongly connected component in the dependence flow graph) as the statement that references the array using that subscript, then the address computation cannot be removed from the inspector loop. For example, the access pattern may be computed inside the loop itself, perhaps in a preparation phase, e.g., a loop first collects in a subscript array the indices of the elements of a much larger data structure that will be

processed subsequently in the loop. In this case, the inspector loop will have to include the access pattern computation. Another difficult case is when the statements accessing the array in question are control flow dependent upon data computation performed inside the loop (with the exception of loop indices and loop invariant variables). In this case, the inspector must either execute the computation that defines the predicates, or assume conservatively that all branches are taken (possibly introducing false dependences). In order to minimize such situations, the compiler should reduce the control flow affecting data accesses as much as possible by using known techniques such as *loop distribution*, *loop invariant hoisting*, and, generally, *code motion*.

Determining When to Apply the Test

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of the tests will be enhanced if their run-time overhead is avoided when the test is likely to fail. There are three main factors that the compiler should consider when deciding whether or not to apply the test: the probability that test will pass (i.e., that the loop is in fact a DOALL), the speedup that would be obtained if the test passes, and the slowdown incurred if the loop is not a DOALL.

In order to predict the outcome of the test, 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 that is in fact a DOALL, the *ideal speedup*, Sp_{id}, of L is the ratio between its sequential and its parallel execution times, T_{seq} and T_{doall}, respectively. However, if L is parallelized by the DOALL test, the *attainable speedup*, Sp, must account for the overhead required by the marking and analysis phases of the test, T_{mark} and $T_{analysis}$, respectively.

$$Sp_{id} = \frac{T_{seq}}{T_{doall}} \qquad Sp = \frac{T_{seq}}{T_{mark} + T_{analysis} + T_{doall}}$$

Using static analysis, the compiler can compute an estimate for Sp by estimating $T_{seq}, T_{doall}, T_{mark}$ and $T_{analysis}$. The values T_{seq} and T_{doall} can be estimated using some architectural model, e.g., instruction counting. Our analysis in Section 2.3 predicts that $T_{mark} = O(na/p + \log p)$ and $T_{analysis} = O(na/p + \log p)$, where p is the number of processors, n is the number of iterations of the loop, and a is the maximum number of accesses to the shared array in a single iteration of the loop. In practice, $T_{analysis}$ should be fairly well modeled by this expression, i.e., $T_{analysis} \approx c(na/p + \log p)$, where c is some small constant. However, the estimate of T_{mark} may not always be as good. The reason for this is that our analysis in Section 2.3 implicitly assumed that the data access pattern is known before loop entry. As discussed above, in the worst case $T_{mark} \approx T_{doall}$, i.e., the inspector loop is computationally equivalent to the original loop. However, in all cases, an estimate of T_{mark} can be obtained by static analysis of the inspector loop, e.g., by instruction counting.

Note that in the worst possible case $T_{mark} \approx T_{analysis} \approx T_{doall}$, and that even in this case the attainable speedup predicted for the DOALL test is $\approx \frac{1}{3} Sp_{id}$. Although 33% of ideal speedup may not appear impressive on an eight processor machine, these tests were designed for massively parallel processors (MPPs), and on such a machine this in an excellent performance when compared to the alternative of sequential execution.

It is also instructive to examine *slowdown* incurred by a failed test, i.e., when the loop must be executed sequentially. In this case, T_{seq} is increased by $T_{mark} + T_{analysis}$. Note that since the DOALL test is fully parallel, in the worst case we have $T_{mark} \approx T_{analysis} \approx \frac{1}{p} T_{seq}$, i.e., when the marking phase is work-equivalent to the loop. Thus, the cost of performing a failed test is proportional to $\frac{1}{p} T_{seq}$:

$$T_{seq} + T_{mark} + T_{analysis} \approx T_{seq} + \frac{2}{p} T_{seq} = (1 + \frac{2}{p}) T_{seq}$$

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

Based on the outcome of the cost/performance analysis, the compiler will determine whether the test should be performed, and if it decides to use the test, it must also decide how the test should be applied: using the inspector/executor paradigm (i.e., first test, and then execute) or in a speculative manner (i.e., test and execute simultaneously, as described in Section 3.1). In practice, when T_{mark} approaches T_{doall}, the use of the tests in a speculative way may be beneficial. The additional overhead mentioned in Section 3.1 (state save/restore) must be considered when deciding whether to use speculative execution.

5 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 Cedar Fortran [10] implementation of the DOALL tests. 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 five loops contained in the PERFECT Benchmarks [4] that could not be parallelized by any compiler available to us. A summary of our results is given in Table 2. For each loop, the methods applied and the speedup obtained are reported. As a reference, we also give the ideal speedup, which was measured using an optimally parallelized (by hand) version of the loop. In addition, when the inspector/executor version of the DOALL test was applied we mention the computation performed by the inspector; the notations *branch predicates* 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 the following, we discuss each loop in more detail. Our estimates are made using a simple instruction counting model: for loops (vectors), we use the product of the number of instructions and the number of iterations (elements).

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 2.3). As discussed in Section 2.3, one way to accomplish this is to use hash tables for the shadow arrays. However, since our implementation of the DOALL tests does not yet optimize with hash tables, in some cases the speedups shown

Benchmark Subroutine Loop	Experimental Results				Description of Data Accesses	Inspector (computation)
	Technique	Speedup		#Proc		
		obtained	ideal			
OCEAN FTRVMT Loop 109	insp/exec	2.1	6.0	8	kernel-like loop accesses a vector with run-time determined strides	data accesses replicates loop
MDG INTERF Loop 1000	insp/exec private	8.8	12.4	14	accesses to a privatizable vector guarded by loop computed predicates	data accesses branch predicates
BDNA ACTFOR Loop 240	insp/exec private	7.6	11.6	14	accesses a privatizable array indexed by a subscript array computed inside loop	data accesses subscript array
TRFD INTGRL Loop 540	insp/exec sched reuse	1.5 2.2	4.6	8	small triangular loop accesses a vector indexed by a subscript array computed outside loop	data accesses replicates loop
TRACK NLFILT Loop 300	speculative	4.2	4.4	8	accesses an array indexed by a subscript array computed outside loop, access pattern guarded by loop computed predicates	not applicable

Table 2: Summary of Experimental Results

in Figures 5 through 9 do not appear to scale. In particular, in cases in which the size of the array under test is $> na/p$, our implementation will display $T_{mark} \approx T_{analysis} \approx s$ instead of $\approx na/p$ as predicted, (i.e., the marking and analysis phases of the test will touch the entire shared array, regardless of its access pattern in the loop under study). This situation is encountered with the loops from Ocean and TRFD.

OCEAN–FTRVMT–Loop 109. This loop is utilized in the computation of a 2-dimensional FFT. It is invoked 26,000 times, and accounts for 40% of the sequential execution time of the program. For this loop we estimate $T_{mark} \approx T_{analysis} \approx T_{doall}$, and predict $Sp \approx \frac{T_{seq}}{3T_{doall}} = \frac{1}{3}Sp_{id}$. In fact, as shown in Fig. 5, the speedup obtained by the DOALL test is about 1/3 of the ideal speedup.

MDG–INTERF–Loop 1000. This loop calculates intermolecular interaction forces. In order to avoid false dependences, our inspector computes the branch predicates and has an estimated complexity $T_{mark} \approx .2T_{doall}$. Since $T_{analysis}$ is small (the shadow vector has only 14 elements), we expect the Privatizing DOALL test to obtain $Sp \approx .7Sp_{id}$. The results shown in Fig. 6 display a pretty good fit to this estimation.

BDNA–ACTFOR–Loop 240. This loop selects certain elements from a large array, and processes the selected elements later in the loop. The speedup obtained for this loop using the Privatizing DOALL test is almost 2/3 of the ideal speedup (see Fig. 7). This speedup cannot be accurately predicted at compile time because the number of selected elements is not known until run-time, and the inspector executes the selection phase (computing the subscripts), but not the subsequent processing phase.

TRFD–INTGRL–Loop 540. For this small triangular loop, $Sp_{id} \approx 4.5$ on the Alliant FX/80 because of load imbalance. Statically, we would predict $T_{doall} \approx T_{mark} \approx T_{analysis}$, and $Sp \approx \frac{1}{3}Sp_{id}$. However, since our implementation does not yet optimize with hash tables, the access pattern of our shadow arrays is rectangular (versus the triangular pattern of the loop) and $T_{mark} \approx T_{analysis} \approx 2T_{doall}$, so that $Sp \approx \frac{1}{5}Sp_{id} < 1$.

However, Loop 540 is executed seven times (within an outer loop) and uses a larger subscript array on each subsequent execution. We could use *schedule reuse* if the current subscript array were a subset of the one from the previous invocation. In order to obtain this subset relation, we executed the calling loop in reverse order – and verified our action with the DOALL test. For the seven invocations of Loop 540, we obtained $Sp = \frac{1}{2}Sp_{id}$ – including the subscript array comparisons (see Fig. 8). The example shows that in the most disadvantageous cases we can obtain significant speedups by using a simple schedule reuse technique.

TRACK–NLFILT–Loop 300. Since the inspector would have had to perform almost all of the computation of the loop, we decided to use *speculative execution* as described in Section 3.1. Of the 59 executions of the loop, only five times it was not a DOALL. In these cases we restored the values of the variables modified by the loop (which were saved before invoking the loop) and re-executed the loop sequentially. The results are depicted in Fig. 9. Note that the speedup obtained here is for a loop that has had both parallel and sequential instantiations. Of course, such a speculative technique can be costly if the test fails most of the time, but in cases such as this where extracting the access pattern is not possible without executing the whole loop, it is an attractive alternative.

6 Conclusion

In this paper we have approached the problem of determining the parallelism of loops at run-time from a different viewpoint – instead of finding a valid parallel execution schedule for the loop, we have solved the problem of simply deciding if the loop is fully parallel - a frequent occurrence in real programs. In addition, the techniques presented are also capable of eliminating some memory-related dependences by dynamically privatizing scalars and arrays.

We have shown that the concept of run-time data dependence checking is a useful solution for loops that cannot be

sufficiently analyzed by a compiler. We would like to re-emphasize that our methods are applicable to all loops, without any restrictions on their data or control flow. Both inspector/executor and speculative 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 the DOALL test is proportional to $\frac{1}{p}T_{seq} + \log p$, where p is the number of processors available. 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 (T_{seq}). When applying the DOALL test to a loop, our performance gain/loss will range from at least 1/3 of the ideal speedup (which can reach into the hundreds for MPPs) when the test passes, to an additional few percentage points of the sequential execution time if the test fails. In other words, the test 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 test 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 test.

In the near future we plan to implement these methods in POLARIS, a restructuring compiler currently being developed at the University of Illinois, that targets the latest generation of massively parallel architectures.

Acknowledgement

We would like to thank William Blume for identifying applications for our test and other useful advice, and Paul Peterson for many fruitful discussions. We would also like to thank Nancy Amato for carefully reviewing the manuscript.

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

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

[8] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proc. of the 1991 Int'l. Conf. on Parallel Processing*, pages 17–25, August 1991.

[9] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, January 1992.

[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 *Proc. of the 8th ACM Symp. 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 *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 83–91, May 1993.

[14] Z. Li. Array privatization for parallel execution of loops. In *Proc. of the 19th Int'l. Symp. on Computer Architecture*, pages 313–322, 1992.

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

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

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

[18] C. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and Their Imp act on Architecture Design. *IEEE Trans. on Comput.*, C-37(8):991–1004, August 1988.

[19] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In Dr. H.D. Schwetman, editor, *Proc. 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 *Proc. of the 1989 ACM Int'l. Conf. 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] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. SigPlan Conf. Programming Languages Design and Implementation (PLDI).*, pages 285–297. ACM Press, New York, 1989.

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

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

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

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

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

[28] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proc. 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.

[29] 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.

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

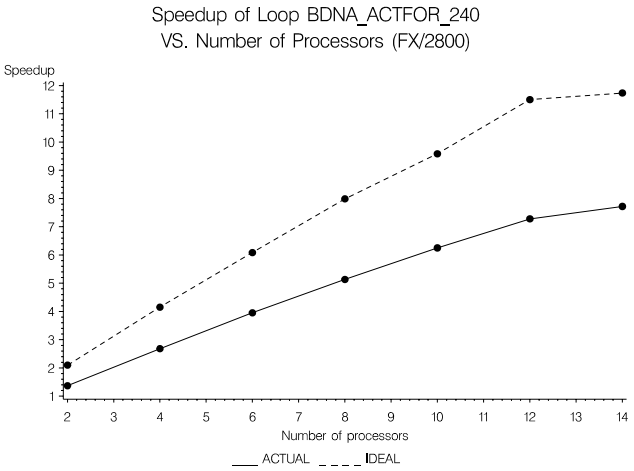

Figure 7:

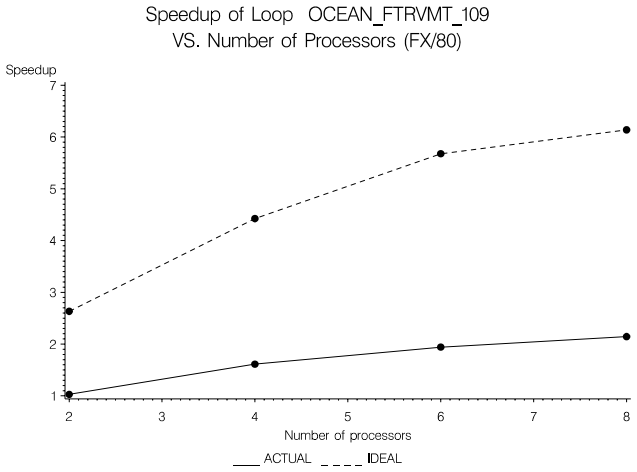

Figure 5:

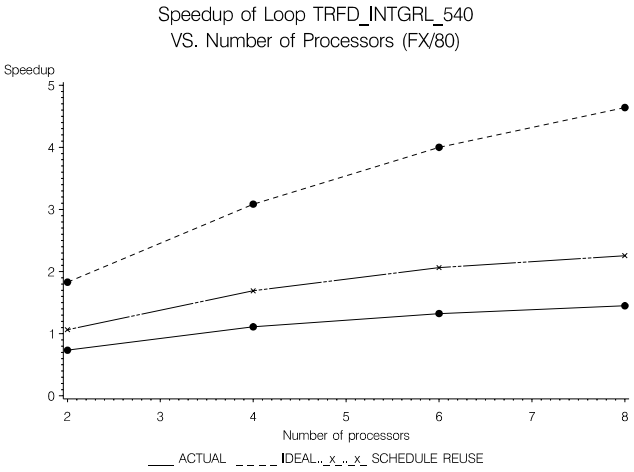

Figure 8:

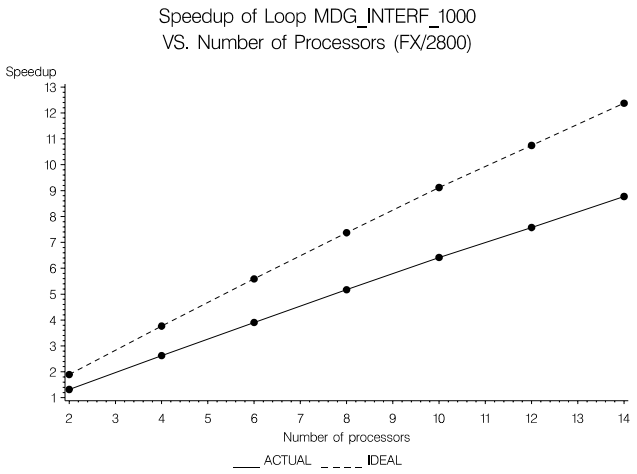

Figure 6:

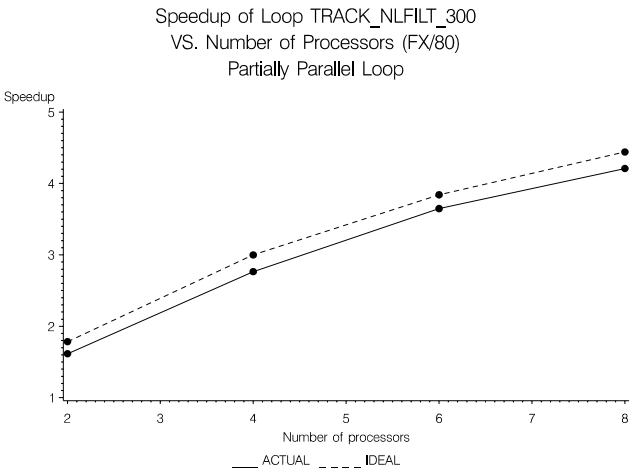

Figure 9: