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

Lawrence Rauchwerger, *Member*, *IEEE*, and David A. Padua, *Senior Member*, *IEEE*

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

Index Terms—Compilers, parallel processing, speculative, run-time, DOALL, reduction, privatization.

1 Introduction

To achieve a high level of performance for a particular program on today's supercomputers, software developers are often forced to tediously hand-code optimizations tailored to a specific machine. Such hand-coding is difficult, increases the possibility of error over sequential programming, and the resulting code may not be portable to other machines. Restructuring, or parallelizing, compilers address these problems by detecting and exploiting parallelism in sequential programs written in conventional languages. Although compiler techniques for the automatic detection of parallelism have been studied extensively over the last two decades (see, e.g., [29], [43]), current parallelizing compilers cannot extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically unknown access pattern. Typical examples of applications containing such loops are complex simulations such as SPICE for circuit simulation, DYNA-3D and PRONTO-3D for structural mechanics modeling, GAUSSIAN and DMOL for quantum mechanical simulation of molecules, CHARMM and DISCOVER for molecular dynamics simulation of organic systems, and FIDAP for modeling complex fluid flows [12].

Thus, in order to realize the full potential of parallel computing it has become clear that static (compile-time) analysis must be complemented by new methods capable of automatically extracting parallelism at *run-time* [10], [12], [15]. Run-time techniques can succeed where static compilation fails because they have complete information about the access pattern. For example, input-dependent or dynamic data distribution, memory accesses guarded by run-time dependent conditions, and subscript expressions can all be analyzed unambiguously at run-time. In contrast, at compile-time, the access pattern of some programs cannot be determined, sometimes due to limitations in the current analysis algorithms, but often because the necessary information is just not available, i.e., the access pattern is a function of the input data. For example, most dependence analysis algorithms can only deal with subscript expressions that are linear in the loop indices. In the presence of nonlinear expressions, a dependence is usually assumed. Also, generally compilers conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this last limitation when the index arrays are computed using only statically known values, nothing can be done at compile-time when the index arrays are a function of the input data [22], [37], [49].

1.1 Speculative `doall` Parallelization

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

- Instead of finding a valid parallel execution schedule for the loop that satisfies all cross-iteration dependences, we focus on the problem of simply deciding if the loop is fully parallel, that is, determining whether

- *L. Rauchwerger is with the Department of Computer Science, Texas A&M University, College Station, TX 77843. E-mail: rwerger@cs.tamu.edu.*
- *D. Padua is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61820. E-mail: padua@cs.uiuc.edu.*

Manuscript received 15 Feb. 1998; revised15 July 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 108177.

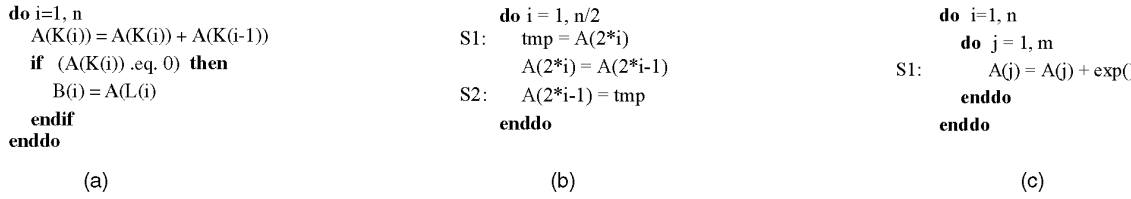


Fig. 1. Examples of loops requiring different types of parallelization techniques.

or not the loop has cross-iteration dependences. (This approach was also taken in [32].)

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

Compilers often transform programs to optimize performance. Two of the most effective transformations for increasing the amount of parallelism in a loop (i.e., removing certain types of data dependences) are *array privatization* and *reduction parallelization*. Krothapalli and Sadayappan [18] proposed an inspector method for run-time privatization which relies heavily on synchronization, inserts an additional level of indirection into all memory accesses, and calls for dynamic shared memory allocation. In our previous work [32], [31], we gave an inspector method without these drawbacks for determining whether a `do` loop can be executed as a `doall`, perhaps by privatizing some shared variables. No previous run-time methods have been proposed for parallelizing reduction operations.

In this paper, we present several new ideas: First, we advocate the use of run-time tests to validate the execution of a loop that is speculatively executed in parallel. The advantage of this approach is that the computation of the loop is performed concurrently with the tests, i.e., the memory access pattern does not need to be extracted and analyzed separately as in inspector/executor methods.[1] As will be shown later, the test performed during the speculative loop execution is almost independent of the original computation, in effect introducing an additional level of fine grain parallelism which can be exploited by modern microprocessors and, therefore, does not significantly increase execution time.

The distribution of a loop into an inspector and executor loop is often not advantageous: If the address computation of the array under test depends on the actual data computation, i.e., there is a dependence cycle between data and address computation, as exemplified by Fig. 1a, then the inspector becomes both computationally expensive and has side effects. This means that shared arrays would be modified during the execution of the inspector loop and saving the state of these variables would be required—making the inspector equivalent to the loop itself. Moreover, a fully parallel execution (`doall`) of such an inspector cannot produce a correct address trace and cannot be used in the computation of a parallel execution schedule that enforces cross-iteration dependences. Inspectors for such loops may have to be run sequentially.

Thus, the inspector/executor approach is not a generally profitable method. However, it should be noted that speculative parallelization is inherently optimistic. When a loop is not parallel the incurred potential slowdown may possibly be greater for the speculative run-time test than for its inspector counterpart. A parallel inspector, if available, may also be used to generate a valid execution schedule using synchronizations. However, our goal is to parallelize only fully parallel loops, which are quite frequent and offer the greatest potential for scalable program speedup.

In Section 5, we present experimental results on loops from the PERFECT Benchmarks which substantiate our claim that speculative techniques can yield significant speedups which are often superior to those obtainable by inspector/executor methods. Second, in addition to array privatization, the new techniques are capable of testing at run-time the validity of the powerful reduction parallelization transformation. In particular, for an array element (or section), our run-time methods are able to detect whether it participated exclusively in a reduction operation, or if all its accesses were either read-only or privatizable. If all the memory references in a `do` loop fall under any of these categories then the speculative concurrent execution of the loop was valid, i.e., the loop was indeed parallel. The new algorithms consider only data dependences caused by actual cross-iteration data-flow (a flow of values). Thus, they may potentially qualify more loops as parallel than the method in [32] which conservatively considered the dependences due to every memory reference—even if no cross-iteration data-flow occurred at run-time. This situation could arise for example when a loop reads a shared variable, but then only uses it conditionally.

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

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

1. If desired, all of our run-time tests can be applied in inspector/executor mode.

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 [47], [48].

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 [7], [20], [29], [43], [50]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write), *anti* (write after read), and *output* (write after write). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory-related dependences, are caused by the reuse of memory, e.g., program variables.

If there are flow dependences between accesses in different iterations of a loop, then the semantics of the loop cannot be guaranteed if the loop is executed in fully parallel form. For example, the iterations of the loop in Fig. 1a must be executed in order of iteration number because iteration $i + 1$ needs the value that is produced in iteration i, for $1 \leq i < n$. In principle, if there are no flow dependences between the iterations of a loop, then the loop may be executed in fully parallel form. The simplest situation occurs when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and the loop, as is, can be executed as a `doall` (i.e., a fully parallel execution). If there are no flow dependences, but there are anti or output dependences, then the loop must be modified to remove all these dependences before it can be executed in parallel. Not all such situations can be handled efficiently. In order to remove certain types of dependences and execute the loop as a `doall`, two important and effective transformations can be applied to the loop: *privatization* and *reduction parallelization*.

Privatization creates, for each processor cooperating on the execution of the loop, private copies of the program variables that give rise to anti or output dependences (see, e.g., [11], [23], [24], [39], [40]). The loop shown in Fig. 1b is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S2` of iteration i and statement `S1` of iteration $i + 1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`. In this paper, the following criterion is used to determine whether a variable may be privatized:

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

In general, dependences that are generated by accesses to variables that are only used as workspace (e.g., temporary variables) *within* an iteration can be eliminated by privatizing the workspace. However, according to the above criterion, if a shared variable is read first, then that variable cannot be privatized. When the value of all variables which are read first comes from outside the loop, then such variables could be privatized if a *copy-in* mechanism for the external value is provided. The *last value assignment* problem is the conceptual dual 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 (nonprivatized) version of that variable. It should be noted that the need for values to be copied into or out of private variables occurs infrequently in practice.

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

DEFINITION. *A* reduction variable *is a variable whose* value *is used in one associative operation of the form* $x = x \otimes exp$, *where* $\otimes$ *is the associative operator and* x *does not occur in* exp *or anywhere else in the loop.*

Due to the finite precision of computers, the properties of the $\otimes$ operator can only be assumed to be true. It is important to note that if the reduction operation is also *commutative*, its substitution with a parallel algorithm can be done with fewer restrictions (e.g., dynamic scheduling of `doall` loops can be employed). Reduction variables are therefore accessed in a certain specific pattern (which leads to a characteristic data dependence graph). A simple but typical example of a reduction is statement `S1` in Fig. 1c. The operator $\otimes$ is exemplified by the + operator, the access pattern of array $A(:)$ is *read, modify, write*, and the function performed by the loop is to add a value computed in each iteration to the value stored in $A(:)$. This type of reduction is sometimes called an *update* and occurs quite frequently in programs.

There are two tasks required for reduction parallelization: *recognizing the reduction variable* and *parallelizing the reduction operation*. Privatization needs to recognize privatizable variables by performing only data dependence analysis. It depends only on the access pattern and not on the operation type. Several parallel methods are known for performing reduction operations. One typical method is to transform the `do` loop into a `doall` and enclose the access to the reduction variable in an unordered critical section [15], [50]. Drawbacks of this method are that it is not scalable and it requires synchronizations which can be very expensive in large multiprocessor systems. A scalable method can be obtained by noting that a reduction operation is an associative and commutative recurrence and can thus be parallelized using a recursive doubling algorithm [19], [21]. In this case, the reduction variable is privatized in the transformed `doall`, and the final result of the reduction operation is computed in an interprocessor reduction phase following the `doall`, i.e., the result is produced using the partial results computed in each processor as operands for a reduction operation (with the same operator) across the processors. Thus, the difficulty encountered by compilers in parallelizing loops with reductions arises not from finding a parallel algorithm but from recognizing the reduction statements. So far this problem has been handled at compile-time by syntactically pattern matching the loop statements

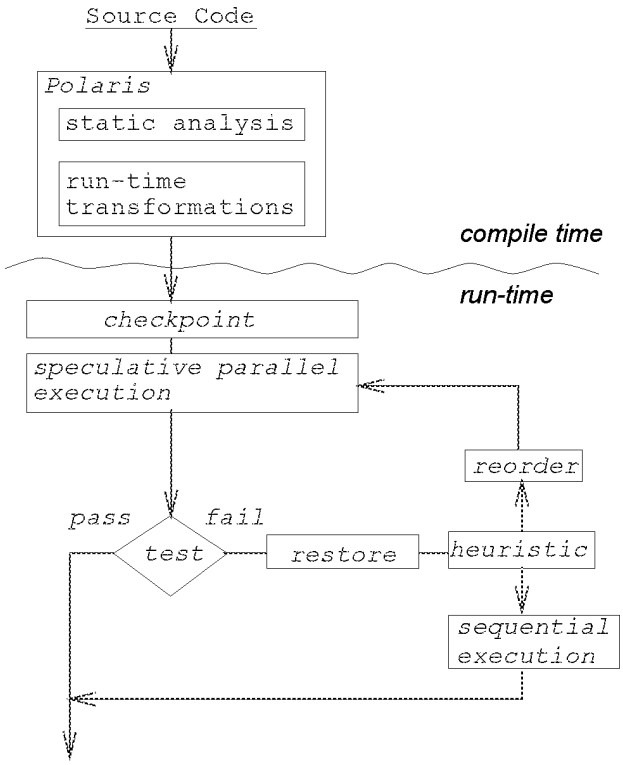


Fig. 2. Integration of speculative run-time parallelization.

with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement [50].

3 Speculative Parallel Execution of do Loops

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

- A mechanism of saving/restoring state: to save the original values of the program variables for the possible sequential reexecution 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.

An overall organization of a system for specualtive execution is shown in Fig. 2.

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

Hazard Detection. There are essentially two types of errors (hazards) that could occur during the speculative parallel execution: 1) exceptions and 2) the presence of cross-iteration dependences in the loop. A simple way to deal with exceptions is to treat them as an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, clear the exception flag, restore the values of any altered program variables, and execute the loop sequentially. Below, we present techniques that can be used to detect the presence of cross-iteration dependences in the loop and to test the validity of any privatization and/or reduction parallelization transformations that were applied.

An Automatable Strategy. The main factors that the compiler should consider when deciding whether to speculatively parallelize a loop are: the probability that the loop is a `doall`, the speedup obtained if the loop is a `doall`, and the slowdown incurred if the loop is not a `doall`. For example, the compiler might base its decision on a ratio of the estimated run-time cost of an erroneous parallel execution to the estimated run-time cost of a sequential execution. If this ratio is small, then significant performance gains would result from a successful (valid) parallelization of the loop, at the risk of increasing the sequential execution time by only a small amount. In order to perform a cost/benefit analysis and to predict the parallelism of the loop, the compiler should use static analysis and run-time statistics (collected on previous executions of the loop or from different codes); in addition, directives about the parallelism of the loop might prove useful. In Section 3.3, a complexity analysis of our run-time tests is presented that can be used to statically predict the minimum obtainable speedup and the maximum potential slowdown for a loop parallelized using our techniques.

3.1 Run-Time Data Dependence Analysis

In this section, we describe an efficient run-time technique that can be used to detect the presence of cross-iteration dependences in a loop that has been speculatively executed in parallel. If there are any such dependences, then this test will not identify them, it will only flag their existence. We note that the test need only be applied to those scalars and arrays that cannot be analyzed at compile-time. In addition, if any shared variables were privatized

for the speculative parallel execution, then this test can determine whether those variables were in fact validly privatized.

An important source of ambiguity that cannot be analyzed statically and potentially generates overly conservative data dependence models is the run-time equivalent of *dead code*. A simple example is when a loop first reads a shared array element into a local variable but then only conditionally uses it in the computation of other shared variables. If the consumption of the read value does not materialize at run-time, then the read access did not in fact contribute to the data flow of the loop and, therefore, could not have caused a dependence. Since predicates seldom can be evaluated statically, the compiler must be conservative and conclude that the read access causes a dependence in every iteration of the loop. The test given here improves upon the more conservative Privatizing `doall` (PD) test described in [32] by checking only the dynamic data dependences caused by the actual cross-iteration flow of values stored in the shared arrays (the PD test was checking exclusively reference patterns). This is accomplished using a technique we call *dynamic dead reference elimination*, which is explained in detail following the description of the test.

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

The Lazy (Value-Based) Privatizing `doall` Test (LPD Test)

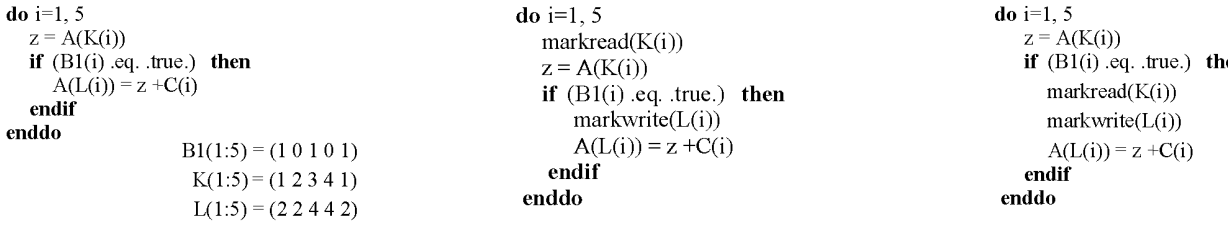
1) *Marking Phase.* (Performed during the speculative parallel execution of the loop.) For each shared array $A[1 : s]$ whose dependences cannot be determined at compile time, we declare read and write shadow arrays, $A_r[1 : s]$ and $A_w[1 : s]$, respectively. In addition, we declare a shadow array $A_{np}[1 : s]$ that will be used to flag array elements that *cannot* be validly privatized. Initially, the test assumes that all array elements *are* privatizable, and if it is found in any iteration that the value of an element is used (read) before it is redefined (written), then it will be marked as not privatizable. The shadow arrays A_r, A_w, and A_{np} are initialized to zero. During each iteration of the loop, all definitions or uses of the values stored in the shared array A are processed:
   a) Uses (done when the value that was read is used): If this array element has not yet been modified (written) in this iteration, then set the corresponding element in A_r and set the corresponding element in A_{np}, i.e., mark it as *not* privatizable.
   b) Definitions (done when the value is written): Set the element in A_w corresponding to the array element that is modified (written) and clear its corresponding A_r (if set).
   c) Count the total number of write accesses to A that are set in this iteration and store the result in $tw_i(A)$, where i is the iteration number.
2) *Analysis Phase.* (Performed after the speculative parallel execution.) For each shared array A under scrutiny:
   a) Compute i) $tw(A) = \sum tw_i(A)$, i.e., the total number of definitions (writes) that were marked by all iterations in the loop, and ii) $tm(A) = sum(A_w[1 : s])$,[2] i.e., the total number of marks in $A_w[1 : s]$. Note that $nod = tm(A) - tw(A)$ represents the number of times the elements of A have been overwritten across iterations ($nod > 0$ implies the existence of cross-iteration output dependences).
   b) If $any(A_w[:] \wedge A_r[:])$,[3] [4] i.e., if the marked areas are common *anywhere*, then the loop *is not* a `doall` and the phase ends. (Since we define (write) and use (read, but do not define) values stored at the same location in different iterations, there is at least one flow or anti-dependence.)
   c) Else if $tw(A) = tm(A)$, then the loop *is* a `doall` (without privatizing the array A). (Since we never overwrite any memory location, $nod = 0$, i.e., there are no output dependences.)
   d) Else if $any(A_w[:] \wedge A_{np}[:])$, then the array A *is not* privatizable. Thus, the loop, as executed, *is not* a `doall` and the phase ends. (There is at least one iteration in which some element of A was used (read) before it has been modified (written).)
   e) Otherwise, the loop was made into a `doall` by privatizing the shared array A. (We remove all memory-related dependences by privatizing this array.)

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

DEFINITION. *A* dynamic dead read reference *in a loop is a read access of a shared variable that does not contribute to the computation of any other shared variable which is live at loop end.*

A dynamic dead reference is not used in the computation of the value of a shared variable live at loop end nor is it used in the control flow affecting such variables. The value obtained through a dynamic dead read does not contribute to the data flow of the loop. Ideally, such accesses should not introduce false dependences in either the static or the run-time dependence analysis. If it is possible to determine the dead references at compile time, then we can just ignore them in our analysis. Since this is generally not possible (control flow could be input dependent), the compiler should identify the references that have the potential to be unused and insert code to solve this problem at run-time. In Fig. 3, we give an example where the compiler can identify such a situation by following the def-use chain built by using array names only. To avoid introducing false dependences, the marking of the read shadow array is postponed until the value that is read into the loop space is indeed used in the computation of other shared variables. In essence, we are concerned with the flow of the values stored rather than with their storage (addresses). We note

2. *sum* returns the number of nonzero elements in A_w.
3. *any* returns the "OR" of its vector operand's elements, i.e., $any(v[1 : n]) = (v[1] \vee v[2] \vee \ldots \vee v[n])$.
4. $v[1] \wedge v[2] = TRUE$ if $(v[1] \neq 0)$ *AND* $(v[2] \neq 0)$.



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

(d)

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

(e)

Fig. 3. (a) The transformation of a **do** loop, (b) using the original version of the PD test, and (c) the lazy version. The **markwrite** (**markread**) operation marks the indicated element in the shadow array A_w (A_r and A_{np}) according to the criteria given in Steps 1a and 1b of the LPD test. Since dynamic dead read references are not marked in the LPD test, the array **A** fails the PD test and passes the LPD test, as shown in (d) and (e), respectively.

that if the search for the actual use of a read value becomes too complex, then it can be stopped gracefully at a certain depth and a conservative marking of the shadow array can be inserted (on all the paths leading to a possible use).

In the PD test described in detail in [32], *all* memory references to the array under test are marked in the same control block in which they appear, regardless whether the values stored at those addresses contribute to the global data flow or not. The PD test is, in essence, equivalent to static data dependence analysis (otherwise, it is identical to the LPD test). As can be observed from the example in Fig. 3, the "lazy" marking employed by the LPD test, i.e., the dynamic dead reference elimination technique, allows it to qualify more loops for parallel execution than the more conservative PD test. In particular, after marking and counting we obtain the results depicted in the tables. The loop fails the PD test since $A_w(:) \wedge A_r(:)$ is not zero everywhere (Step 2b). However, the loop passes the LPD test as $A_w(:) \wedge A_r(:)$ is zero everywhere, but only after privatization, since $tw(A) \neq tm(A)$ and $A_w(:) \wedge A_{np}(:)$ is zero everywhere.

Private shadow structures. The LPD test can take advantage of the processors' private memories by using private shadow structures for the marking phase of the test. Then, at the conclusion of the private marking phase, the contents of the private shadow structures are merged into the global shadow structures. Note that since the order of the writes (marks) to an element of the shadow structure is not important, all processors can transfer their private shadow structures to the global structure without synchronization. In fact, using private shadow structures enables some additional optimization of the LPD test as follows: Since the shadow structures are private to each processor, the iteration number can be used as the "mark." In this way, no reinitialization of the shadow structures is required between successive iterations, and checks such as "has this element been written in this iteration?" simply require checking if the corresponding element in A_w is marked with the iteration number. Another benefit of the iteration number "marks" is that they can double as time-stamps, which are needed for performing the last-value assignment to any shared variables that are live after loop termination.

An example using iteration numbers as "marks" in private shadow arrays is shown in Fig. 4. If the speculative execution of the loop passes the analysis phase, then the scalar reduction results are computed by performing a reduction across the processors using the processors' partial results. Otherwise, if the test fails, the loop is reexecuted sequentially.

The shadow structures can have various implementations, depending on the nature of the access pattern. A dense access pattern will benefit from shadow *arrays*, which provide random access and efficient use of memory space. Note that the state information used by the run-time test, A_w, A_r, A_{np}, can be represented with only two fields since there are at most four distinct states per shadowed element (*read-only*, *write-first*, *read-first*, *not-referenced*).On the other hand, a sparse access into a large program data structure may benefit from shadow *hash tables*. Instead of "marking" into an array we insert the "marks" into private hash tables which will be merged during the analysis phase. Hash tables will compact a sparse region into tight storage and maintain the benefit of random access (constant access time), albeit at a higher cost per reference.

3.1.1 Processor-Wise Version of the LPD Test

The LPD Test determines whether a loop has any cross-*iteration* data dependences. It turns out that essentially the same method can be used to test whether the loop, as executed, has any cross-*processor* data dependences [1]. The only difference is that all checks in the test refer to processors rather than to iterations, i.e., replace "iteration" by "processor" in the description of the LPD test so that all iterations assigned to a processor are considered as one "super-iteration" by the test. It is important to note that a loop that is not fully parallel (it has cross-iteration dependences) could potentially

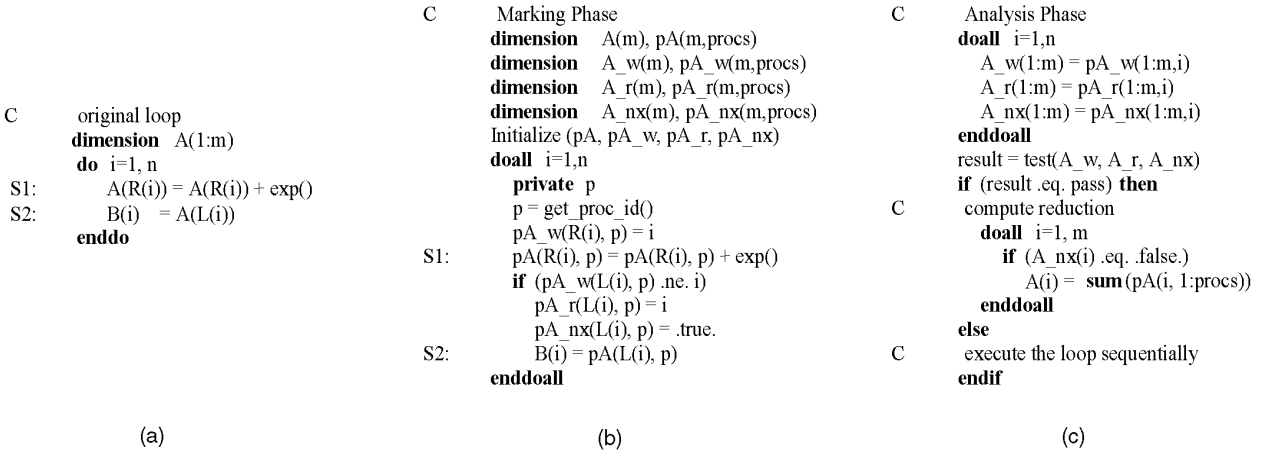


Fig. 4. The simplified code generated for the **do** loop in (a) is shown in (b) and (c). Privatization is not tested because of a read before a write reference. The **markredux** operations are as described in Fig. 5.

pass the processor-wise version of the LPD test because data dependences among iterations assigned to the same processor will not be detected. This is desirable (and correct) provided that each processor executes its assigned iterations in increasing order. The processor-wise version of the test can therefore parallelize more loops and, at the same time, incur less time and space costs: The shadow structures need to be initialized only once and can use Boolean values (2 bits per shadowed memory element) for marking. When last value assignment is required, then last written value needs to be copied out from the privatized array to the original shared array. For balanced loops, where static scheduling performs well, the needed last-write-timestamps are expressed implicitly in the value of the processor identifier.

3.1.2 Supporting Copy-In of External Values

Suppose that a loop is determined as fully parallel by the LPD test except for the accesses to one element *a*. If the first times(s) *a* is accessed by the loop through a read, and for every later iteration that accesses *a* it is always written before it is read, then the loop could actually be executed as a **doall** by having the initial accesses to *a copy-in* the global value of *a*, and having the iterations that wrote *a* use private copies of *a*. In this way loops with a $(Read)^*(Write/Read)^*$ access pattern can be safely transformed into a **doall**. The LPD test can be augmented to detect this situation by keeping track of the maximum iteration i_r^+ that read *a* (before it was ever written), and the minimum iteration i_w^- that wrote *a*. Then, if $i_r^+ \le i_w^-$, the loop can be executed in parallel. In order to collect this information, we need two additional private shadow structures, which are merged, pairwise, into the global shadow structure.

In the processor wise LPD test, these additional shadow structures are not necessary because the information is available implicitly through static scheduling. If the iteration space is assigned to the processors in contiguous chunks, i.e., processor *i* gets iterations $(n/p) * i$ through $(n/p) * (i+1) - 1$, $0 \le i < p$, then we need only check that the first write to *a* appears on a processor with an *id* that is not less than the last processor *id* in which *a* is marked as nonprivatizable or read-only.

3.2 Run-Time Reduction Parallelization

As mentioned in Section 2, there are two tasks required for reduction parallelization: recognizing the reduction variable and parallelizing the reduction operation. Of these, we focus our attention on the former since, as previously noted, techniques are known for performing reduction operations in parallel. So far, the problem of reduction variable recognition has been handled at compile-time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to validate it as a reduction variable [50]. There are two major shortcomings of such pattern matching identification methods.

1) The data dependence analysis necessary to qualify a statement as a reduction cannot be performed statically in the presence of input-dependent access patterns.
2) Syntactic pattern matching cannot identify all potential reduction variables (e.g., in the presence of subscripted subscripts).

Below, we show how each of these two difficulties can be overcome with a combination of static and run-time methods.

3.2.1 The LRPD Test: Extending the LPD Test for Reduction Validation

In this section, we consider the problem of verifying that a statement is a reduction using run-time data dependence analysis. The potential reduction statement is assumed to syntactically pattern match the generic reduction template $x = x \otimes exp$; reduction statements that do not meet this criterion are treated in the next section. To verify that such a statement is a reduction, we need to check that the reduction variable *x* satisfies the definition given in Section 2, i.e., that *x* is only accessed in the reduction statement, and that it does not appear in *exp*.

Our basic strategy is to extend the LPD test to check all statically unverifiable reduction conditions. We first consider how the test would be augmented to check only that the reduction variable is not accessed outside the single reduction statement. This situation could arise if the reduction variable is an array element whose subscript expressions are not statically analyzable. For example, although

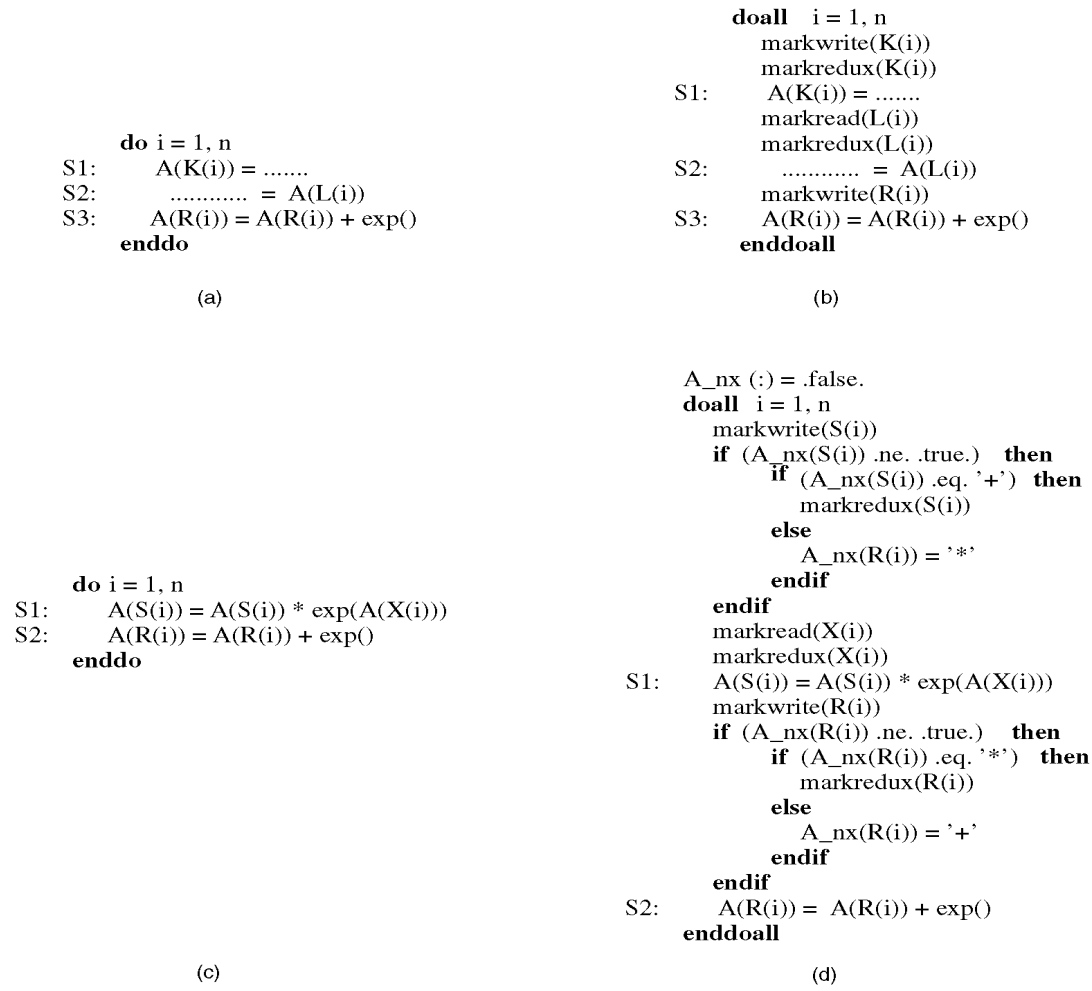


Fig. 5. The transformation of the **do** loops in (a) and (c) is shown in (b) and (d), respectively. The **markwrite** (**markread**) operation marks the indicated element in the shadow array A_w (A_r and A_{np}) according to the criteria given in Steps 1a and 1b of the LPD test. The **markredux** operation sets the shadow array element of **A_nx** to true. In (d), the type of the reduction is tested by storing the operator in **A_nx**. **A_nx** can be **true, false, +, ***.

statement **S3** in the loop in Fig. 5a matches a reduction statement, it is still necessary to prove that the elements of array **A** referenced in **S1** and **S2** do not overlap with those accessed in statement **S3**, i.e., that: $K(i) \neq R(j)$ and $L(i) \neq R(j)$, for all $1 \leq i, j \leq n$. Thus, the LRPD test must check at runtime that there is no intersection between the references in **S3** and those in **S1** and/or **S2**; in addition, it will be used to prove, as before, that any cross-iteration dependences in **S1** and **S2** are removed by privatization. To test this new condition, we use another shadow array A_{nx} to flag the array elements that are not valid reduction variables. Initially, all array elements are assumed to be valid reduction variables, i.e., $A_{nx}[:]$ = *.false.*. In the marking phase of the test, i.e., during the speculative parallel execution of the loop, any array element defined or used outside the reduction statement is invalidated as a reduction variable, i.e., its corresponding element in A_{nx} is set to true. As before, after the speculative parallel execution, the analysis phase of the test is performed. An element of A is a valid reduction variable if and only if it was not invalidated during the marking phase, i.e., it was not marked in A_{nx} as not a reduction variable for any iteration. The other shadow arrays A_{np}, A_w, and A_r are initialized, marked, and interpreted just as before.

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

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

1(d). Definitions *and* uses: If a reference to A *is not* one of the two known references to the reduction variable (i.e., it is outside the reduction statement or it is contained in *exp*), then set the corresponding element of A_{nx} to true (to indicate that the element is *not* a reduction variable). (See Figs. 5a and 5b.)

In the Analysis Phase, Steps 2d and 2e are replaced by the following:

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

(perhaps different) in which an element of A is not a reduction variable and in which it is used (read) and subsequently modified.)

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

If the analysis phase validates (passes) the speculative parallel execution of the loop, then, as before, the last-value assignments are performed for any live shared variables, and the scalar result of each reduction is computed using the processors' partial results in a reduction across the processors. (See Fig. 4.) (If reductions are implemented by placing the reduction statements in critical sections, then this last step is not necessary.)

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

In Figs. 5c and 5d, we show how a loop containing two potential reduction statements with different operators and an *exp* operand that contains references to the array under test can be transformed to perform a run-time dependence and reduction test. The subsequent analysis of the shadow arrays will detect which elements were used in a reduction and which are privatizable or read-only. If any element is found not to belong to one of these categories, then the speculative parallelization was incorrect and a sequential reexecution must be initiated.

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

3.2.2 Static Reduction Recognition and Run-Time Check

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

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

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

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

The situation is a bit more complex when the RHS contains multiple references to the array A. Consider the statement $A(R(i)) = A(X_1(i)) + A(X_2(i)) + \ldots + A(X_k(i))$. This statement is a reduction if and only if $R(i) = X_j(i)$ for exactly *one* value of j (see Section 2). As the operation is commutative and associative, we cannot discount the possibility of a reduction. In this example, we must check for equality between $R(i)$ and *every* $X_j(i)$, $1 \le j \le k$. If this equality condition is not met exactly once, then $A_{nx}(R(i))$ is set to true (to indicate it was not a reduction). We note that a more aggressive strategy could be taken when there are multiple references to $A(R(i))$ on the RHS: Promote the "+" reduction to a "*" reduction. However, as mentioned in Section 3.2.1, the reduction type can only be promoted once in the entire loop. Fig. 6 shows the code generated for run-time validation when the RHS contains multiple references to A. In the interest of clarity, reduction type promotion is not shown.

Multiple Statement Reduction Recognition: Expanded Reduction Statements. We now relax all restrictions on the RHS and allow in it variables that are neither explicit functions of the array appearing on the LHS nor explicit loop invariants. Our goal is to uncover any possible link between the LHS and the RHS, if indeed one exists. The general strategy of our methods is a fairly straightforward demand-driven forward substitution of all the variables on the RHS, a process by which all control flow dependences are substituted by data dependences as described in [2], [40]. Once this expression of the RHS is obtained it can be analyzed and validated by the methods described in the previous section. In the following, we explain by way of example how our new method can identify reductions.

In Fig. 7a, statement `S3` is first labeled at compile time as a potential reduction. Then, by following the *def-use chains* of the variables on the RHS (i.e., z and y) within the scope

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

(a)

```
function checkequal(x, y, ct)
if (x .ne. y) then
    markredux(x)
else
    ct = ct + 1
endif
return
end
```

(b)

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

(c)

Fig. 6. The code generated for the **do** loop in (a) is shown in (c). In (c), the procedure in (b) is called. The **markredux** operations are as described in Fig. 5.

of the loop we find that in statement **S1**, *z* may potentially carry the value of $A(R(i))$, while *y* is a constant with respect to *A*. The algorithm then *examines* statement **S3** after forward substitution, but does not actually replace **S3** in the generated code. The substitution is done only for compiler analysis purposes. This new version of **S3**, referred to as **S3'**, is of the form: *S3′*: $A(R(i)) = A(K(i)) + constant$. Similarly, **S5** becomes *S5′*: $A(L(i)) = A(K(i)) + constant$. Next, we label the statement pairs (**S1**, **S3**) and (**S1**, **S5**) in the original loop as *expanded reduction statements* (ERSs). If we treat each ERS as a single reduction statement, then this problem is reduced to the one treated above.

The code generated for the run time marking of the ERS is inserted for both sides of an assignment statement (RHS and LHS), but only in the same basic block where

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

(a)

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

(b)

Fig. 7. The code generated for the **do** loop in (a) is shown in (b). The **markredux** operations are as described in Fig. 5.

the assignment is. As we will see in a later example, this rule insures that both sides are marked when and if there is an assignment, i.e., it insures that a value is actually passed from the RHS to LHS. Any uses of values participating in the reduction that occur outside the ERS invalidate the ERS, i.e., set the corresponding element of the shadow array A_{nx} to true. In the case of ERSs obtained through forward substitution, the value of the reduction reference may pass through several memory locations (intermediate variables) before reaching the statement of the LHS. As any use of an intermediate variable represents a use of a value that participates in the reduction, it invalidates the reduction for the corresponding element of *A*. The uses can be obtained by following the def-use chain within the scope of the loop. However, based on the dead reference elimination principle described in Section 3.1, only references that are not dead will be processed (marked). In Fig. 7a, statement **S4** passes the value of $A(K(i))$ to the local variable *t*, which in turn passes it to $A(L(i))$ in **S5**. The same value is also passed to the shared variable $B(f(i))$ in **S6**. Both uses (in **S5** and **S6**) should, in principle, invalidate $A_{nx}(K(i))$. On the other hand,

```
        do  i = 1, n
S1:        w = A(M(i))
S2:        t = A(J(i))
S3:        if   (B1) then
S4:             z = A(K(i))
           else
S5:             z = A(L(i))
           endif
S6:        if   (B2) t = z
S7:        if   (B3) w = t
S8:        if   (B4) A(R(i)) = A(R(i)) + z
S9:        if   (B5) Y(i) = w
        enddo
```

(a)

```
        doall  i = 1, n
S1:        w = A(M(i))
S2:        t = A(J(i))
S3:        if   (B1) then
S4:             z = A(K(i))
           else
S5:             z = A(L(i))
           endif
S6:        if   (B2) t = z
S7:        if   (B3) w = t
S8:        if   (B4) then
              markread(B1*K(i) + notB1*L(i))
              markredux(B1*K(i) + notB1*L(i))
              markwrite(R(i))
              A(R(i)) = A(R(i)) + z
           endif
S9:        if   (B5) then
              markread(B3*B2*B1*K(i) + B3*B2*notB1*L(i)
                       + B3*notB2*J(i) + notB3*M(i))
              markredux(B3*B2*B1*K(i) + B3*B2*notB1*L(i)
                       + B3*notB2*J(i) + notB3*M(i))
              Y(i) = w
           endif
        enddoall
```

(b)

Fig. 8. The code generated for the **do** loop in (a) is shown in (b). The **markredux** operations are as described in Fig. 5. The expressions in the **markread** and **markredux** operations are abbreviations of **if then else** statements representing the different assignments to **z** (**S8**) and **w** (**S9**) as in (1). The operators "*," "+," and "**not**" represent logical "and," "or," and "complement" operators, respectively.

statement **S5** is another potential reduction of the same type as in **S3** and, thus, only the use in **S6** invalidates $A_{nx}(K(i))$. The transformed code is shown in Fig. 7b.

We note that if one of the intermediate variables is itself an array element addressed indirectly, then an additional run-time test must be performed. For example, if **S1** and **S3** in Fig. 7a were of the form: $S1$: $X(N(i)) = A(K(i))$ and $S3$: $A(R(i)) = X(P(i)) + y$, then an additional test would be needed to check if $N(i) = P(i)$, i.e., if a value is passed form **S1** to **S3**. However, if the array X is privatizable, and occurs *only* in these two statements, then this additional run-time test is not necessary, i.e., if $N(i) = P(i)$, then the marking of the read of $A(K(i))$ will be performed when $X(P(i))$ is used in **S3**.

Taking control flow into account. The final situation we consider is when the forward substitution procedure must take into account conditional branches and carry information into the ERS (see Fig. 8). The additional difficulty presented by this case is the fact that the exact form of the RHS is not known statically. What is known, however, is the set of all possible RHS forms, which can be computed by following all potential paths in the control flow graph. A direct approach uses a *gated static single assignment* (GSSA) [6], [41] representation of the program. In such a representation, scalar variables are assigned only once. At the points of confluence of conditional branches a ϕ function of the form $\phi(B, X_1, X_2)$ is used (in the GSSA representation) to select one of the two possible definitions of a variable (X_1 or X_2), depending on the Boolean expression B. By proceeding backwards through the def-use chains (which include the ϕ functions) it is easy to expand a scalar variable in terms of Boolean expressions, other scalar variables, and array elements. In the example of Fig. 8, the variable w in statement **S9** would be expanded as follows:

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

which means that the value of w is:

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

This compound equation can then be used to generate a **markread** and a **markredux** operation at statement S9 where w is read. To save unnecessary work, we only expand those scalars that are on the RHS of assignments to shared variables or in potential reduction statements (e.g., in the case of z in statement **S8**). All other scalar references can be safely ignored. Fig. 8b shows the program in Fig. 8a after the insertion of the **markread** and **markredux** operations, which are based on the expansion of the scalar variables. The possible drawback of this approach is that the number of potential reductions and the number of terms in the logic expressions generated may be quite large. If this happens, we can gracefully degrade to a more conservative approach: Test only some of the expressions of the ERS and invalidate all the rest.

It is important to note that the loop in Fig. 8 exemplifies the type of loop found in the SPICE2G6 program (subroutine LOAD) which can account for 70 percent of

the sequential execution time (Its vectorization has been dealt with before in [42]).

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

3.3 Complexity of the LRPD Test

The time required by the LRPD test is $T(n, s, a, p) = O(s + na/p + \log p)$, where p is the number of processors, n is the total iteration count of the loop, s is the number of elements in the shared array(s), and a is the (maximum) number of accesses to the shared array in a single iteration of the loop. We assume that the implementation of the test uses private shadow structures, that all accesses to the shared array(s) are tested, and that each processor is assigned $O(n/p)$ iterations. The analysis below is valid for all variants of the LRPD test. As mentioned later, the implementation of shadow structures with hash tables can reduce the complexity of the LRPD test to $O(na/p + \log p)$.

The marking phase (Step 1) takes $O(na/p + s + \log p)$ time. We record the read and write accesses and the reduction and privatization flags in private shadow arrays using iteration number "marks." In order to check whether for a read of an element there is a write in the same iteration, we simply check that element in the shadow array—a constant time operation. All accesses can be processed in $O(na/p)$ time, since each processor will be responsible for $O(na/p)$ accesses. After all accesses have been marked in private storage, the private shadow arrays can be merged into the global shadow arrays in $O(s + \log p)$ time; the $\log p$ contribution arises from the possible write conflicts in global storage that could be resolved using software or hardware combining. The counting in Step 2a can be done in parallel by giving each processor s/p values to add within its private memory, and then summing the p resulting values in global storage, which takes $O(s/p + \log p)$ time [21]. The comparisons in Step 2b (2d) of A_w with A_r (with A_{np} and A_{nx}) take $O(s/p + \log p)$ time.

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

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

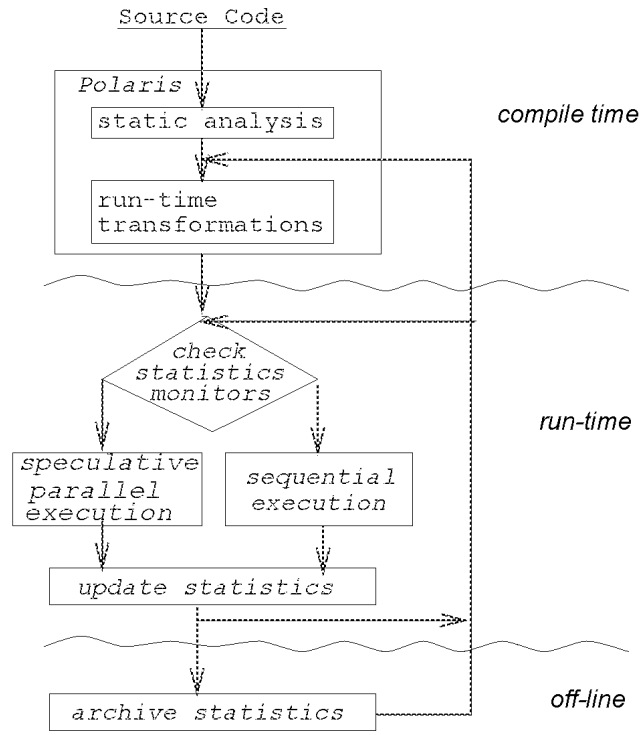


Fig. 9. Adaptive framework.

4 Putting It All Together

In the previous sections, we described run-time techniques that can be used for the speculative parallelization of loops. These techniques are automatable and a good compiler could easily insert them in the original code. In this section, we give a brief outline of how a compiler might proceed when presented with a `do` loop whose access pattern cannot be statically determined. In Fig. 2, we present a global view of how the run-time parallelization technology interacts with the compilation and execution system. Fig. 9 shows how the run-time system can be integrated into an adaptive feedback loop. We further present a few general optimization strategies.

4.1 At Compile Time

1) A cost/benefit analysis is performed using both static analysis (based on the asymptotic complexity of the LPRD test given below) and run-time collected statistics to determine whether the loop should be:

a) Speculatively executed in parallel using the LRPD test,
b) First tested for full parallelism, and then executed appropriately (using an inspector/executor version of the LRPD Test), or
c) Executed sequentially.

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

4.2 At Run-Time

1) Checkpoint if necessary, i.e., save the state of program variables.
2) Execute the parallel version of the loop, which includes the marking phase of the test.
3) Execute the analysis phase of the test, which gives the pass/fail result of the test.
4) If the test passed, then compute the final results of all reduction operations (from the processors' partial results) and copy-out the values of any live private variables. If the test failed, then restore the values of any altered program variables and execute the sequential version of the loop.
5) Collect statistics for use in future runs, and/or for schedule reuse in this run.

4.2.1 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 the 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. As shown in Fig. 9, the compiler and run-time system form a feedback loop. Statistical information about parallelism is collected at run-time (after every run-time test) and used at different levels of the adaptive system. Based on the outcome of the previous n instantiations of a loop, the system can decide (through compiler generated code execution) to either continue to speculate that the loop is parallel or execute sequentially. The same information can be stored in a file at the end of the program, processed further, and read-in during the next run of the program. Moreover, it can be used as input to any recompilation. For example, if a loop is never found to be parallel a recompilation will generate only sequential code for it. In addition, user directives about the parallelism of the loop might prove useful.

In the remainder of this section, we give an approximate estimate of the potential speedup and slowdown for run-time parallelization using the LRPD test. While a more accurate analysis could prove very useful in predicting actual speedups, our analysis is only intended to be used by a compiler when deciding whether the application of the LRPD test can be potentially profitable. Given a fully parallel loop L, the *ideal speedup*, Sp_{id}, is the ratio between its sequential and its parallel execution times, T_{seq} and T_{doall}, respectively. However, when L is parallelized using 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.

$$Sp_{id} = \frac{T_{seq}}{T_{doall}}$$

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

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 3.3 predicts that $T_{mark} = T_{analysis} = O(na/p + \log p)$, 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 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. It is important to note that the instructions added for marking the references under test are almost independent of the activity of the loop itself. Thus, marking introduces an almost independent thread of execution that can be executed concurrently with the loop and which can be exploited by multiple issue microprocessors. The net result may be a significant reduction of the impact of reference marking on the overall execution time of the speculative loop.

Note that in the worst possible case $T_{mark} \approx T_{analysis} \approx T_{doall}$, and even then the attainable speedup predicted for the **doall** test is $\approx \frac{1}{3} Sp_{id}$. Although 33 percent 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 the *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

TABLE 1
SUMMARY OF EXPERIMENTAL RESULTS

Benchmark[2] Subroutine Loop	Experimental Results Technique	Speedup	potential Slowdown	Tested	Description of Loop (% of sequential execution time of program)	Inspector (computation)
MDG	14 processors					privatization
INTERF	speculative	11.55	1.09	doall	accesses to a privatizable vector guarded	data accesses
loop 1000	insp/exec	8.77	1.03	privat	by loop computed predicates (92% T_{seq})	branch predicate
BDNA	14 processors					privatization
ACTFOR	speculative	10.65	1.09	doall	accesses privatizable array indexed by a	data accesses
loop 240	insp/exec	7.72	1.04	privat	subscript array computed inside loop (32% T_{seq})	subscript array
	8 processors					
TRFD	speculative	.85	2.17	doall	small triangular loop accesses a vector	data accesses
INTGRL	sched reuse	1.93	2.17		indexed by a subscript array computed	replicates loop
loop 140	insp/exec	1.05	1.74		outside loop (5% T_{seq})	
	sched reuse	2.10	1.74			
TRACK					accesses array indexed by subscript array	
NLFILT	8 processors			doall	computed outside loop, access pattern	not applicable
loop 300	speculative	4.21	1.01		guarded by loop computed predicates (39% T_{seq})	
ADM					accesses privatizable array thru aliases,	
RUN	14 processors			doall	array re–dimensioned, access	not applicable
loop 20	speculative	9.01	1.02	privat	pattern control flow dependent (44% T_{seq})	
OCEAN	8 processors				kernel–like loop accesses a vector with	data accesses
FTRVMT	speculative	2.23	1.45	doall	run–time determined strides	replicates loop
loop 109	insp/exec	2.14	1.30		26K invocations account for 43% T_{seq}	
SPICE					traverses linked list terminated by a NULL	
LOAD	8 processors			doall	pointer, all referenced arrays equivalenced	data accesses
loop 40	insp/exec	2.75	1.09	reduct	to a global work array	

All benchmarks are from the PERFECT Benchmark Suite.

phase is work-equivalent to the loop. Thus, the cost of performing a failed test is proportional to $\frac{1}{p}T_{seq}$:

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

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 determines 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). In practice, when T_{mark} approaches T_{doall}, speculative use of the tests may be beneficial. The overhead needed to save/restore state must also be considered when deciding whether to use speculative parallel execution.

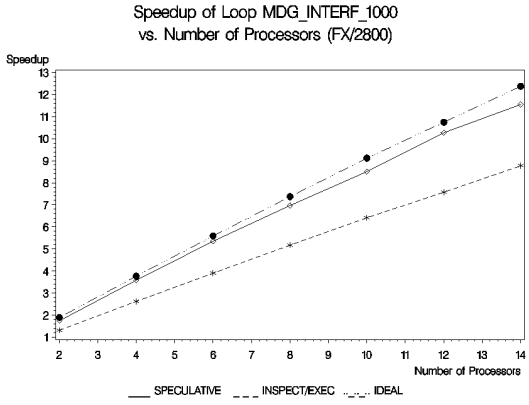
4.2.2 Schedule Reuse

Thus far, we have assumed that a `doall` test is run *each* time a loop is executed in order to determine if the 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., Saltz et al. [37]). 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. We remark that most programs are of a repetitive nature and, thus, there exists the potential for schedule reuse. A simple example in which schedule reuse could be considered is for multiply nested loops. If possible, it is generally best to parallelize the outer loop in the nesting. However, if this is not possible, then it may be the case that schedule reuse could be attempted when parallelizing the inner loops.

5 EXPERIMENTAL RESULTS

In this section, we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [4]) and 14 processors (Alliant FX/2800 [5]) using a Fortran implementation of our run-time library. The codes were manually instrumented with calls to the run-time library. However, we remark that our results scale with the number of processors and the data size and thus can therefore be extrapolated for massively parallel processors (MPPs).

We considered seven `do` loops from the PERFECT Benchmarks [8] that could not be parallelized by any compiler available to us. Our results are summarized in Table 1. We have applied the LRPD test in both speculative and inspector/executor mode (with the notable exception of TRACK). For the speculative version we have augmented the loop with marking code and calls to initialization/checkpointing and analysis routines before and, respectively, after the loop. For the inspector/executor experiments, we have extracted a *parallel* inspector loop, i.e., a loop that *only* computes and traverses the original access pattern without modifying the shared arrays. Because an inspector is side effect free, no checkpointing was necessary. We then applied the LRPD test to the parallel inspector loop in the same manner as we have applied it in the speculative version. If the test passed, the original loop was



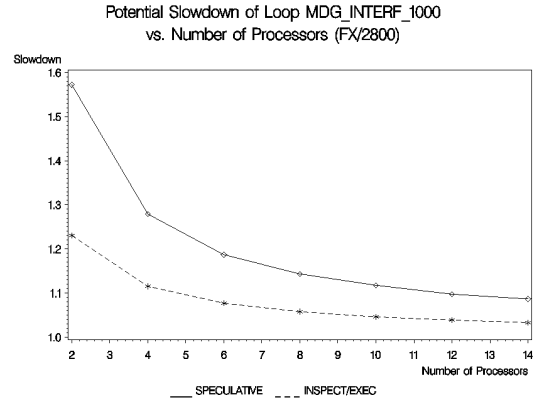
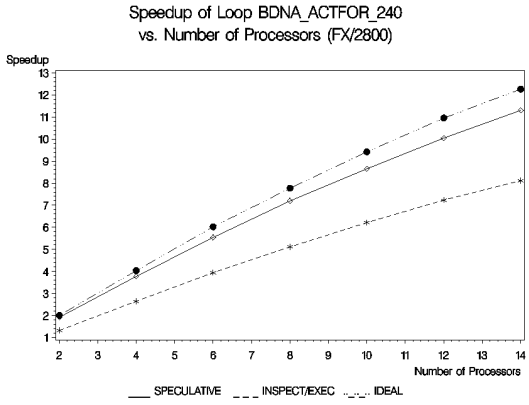


Fig. 10.



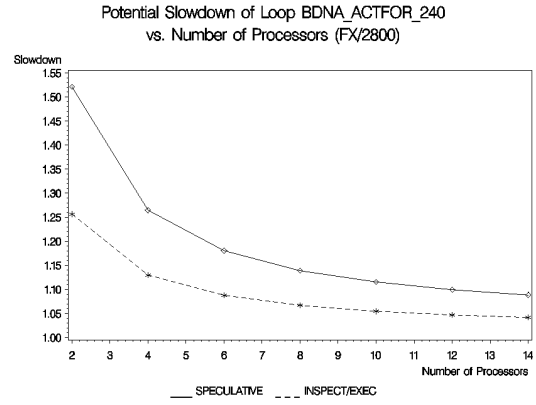


Fig. 11.

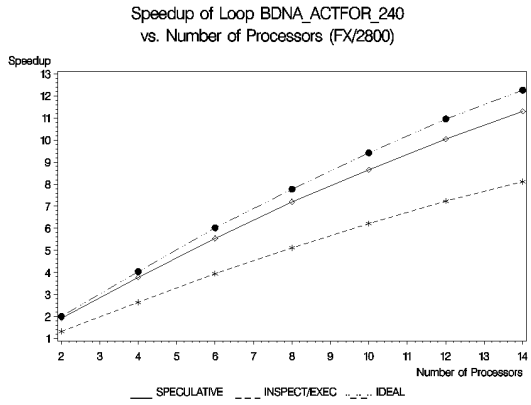
then executed in parallel (the executor), otherwise, it was executed sequentially.

For both experiments, a serial version of the loop is provided for the case when the LRPD test fails. For each loop, we note the type of test applied: *doall* indicates cross-iteration dependences were checked (Lazy Doall (LD) test), *privat* indicates privatization was checked (LPD test), *reduct* indicates reduction parallelization was checked (LRD test). For each method applied to a loop, we give the speedup that was obtained and the potential slowdown that *would have been incurred* if, after applying the method, the loop had to be reexecuted sequentially. If the inspector/executor version of the LRPD test was applied, the computation performed by the inspector is shown in the table: The notation *privatization* indicates the inspector verified that the shared array was privatizable and then dynamically privatized the array for the parallel execution, *branch predicate* and *subscript array* mean that the inspector computed these values, and *replicates loop* means that the inspector was work-equivalent to the original loop.

In addition to the summary of results given in Table 1, we show in Figs. 10 through 16 the speedup and the *potential* slowdown measured for each loop as a function of the number of processors used. For reference, these graphs show the ideal speedup, which was calculated using an optimally parallelized (by hand) version of the loop. The potential slowdown reported is the percentage of the execution time that would be paid as a penalty if the test had failed and the loop was then executed sequentially. In cases where extraction of a reduced inspector loop was impractical because of complex control flow and/or interprocedural problems, we only applied the speculative methods. Note that the reported slowdowns are a worst case scenario in which a dependence is found at the end of the analysis phase and not earlier during the parallel speculative execution of the loop (in practice, as soon as a dependence is found, speculative execution is halted).

Whenever necessary in the speculative executions, we performed a simple preventive backup of the variables potentially written in the loop. In some cases, the cost of saving/restoring might be significantly reduced by using another strategy. In order for our methods to scale with the number of processors, the shadow arrays must be distributed over the processor space, rather than replicated on each processor (Section 3.3). 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 SPICE and TRFD Benchmarks. For these reasons the best results (which are reported here) were obtained using shadow arrays. However, on a larger machine we would expect the use of hash tables to generate more scalable speedups than those shown here.

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



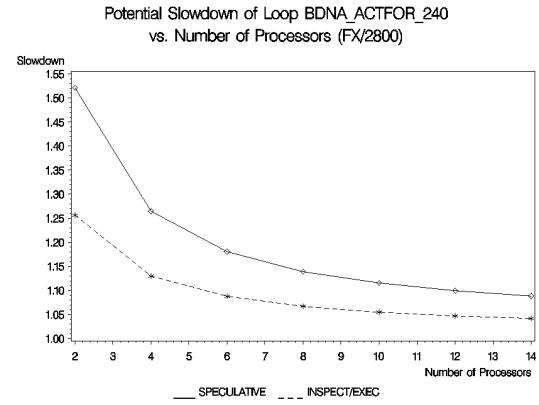
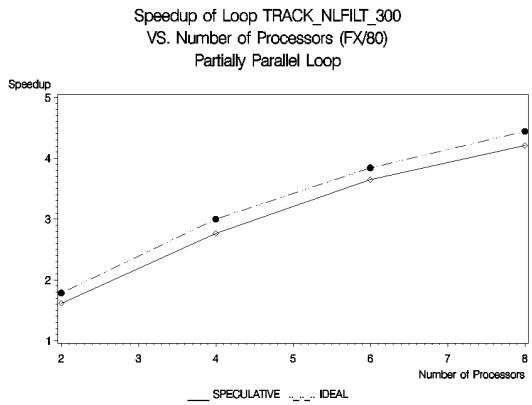


Fig. 12.



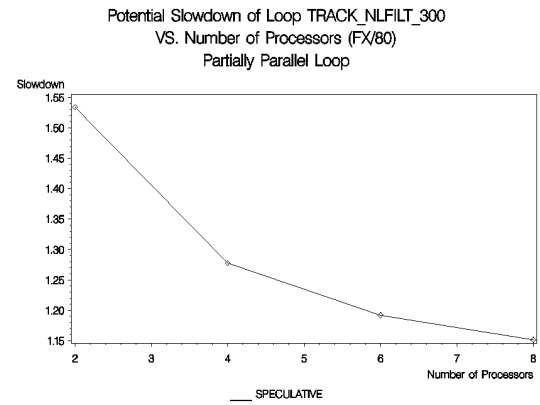
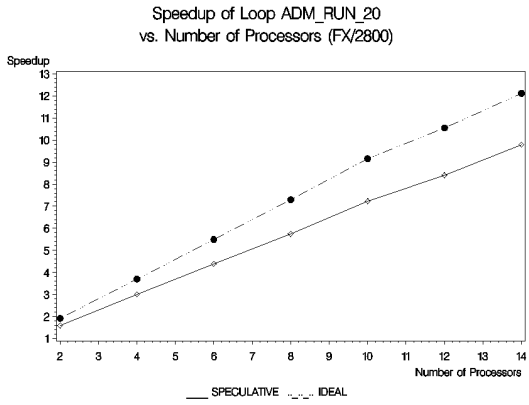


Fig. 13.



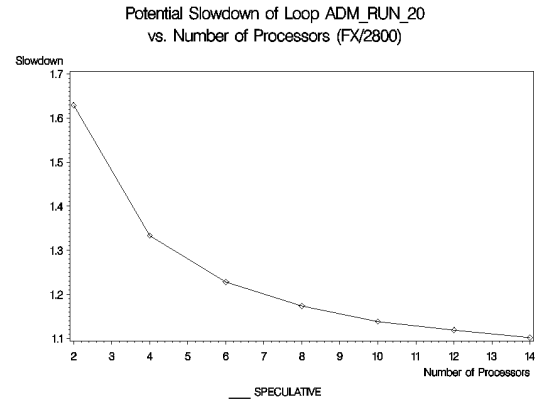


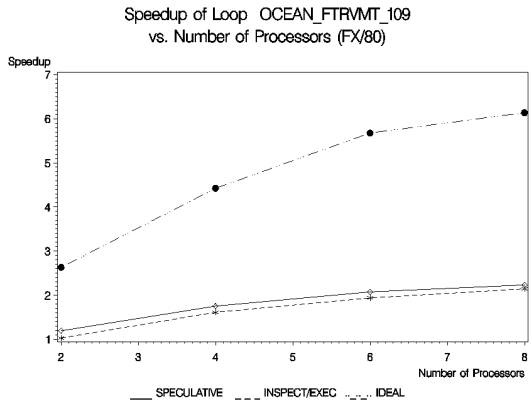
Fig. 14.

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. 12), the speculative strategy gives superior speedups versus the inspector/executor method. For both methods the potential slowdown is small, and decreases as the number of processors increases. As expected, the potential slowdown is smaller for the inspector/executor method.

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

The loop from TRACK is parallel for only 90 percent of its invocations. In the cases when the test failed, we restored state, and reexecuted the loop sequentially. The speedup reported includes both the parallel and sequential instantiations (Fig. 13).

Loop 40 from SPICE is representative of the type of the loop contained in the LOAD subroutine, which accounts for 70 percent of the sequential execution time. Since all the arrays are equivalenced to a global work array, all accesses in the loop were shadowed in the LRD test, i.e., each array element was proven to be either a reduction variable, read-only, or independent (i.e., accessed in only one iteration). For this loop, we used an inspector/executor version of the LRD test (instead of a speculative parallelization) because



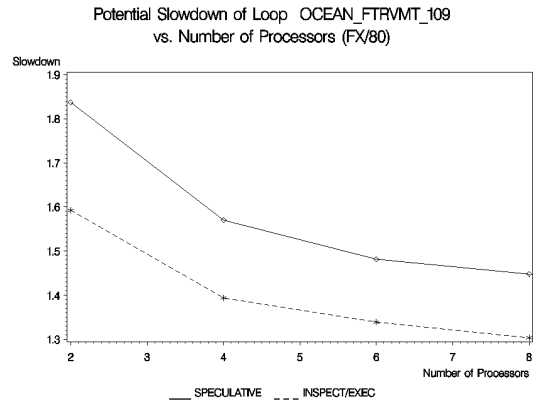
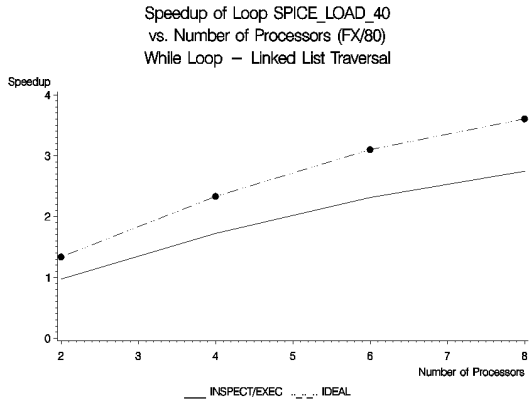


Fig. 15.



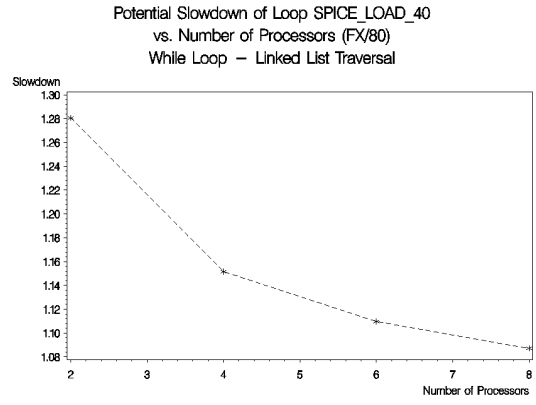


Fig. 16.

of complex memory management problems for the shadow arrays in the presence of highly irregular and sparse access patterns. The ideal speedup of loop 40 is not very large since the loop is small, imbalanced between iterations, and traverses a linked list. The linked list traversal was parallelized using techniques we developed for automatically parallelizing `while` loops [33]. Thus, although the obtained speedup is modest, it represents a significant fraction of the ideal speedup (see Fig. 16). Therefore, since loop 40 is one of the smallest loops in the LOAD subroutine, we expect to obtain better speedups on the larger loops (since they have larger ideal speedups).

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

6 Previous Run-Time Techniques for Parallelizing Loops

As mentioned in Section 1, there has been some work on developing techniques for the run-time analysis and scheduling of loops with cross-iteration dependences. Note that this is a more general problem than the one studied in this paper since the construction of a valid parallel execution schedule requires *finding and analyzing all* cross-iteration dependences in the loop, whereas we have been concerned with simply *detecting the presence of any* dependences that prohibit parallelization. Therefore, these techniques are necessarily more complex than the PD test and, in almost all cases, use global synchronization primitives, make conservative assumptions regarding cross-iteration dependences, have significant sequential components, and/or do not find an optimal parallel execution schedule for the iterations of the loop. Most of these schemes partition the set of iterations into subsets called *stages* so that the iterations in each stage can be executed in parallel, i.e., there are no data dependences between iterations in a stage. Stages formed by a regular pattern of iterations are named wavefronts. They are executed sequentially by placing a synchronization barrier between each pair of consecutive wavefronts.

One of the first run-time methods for scheduling partially parallel loops was proposed by Zhu and Yew [49]. It computes the stages one after the other in successive phases. In a phase, an iteration is added to the current stage if it is the lowest iteration (not assigned to a previous stage) that accesses (reads or writes) any of the data (variables) used in that iteration, i.e., none of the data accessed in that iteration is accessed by any lower unassigned iteration. In each phase, the lowest unassigned iteration to access any variable (e.g., array element) is found using atomic *compare-and-swap*

synchronization primitives to record the minimum such iteration in a shadow version of that variable. By using separate shadow variables to process the read and write operations, Midkiff and Padua [27] improved this basic method so that concurrent reads from a memory location are allowed in multiple iterations. Recently, Chen et al. [13] proposed another variant of the Zhu and Yew method which improves performance in the presence of *hot-spots* (i.e., many accesses to the same memory location) by first doing some of the computation in private storage. Xu and Chaudhary [46], [45] improve upon [13] by not serializing on multiple reads to the same location. All of the above mentioned methods construct maximal stages in the sense that each iteration is placed in the earliest possible stage, giving a minimal depth schedule, i.e., a minimal number of stages.

Polychronopoulos [30] gives a method that assigns iterations to stages in a different way: Each wavefront consists of a maximal set of contiguous iterations which contain no cross-iteration dependences. It is easy to see that this method may not yield a minimum depth schedule. Like the method of Zhu and Yew and its variants, this method uses shadow versions of the variables to detect possible dependences. The wavefronts can be constructed sequentially by inspecting all the shared variable accesses, or in parallel with the aid of critical sections. Note that since all computations are performed at run-time, it is important for them to be efficiently parallelizable.

Krothapalli and Sadayappan [18] proposed a run-time scheme for removing anti (write-after-read) and output (write-after-write) dependences from loops. These types of dependences are also known as *memory-related* dependences because they arise from the reuse of storage and are not essential to the computation, as are flow (read-after-write) dependences, which express a fundamental relationship about data flow in the program. Their scheme includes a parallel preprocessing phase which uses critical sections as in the method of Zhu and Yew, to determine the number and types of accesses to each memory location. Next, for each memory location they build a flow graph, allocate any additional storage needed to remove the anti- and output dependences, and explicitly construct the mapping between all the memory accesses in the loop and the storage (both old and new). Then, the loop is executed in parallel using synchronization (locks) to enforce the flow dependences. This scheme relies heavily on synchronization, inserts an additional level of indirection into all memory accesses, and calls for dynamic shared memory allocation.

The problem of analyzing and scheduling loops at run-time has been studied extensively by Saltz et al. [9], [35], [36], [37], [44]. In most of these methods, the original source loop is transformed into an *inspector*, which performs some run-time data dependence analysis and constructs a (preliminary) schedule, and an *executor*, which performs the scheduled work. The original source loop is *assumed to have no output dependences*. In [37], the inspector constructs stages that respect the flow dependences by performing a *sequential* topological sort of the accesses in the loop. The executor enforces any anti dependences by using old and new versions of each variable. Note that the anti dependences can only be handled in this way because the original loop does not have any output dependences, i.e., each variable is written at most once in the loop. The inspector computation (the topological sort) can be parallelized somewhat using the *DOACROSS parallelization technique* of Saltz and Mirchandaney [35], in which processors are assigned iterations in a wrapped manner, and busy-waits are used to ensure that values have been produced before they are used (again, this is only possible if the original loop has no output dependences).

Recently, Leung and Zahorjan [22] have proposed some other methods of parallelizing the inspector of Saltz et al. These techniques are also restricted to loops with no output dependences. In *sectioning*, each processor computes an optimal parallel schedule for a contiguous set of iterations, and then the stages are concatenated together in the appropriate order. Thus, sectioning will usually produce a suboptimal schedule since a new synchronization barrier is introduced into the schedule for each processor. In *bootstrapping*, the inspector of Saltz et al. (i.e., the sequential topological sort) is parallelized using the sectioning method. Although bootstrapping might not optimally parallelize the inspector (due to the synchronization barriers introduced for each processor), it will produce the same minimum depth schedule as the sequential inspector of Saltz et al.

In summary, the previous run-time methods for parallelizing loops rely heavily on global synchronizations (communication) [13], [18], [22], [27], [30], [35], [37], [49], are applicable only to restricted types of loops [22], [35], [37], have significant sequential components [30], [35], [37], and/or do not extract the maximum available parallelism (they make conservative assumptions) [13], [22], [30], [35], [37], [49].

A high level comparison of the various methods is given in Table 2.

7 Related Work

7.1 Race Detection for Parallel Program Debugging

A significant amount of work has been invested in the research of hazards (race conditions) and access anomalies for debugging parallel programs. Generally, access anomaly detection techniques seek to identify the point in the parallel execution at which the access anomaly occurred. In [3], [16], the authors discuss methods that statically analyze the source program and methods that analyze an execution trace of the program. Since not all anomalies can be detected statically, and execution traces can require prohibitive amounts of memory, *run-time* access anomaly detection methods that minimize memory requirements are desirable [38], [14], [28]. In fact, a run-time anomaly detection method proposed by Snir, and optimized by Schonberg [38] and later by Nudler and Rudolph [28], bears similarities to a simplified version of the LRPD test presented in Section 3 (i.e., a version without privatization). In essence, all nodes in an arbitrary concurrency graph (a DAG) with nested *forks* and *joins* is labeled in such a way as to uniquely reflect the possibility of a race condition, i.e., concurrent read and/or write accesses to a shared variable. At execution time the race detection mechanism maintains an access history of the shared locations and is checked “on the fly” for an illegal

TABLE 2
A COMPARISON OF RUN-TIME PARALLELIZATION TECHNIQUES FOR `do` LOOPS

Method	obtains optimal schedule	contains sequential portions	requires global synchron	restricts type of loop	privatizes or finds reductions
Rauchwerger/Amato/Padua [31]	Yes	No	No	No	P,R
Zhu/Yew [49]	No[1]	No	Yes[2]	No	No
Midkiff/Padua [27]	Yes	No	Yes[2]	No	No
Krothapalli/Sadayappan [18]	No[3]	No	Yes[2]	No	P
Chen/Yew/Torrellas [13]	No[1,3]	No	Yes	No	No
Xu/Chaudary [46], [45]	Yes	No	Yes	No	No
Saltz/Mirchandaney [35]	No[3]	No	Yes	Yes[5]	No
Saltz *et al.* [37]	Yes	Yes[4]	Yes	Yes[5]	No
Leung/Zahorjan [22]	Yes	No	Yes	Yes[5]	No
Polychronopoulos [30]	No	No	No	No	No
Rauchwerger/Padua [32], [34]	No[6]	No	No	No	P,R

In the table entries, P and R show that the method identities privatizable and reduction variables, respectively. The superscripts have the following meanings: 1) the method serializes all read accesses, 2) the performance of the method can degrade significantly in the presence of hotspots, 3) the scheduler/executor is a `doacross` *loop (iterations are started in a wrapped manner) and busy waits are used to enforce certain data dependences, 4) the inspector loop sequentially traverses the access pattern, 5) the method is only applicable to loops without any output dependences (i.e., each memory location is written at most once), 6) the method only identifies fully parallel loops.*

reference sequence, i.e., a race condition. However, Schonberg's method requires a large amount of storage for recording the access history, i.e., O(T) for each monitored variable, where T is the number of dynamically concurrent threads. Viewed in the framework of the LRPD test, a separate shadow array for each *iteration* in a loop must be maintained. In 1991, Mellor-Crummey [25] improved this technique by dramatically reducing the memory requirements; the maximum access history storage is O(N), where N is the maximum level fork-join nesting. Since, in current systems at most two levels of parallelism are supported, this memory overhead seems quite manageable, making the technique more attractive to race detection. The execution time overhead is still very high, because every reference monitored has to be logged and checked against the access history in a critical section. In [26], an order of magnitude increase in execution time of instrumented codes is reported for experiments on a sequential machine. Even after reducing the shadowed references through compile time analysis, the time expansion factor remains around 5. The need for critical sections for the parallel execution of this experiment would only add to the run-time overhead of this technique. While the method may not be suitable for performance-oriented parallelization of `doall` loops, it is a clever technique for debugging arbitrary fork-join parallelism constructs.

7.2 Optimistic Execution

A concept related to the speculative approach described in this paper is *virtual time*, first introduced in [17] and defined as a "... paradigm for organizing and synchronizing distributed systems.... [It] provides a flexible abstraction of real time in much the same way that virtual memory provides an abstraction of real memory. It is implemented using the Time Warp mechanism, a synchronization protocol distinguished by its reliance on look-ahead rollback, and by its implementation of rollback via antimessages." The granularity and overhead associated with this method seem to make it more applicable to problems such as discrete event simulation and database concurrency control rather than loop parallelization. In fact, this concept has been applied in database design.

8 CONCLUSION

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

Our experimental results show that the concept of run-time data dependence checking is a useful solution for loops that cannot be analyzed sufficiently by a compiler. Both speculative and inspector/executor strategies have been shown to be viable alternatives for even modestly parallel machines like the Alliant FX/80 and 2800. We

would like to emphasize that our methods are applicable to all loops, without any restrictions on their data or control flow.

We believe that the significance of the methods presented here will only increase with the advent of massively parallel processors (MPPs) for which the penalty of not parallelizing a loop could be a massive performance degradation. As we have shown, our run-time tests are efficient and scalable, and thus if the target machine has many (hundreds) processors, then the cost of our techniques will become a very small fraction of the sequential execution time. In other words, speculating that the loop is fully parallel has the potential to offer large gains in performance (speedup), while at the same time risking only small losses. To bias the results even more in our favor, the decision on when to apply the methods should make use of run-time collected information about the fully parallel/not parallel nature of the loop. In addition, specialized hardware features could greatly reduce the overhead introduced by the methods [47].

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

ACKNOWLEDGMENTS

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

This research was supported in part by Intel and NASA Graduate Fellowships, the U.S. National Science Foundation CAREER Award CCR-9734471, and by Army contract no. DABT63-92-C-0097, and a partnership award from IBM. This work is not necessarily representative of the positions or policies of the Army or the Government. A preliminary version of this paper appeared in *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 218-232, June 1995.

REFERENCES

[1] S. Abraham, *private communication*, Hewlett Packard Laboratories, 1994.

[2] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proc. 10th ACM Symp. Principles of Programming Languages*, pp. 177–189, Jan. 1983.

[3] T. Allen and D.A. Padua, "Debugging Fortran on a Shared-Memory Machine," *Proc. 1987 Int'l Conf. Parallel Processing*, pp. 721–727, St. Charles, Ill., 1987.

[4] *FX/Series Architecture Manual*. Alliant Computer Systems Corp., 1986.

[5] *Alliant FX/2800 Series System Description*. Alliant Computer Systems Corp., 1991.

[6] R. Ballance, A. Maccabe, and K. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-Data- and Demand-Driven Interpretation of Imperative Languages," *Proc. SIGPLAN '90 Conf. Programming Language Design and Implementation*, pp. 257–271, June 1990.

[7] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, Mass.: Kluwer Academic, 1988.

[8] 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, Univ. of Illinois, Urbana-Champaign, May 1989.

[9] H. Berryman and J. Saltz, "A Manual for PARTI Runtime Primitives," Interim Report 90-13, ICASE, 1990.

[10] W. Blume and R. Eigenmann, "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks™ Programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 6, pp. 643–656, Nov. 1992.

[11] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh, "Automatic Generation of Nested, Fork-Join Parallelism," *J. Supercomputing*, pp. 71–88, 1989.

[12] W.J. Camp, S.J. Plimpton, B.A. Hendrickson, and R.W. Leland, "Massively Parallel Methods for Engineering and Science Problems," *Comm. ACM*, vol. 37, no. 4, pp. 31–41, Apr. 1994.

[13] D.K. Chen, P.C. Yew, and J. Torrellas, "An Efficient Algorithm for the Run-Time Parallelization of doacross Loops," *Proc. Supercomputing 1994*, pp. 518–527, Nov. 1994.

[14] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (*PPOPP*), pp. 1–10, 1990.

[15] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua, "Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs," *Proc. Fourth Workshop on Languages and Compilers for Parallel Computing*, pp. 65–83, Santa Clara, Calif., Aug. 1991.

[16] P.A. Emrath, S. Ghosh, and D.A. Padua, "Detecting Nondeterminacy in Parallel Programs," *IEEE Software*, pp. 69–77, Jan. 1992.

[17] D.R. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[18] V. Krothapalli and P. Sadayappan, "An Approach to Synchronization of Parallel Computing," *Proc. 1988 Int'l Conf. Supercomputing*, pp. 573–581, June 1988.

[19] C. Kruskal, "Efficient Parallel Algorithms for Graph Problems," *Proc. 1986 Int'l Conf. Parallel Processing*, pp. 869–876, Aug. 1986.

[20] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. Eighth ACM Symp. Principles of Programming Languages*, pp. 207–218, Jan. 1981.

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

[22] S. Leung and J. Zahorjan, "Improving the Performance of Runtime Parallelization," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (*PPOPP*), pp. 83–91, May 1993.

[23] Z. Li, "Array Privatization for Parallel Execution of Loops," *Proc. 19th Int'l Symp. Computer Architecture*, pp. 313–322, 1992.

[24] D.E. Maydan, S.P. Amarasinghe, and M.S. Lam, "Data Dependence and Data-Flow Analysis of Arrays," *Proc. Fourth Workshop Programming Languages and Compilers for Parallel Computing*, Aug. 1992.

[25] J. Mellor-Crummey, "On-the-Fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Proc. Supercomputing 1991*, pp. 24–33, Albuquerque, N.M., Nov. 1991.

[26] J. Mellor-Crummey, "Compile-Time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, San Diego, Calif., pp. 129–139,May 1993.

[27] S. Midkiff and D. Padua, "Compiler Algorithms for Synchronization," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,485–1,495, Dec. 1987.

[28] I. Nudler and L. Rudolph, "Tools for the Efficient Developement of Efficient Parallel Programs," *Proc. First Israeli Conf. Computer System Eng.*, 1988.

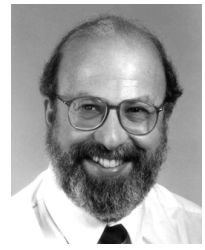
[29] D.A. Padua and M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. ACM*, vol. 29, pp. 1,184–1,201, Dec. 1986.

[30] C. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. Computers*, vol. 37, no. 8, pp. 991–1,004, Aug. 1988.

[31] L. Rauchwerger, N. Amato, and D. Padua, "A Scalable Method for Run-Time Loop Parallelization," *Int'l J. Parallel Processing*, vol. 26, no. 6, pp. 537–576, July 1995.
[32] L. Rauchwerger and D. Padua, "The Privatizing doall Test: A Run-Time Technique for doall Loop Identification and Array Privatization," *Proc. 1994 Int'l Conf. Supercomputing*, pp. 33–43, July 1994.
[33] L. Rauchwerger and D.A. Padua, "Parallelizing WHILE Loops for Multiprocessor Systems," *Proc. Ninth Int'l Parallel Processing Symp.*, Apr. 1995.
[34] L. Rauchwerger and D.A. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proc. SIGPLAN 1995 Conf. Programming Language Design and Implementation*, pp. 218–232, La Jolla, Calif., June 1995.
[35] J. Saltz and R. Mirchandaney, "The Preprocessed doacross Loop," *Proc. 1991 Int'l Conf. Parallel Processing*, Dr. H.D. Schwetman, ed., pp. 174–178. CRC Press, Inc., 1991.
[36] J. Saltz, R. Mirchandaney, and K. Crowley, "The doconsider Loop," *Proc. 1989 Int'l Conf. Supercomputing*, pp. 29–40, June 1989.
[37] J. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Trans. Computers*, vol. 40, no. 5, May 1991.
[38] E. Schonberg, "On-the-Fly Detection of Access Anomalies," *Proc. SIGPLAN 1989 Conf. Programming Language Design and Implementation*, pp. 285–297, Portland, Ore., 1989.
[39] P. Tu and D. Padua, "Array Privatization for Shared and Distributed Memory Machines," *Proc. Second Workshop Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, Sept. 1992.
[40] P. Tu and D. Padua, "Automatic Array Privatization," *Proc. Sixth Ann. Workshop on Languages and Compilers for Parallel Computing*, Portland, Ore., Aug. 1993.
[41] P. Tu and D. Padua, "GSA Based Demand-Driven Symbolic Analysis," Technical Report 1339, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Feb. 1994.
[42] A. Vladimirescu, "LSI Circuit Simulation on Vector Computers," Technical Report no. UCB/ERL M82/75, Electronics Research Lab., Univ. of California, Berkeley, Oct. 1982.
[43] M. Wolfe, *Optimizing Compilers for Supercomputers*. Boston, Mass.: The MIT Press, 1989.
[44] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman, "Runtime Compilation Methods for Multicomputers," *Proc. 1991 Int'l Conf. Parallel Processing*, Dr. H.D. Schwetman, ed., pp. 26–30. CRC Press, Inc., 1991.
[45] C. Xu, "Effects of Parallelism Degree on Runtime Parallelism of Loops," *Proc. 31st Hawaii Int'l Conf. System Sciences*, pp. 86–95, Jan. 1998.
[46] C. Xu and V. Chaudhary, "Time-Stamping Algorithms for Parallelization of Loops at Run-time," *Proc. 11th Int'l Parallel Processing Symp.*, Apr. 1997.
[47] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors," *Proc. Fourth Int'l Symp. High Performance Computer Architecture 1998* (*HPCA-4*), pp. 162–173, Feb. 1998.
[48] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture* (*HPCA-5*), Jan. 1999.
[49] C. Zhu and P.C. Yew, "A Scheme to Enforce Data Dependence on Large Multiprocessor Systems," *IEEE Trans. Software Eng.*, vol. 13, no. 6, pp. 726–739, June 1987.
[50] H. Zima, *Supercompilers for Parallel and Vector Computers*. New York: ACM Press, 1991.

Lawrence Rauchwerger received the Diploma Engineer degree in electronics and telecommunications from the Polytechnic Institute, Bucharest, Romania, the MS degree in electrical engineering from Stanford University, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He was an engineer at Varian Associates Inc., Thin Film Technology Division, Research and Development Department, Palo Alto, California, and at Beckman Instruments Inc., Scientific Instruments Division, Irvine, California. In 1995-1996, he was a visiting assistant professor at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign and a visiting scientist at AT&T Research Laboratories, Murray Hill, New Jersey. Since 1996, he has been an assistant professor in the Department of Computer Science at Texas A&M University.

His research interests include parallelizing compilers, parallel architectures, and performance evaluation and modeling. He is a member of the IEEE and the ACM.



David Padua received a PhD in computer science from the University of Illinois in 1980. Dr. Padua is a professor of computer science at the University of Illinois at Urbana-Champaign. From 1990 to 1993, he was associate director for software of the Center for Supercomputing Research and Development and is currently a member of the Science Steering Committee of the ASCI-funded Center for Simulation of Advanced Rockets. His research interests include compiler technology for parallel computers, software development tools, and parallel computer organization. He served on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*, was editor-in-chief of the *International Journal of Parallel Programming* (*IJPP*), and now serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and *IJPP*. He has served as chair of several conferences, including the 1990 and 1995 ACM Symposia on Principles and Practices of Parallel Programming. He is a coorganizer of the Workshops on Languages and Compilers for Parallel Computing, which have been held annually for the past 11 years.