Run-Time Parallelization: It's Time Has Come

Lawrence Rauchwerger

Department of Computer Science
Texas A&M University
College Station, TX 77843

Corresponding Author: Lawrence Rauchwerger telephone: (409) 845-8872
email: rwerger@cs.tamu.edu fax: (409) 847-8578

KEYWORDS: Parallelization, Speculative, Run-Time, Inspector/Executor, Compiler, Scheduling, Debugging, Pointer Aliasing, Subsripted Subscripts, Irregular Applications.

Abstract

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. This type of loop mostly occurs in irregular, dynamic applications which represent more than 50% of all applications [20]. Making parallel computing succeed has therefore become conditioned by the ability of compilers to analyze and extract the parallelism from irregular applications. In this paper we present a survey of techniques that can complement the current compiler capabilities by performing some form of data dependence analysis during program execution, when all information is available.

After describing the problem of loop parallelization and its difficulties, a general overview of the need for techniques of run-time parallelization is given. A survey of the various approaches to parallelizing partially parallel loops and fully parallel loops is presented. Special emphasis is placed on two parallelism enabling transformations, privatization and reduction parallelization, because of their proven efficiency. The technique of speculatively parallelizing `doall` loops is presented in more detail.

This survey limits itself to the domain of Fortran applications parallelized mostly in the shared memory paradigm. Related work from the field of parallel debugging and parallel simulation is also described.

1 Automatic Parallelization

In recent years parallel computers have finally begun to move from university and national laboratories into the mainstream of commercial computing. This move has been caused by advances in technology (miniaturization), lower costs and an ever increasing need for more computing power. Unfortunately, although the hardware for these systems has been built and is commercially available, exploiting their potential in solving large problems fast, i.e., obtaining scalable speedups, has remained an elusive goal because the existing software to run these machines did not keep up with the technological progress.

We believe that a necessary condition for parallel processing to truly become mainstream is to deliver sustainable performance (speedups across a wide variety of applications) while requiring only the same or similar efforts on the part of the user as is the case for sequential computing. We recognize three complementary avenues towards obtaining a scalable speedups on parallel machines:

- Good parallel algorithms – to create intrinsic parallelism in a program.
- Development of a standard parallel language for portable parallel programming.
- Restructuring compilers to optimize parallel code and parallelize sequential programs.

We believe parallel algorithms are absolutely essential for writing a program that is to be executed on a parallel system. It is not possible to obtain a scalable concurrent execution from an application that employs inherently sequential methods. Once this is agreed upon, there are two different ways to express this parallelism: explicitly or implicitly. Explicitly written parallel programs can potentially produce good performance if the programmer is very good and the problem size (and difficulty) is manageable. Achieving this requires a parallel language that is both expressive as well as standardized. If we cannot clearly express the algorithmic parallelism then the effort is wasted, and if the language is not a recognized standard, then portability difficulties will make it non-economical. Additionally, we know from experience that even for very well trained people coding only with explicit parallelism may be significantly more difficult and time-consuming than programming in a sequential language. In particular solving concurrency and data distribution issues is a very difficult and error-prone task, which is contrary to our principle that parallel and serial program development should require similar effort. Expressing explicitly *all* levels (from instruction to task level) of parallelism within the same language and performing explicit optimizations that are architecture specific are additional difficulties in coding parallel programs.

Just as important is the fact that parallel systems don't run only newly written applications. There is an enormous body of existing software that must be ported and perform well on these new systems. One solution is to rewrite the so named 'legacy' programs [9], but this could prove to be prohibitively expensive. The alternative is to automatically transform them for concurrent execution by means of a restructuring or parallelizing compiler. This compiler should be able to safely (without errors) detect the available parallelism and transform the code into an explicitly parallel (machine) language. Most likely it would not be able to modify the algorithms used by the original programmer and therefore the performance will be limited (upper bounded by the intrinsic, and limited, parallelism of the original application).

We therefore think that the ideal parallelizing compiler should be capable of transforming both 'legacy' code and modern code into a parallel form. The new codes that use parallel algorithms can be written in an established, standard language like Fortran 77, in an explicitly parallel language or, even simpler, in a sequential language with parallel assertions. Coding in a hybrid language (serial language with non-mandatory directives) should require the least amount of effort: the programmer expresses as much knowledge about the code as he/she has and leaves the rest for automatic processing. Such a compiler must address two major issues: (a) *parallelism detection* and (b) *parallelism exploitation*.

Parallelism detection means finding the portions of code that can be safely executed concurrently and the relative order (the synchronizations) that must be maintained between their components. For this purpose the compiler needs to analyze the code and transform it into explicit parallel constructs.

Parallelism exploitation or optimization requires using the data and control dependence information previously detected for scheduling the work on a target architecture. In this process issues like data distribution for communication minimization and scheduling for load balancing have to be addressed.

1.1 The Need for Run-Time Parallelization

The first task of a restructuring compiler is to perform a data dependence analysis and detect the segments of code that can be executed concurrently. Essentially this entails performing an analysis of the memory access pattern, mainly through inspection of the expressions through which array elements are indexed. From the point of view of access pattern analysis we distinguish two major classes of applications:

1. *Regular programs* whose memory accesses can be described by a 'well-behaved' analytic function which is statically defined. Current state-of-the-art compilers do a reasonable job in analyzing and extracting the parallelism from these programs.

2. *Irregular programs* whose memory accesses are described through an ad-hoc map, usually represented by a subscript array in Fortran or pointers in C. These indirection maps cannot be analyzed at compile time if they are filled in only during program execution. Depending on how the subscript arrays are defined at run-time we distinguish two subclasses of irregular applications:

   (a) Programs with *static memory patterns* which are defined at the beginning of the execution by reading in an input file (the data set). After the initial setup, the access pattern does not change (but it is not available statically).

   (b) Programs with *dynamic memory patterns* which are computation dependent and are modified from one execution-phase to another, e.g., because of the changing interactions of the underlying physical phenomena they are simulating.

Techniques addressing the issue of data dependence analysis have been studied extensively over the last two decades [34, 47] but parallelizing compilers cannot perform a meaningful data dependence analysis and extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. Unfortunately irregular programs, as previously defined, represent a large part of all scientific applications. Several examples of codes that fall into this category are summarized in Table 1. The first two, SPICE and DYNA3D are input dependent while the others are dynamic programs. Since modeling techniques are becoming more sophisticated we believe that future large scale simulations and computation will be dynamic in nature and will only increase the fraction of statically non-analyzable codes. It is widely assumed that more than 50% of codes [20] are of the irregular type.

Thus, in order to realize the full potential of parallel computing it has become clear that static (compile–time) analysis must be augmented with new methods [8, 12, 15]. We need techniques that let us access the information necessary to decide if a loop is parallel and perform parallelizing transformations. The only time this data is available is during program execution, at *run-time*. Run–time techniques can succeed where static compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run–time dependent conditions, and subscript expressions can all be analyzed unambiguously during execution.

It should be noted that compilers can fail also due to limitations in their current analysis algorithms. For example, most dependence analysis algorithms can only deal with subscript expressions that are linear in the loop indices. In the presence of complex non–linear expressions, a dependence is usually assumed. Global

Program	Application Domain
SPICE	circuit simulation
DYNA–3D PRONTO–3D	structural mechanics
GAUSSIAN DMOL	quantum mechanical simulation of molecules
CHARMM DISCOVER	molecular dynamics simulations of organic systems
FIDAP	modeling complex fluid flows

Table 1: Irregular applications

parallelization, i.e., deeply nested loops containing subroutine calls, is quite often not possible because inter-procedural analysis generates extremely complex expressions that are in the end intractable although statically defined.

It is important to specify at this point that the scope of this paper encompasses techniques that have been developed for the exploitation of loop parallelism in Fortran programs, mostly in a shared memory environment. We believe that the SPMD model is easier to implement and that the global virtual address space will become the dominant programming paradigm. Although Fortran has been for many years the prevalent language for scientific programming leaving an enormous legacy we do not believe that it is going to retain its dominance; however techniques developed for modern, irregular Fortran codes may be easily applied in C or C++. We therefore think that the restrictions imposed in this paper will not result in any loss of generality.

After briefly explaining some fundamental aspects of data dependences that are directly applicable to run-time parallelization, in Section 3 we present an overall picture of the different approaches that researchers have taken to run-time parallelization. In the next sections we first survey the most important techniques used for partially parallel loops. Then we introduce the newer technique of speculative parallelization of fully parallel loops and present some experimental results. In Section 7 we mention some related work in parallel debugging and simulation. We conclude in Section 8 with our view of the future challenges that must be addressed to transform run-time parallelization from an experimental to a standard technique.

2 Foundations

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. We will refer to such loops as `doall` loops. If, however, not *all* iterations can be executed independently then a number of synchronizations are necessary to insure the correct execution order of iterations and a sequentially consistent outcome of the computation. We will call such loops, *partially parallel* loops, i.e., loops that need synchronizations. At one extreme we find completely sequential loops which execute in a number of timesteps equal to their iteration space (critical path length equal to number of iterations) and at the other extreme we find loops that need only one synchronization (critical path length is 1).

So far, most static analysis techniques have focused on finding fully parallel loops because they scale well with data set size and number of processors. Sometimes, when compilers detect a very regular dependence pattern (with a constant dependence distance) the loop can be transformed into a `doacross` (pipelined)

type of parallel execution that usually only pays off only in small systems where synchronization costs are low. In principle the applicability of the `doacross` can be extended if the work to be synchronized has a large granularity and therefore does not occur very often.

In the following subsections we review some basic principles for detecting `doall` fully parallel loops (Section 2.1) and some techniques for eliminating loop carried data dependences which can enable full or partial parallelism (Section 2.2). Section 2.3 presents techniques used in the parallelization of partially parallel loops on multiprocessors. The collection of basic techniques presented in the next section is only a subset of those available to modern restructuring compilers but includes the majority of those that are important when static compiler analysis fails, i.e., for irregular, dynamic applications.

2.1 Fully Parallel (`Doall`) Loops

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 [5, 24, 34, 47, 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 are data producer and consumer dependences, i.e., they express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory–related dependences, are caused by the reuse of memory, e.g., program variables.

```
do i=1, n
   A(K(i)) = A(K(i)) + A(K(i-1))
   if  (A(K(i)) .eq. .true.)  then
     .................
   endif
enddo
```

(a)

```
      do i = 1, n/2
S1:      tmp = A(2*i)
         A(2*i) = A(2*i-1)
S2:      A(2*i-1) = tmp
      enddo
```

(b)

```
do  i=1, n
   do  j = 1, m
S1:      A(j) = A(j) + exp()
   enddo
enddo
```

(c)

Figure 1: Examples of loops with different data dependences

If there are flow dependences between accesses in different iterations of a loop, then the semantics of the loop cannot be guaranteed if the loop is executed in fully parallel form. The iterations of such a loop are not independent because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration of the loop. For example, the iterations of the loop in Fig. 1(a), which computes the prefix sums for the array A, must be executed in order of iteration number because iteration $i + 1$ needs the value that is produced in iteration i, for $2 \leq i \leq n$.

Quite often, if there are no flow dependences between the iterations of a loop, then the loop may be executed in fully parallel form. The simplest situation occurs when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and the loop, as is, can be executed in parallel. For example, there are no cross-iteration dependences in the loop shown in Fig. 1(b), since iteration i only accesses the data in $A[i]$, for $1 \leq i \leq n$. If there are no flow dependences, but there are anti or output dependences, then the loop must be modified to remove all these dependences before it can be executed in parallel. Unfortunately, not all situations can be handled efficiently.

2.2 Parallelism Enabling Transformations

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*
- *reduction parallelization*

Privatization can remove certain types of anti and output dependences by creating, whenever allowed, for each processor cooperating on the execution of the loop, private copies of program variables that give rise to anti or output dependences (see, e.g., [34, 10, 27, 28, 45, 46]). The loop shown in Fig. 1(b), which, for even values of i, swaps $A[i]$ with $A[i-1]$, is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement `S1` of iteration i and statement `S2` of iteration $i+1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable `tmp`.

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.

Reduction variables are therefore accessed in a certain specific pattern (which leads to a characteristic data dependence graph). A simple but typical example of a reduction is statement `S1` in Fig. 1(c). The operator $\otimes$, is in this case the $+$ operator, the access pattern of array $A(:)$ is *read, modify, write*, and the function performed by the loop is to add a value computed in each iteration to the value stored in $A(:)$. This type of reduction is sometimes called an *update* and occurs quite frequently in programs.

There are two tasks required for reduction parallelization: *recognizing the reduction variable*, and *parallelizing the reduction operation*. (In contrast, privatization needs only to recognize privatizable variables by performing data dependence analysis, i.e., it is contingent only on the access pattern and not on the operations.)

Parallel reductions algorithms have been developed for quite some time. If, as is frequently the case, the reduction operation is *commutative*, then the implementation of such methods is less restrictive. One typical method for the case of *commutative* reductions is to transform the `do` loop into a `doall` and enclose the access to the reduction variable in an unordered critical section [15, 50] – a section of code guarded by a lock–unlock operation which allows mutually exclusive operations on the shared variable. Drawbacks of this method are that it is not always scalable and requires synchronizations which can be very expensive in large multiprocessor systems.

A scalable method can be obtained by noting that a reduction operation is an associative recurrence and can thus be parallelized using a recursive doubling algorithm [22, 23, 25]. In this case the reduction variable is privatized in the transformed `doall`, and the final result of the reduction operation is computed in an interprocessor reduction phase following the `doall`, i.e., a scalar is produced using the partial results computed in each processor as operands for a reduction operation (with the same operator) across the processors. We note here that if the reduction operation is commutative then it can be parallelized using dynamically scheduled `doall`s (not only statically scheduled in monotonic order) and the cross-processor merging phase can be done in any order. There is anecdotal evidence that most of the reductions encountered in benchmarks are commutative.

Thus, the difficulty encountered by compilers in parallelizing loops with reductions arises not from finding a parallel algorithm but from recognizing the reduction statements. So far this problem has been handled at compile–time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement.

2.3 Partially Parallel Loops

Loops that cannot be transformed into an equivalent `doall` loop may still have their execution time significantly reduced if their partial parallelism can be uncovered and exploited. Assuming that we represent the dependence relation among iterations with a DAG (directed acyclic graph) we distinguish the following cases:

1. The critical path length (cpl) of the data dependence DAG is equal to the total number of iterations. These loops can be considered sequential and are not well suited for multiprocessor parallelization.

2. The critical path length is less than the number iterations, i.e., partially parallel loops which require cpl synchronizations if they are executed on a multiprocessor.

The general technique for speeding up the execution of sequential and partially parallel loops is the use of cross-iteration synchronizations to enforce flow dependences (Assuming that anti- and output- dependences have been removed through some techniques such as privatization and reduction parallelization).

Sequential loops with constant dependence distance can be sped up by overlapping some segment of their iterations. This technique has been well studied and is currently successfully applied for fine grain optimizations, where synchronizations are inexpensive and the degree of parallelism supported by hardware is small. Superscalar processors can software pipeline their loops if their associated dependence structure is statically analyzable. In the case of irregular applications, where compiler analysis is insufficient, software pipelining is not possible, because the safe initiation interval unknown and often variable.

Statically analyzable `doacross` loops have been exploited in multiprocessors through the use of hardware or software synchronization primitives [31, 49]. The compiler can place `post` and `await` instructions around the accesses that cause flow dependences. The speedup obtained with this technique is upper bounded by the size of the loop iterations that may be overlapped and the overhead of synchronizations.

If the data access pattern of the loop is not statically available then synchronizations have to be employed conservatively for all potential dependences and the resulting performance may not show any improvement over serial execution.

Partially parallel loops can have their iteration space partitioned in a sequence of disjoint sets of mutually independent iterations. These sets, also known as *wavefronts*, can be executed as chain of `doalls` separated by global synchronizations. When a large variance of the size of the *wavefronts* causes a load imbalance, the loop can be executed using `post` and `await` synchronizations. Of course this technique requires *a priori* knowledge of the dependence DAG of the whole iteration space. A third technique, which requires even more detailed information about the data dependences in the loop, is using mutually independent `threads` of dependent iterations, which may have to be synchronized at different points.

The general parallelization methods for loops that cannot be transformed for `doall` execution require a precise analysis and knowledge of the traversed access pattern. In the case of irregular, dynamic applications this analysis cannot be performed at compile time because the information is simply not available – it exists only at execution time. This is the reason behind the development of many run-time techniques for optimization in general and parallelization in particular.

3 Approaches to Run-Time Parallelization

The last ten years have been increasingly productive in the development of run-time techniques for detecting parallelism. We believe that this is, paradoxically, due to the continuously improving power of parallelizing compilers which, after managing most of the 'well-behaved' codes, failed when confronted with the newer, sparse and dynamic simulation/computation programs.

All run-time parallelization techniques developed so far have the common goal of detecting and enforcing data dependences on shared data structures, mostly arrays. As previously mentioned, the inability of the compiler to detect whether there are conflicting accesses is sometimes due to insufficiently powerful algorithms but mostly because of insufficient static information. All run-time techniques monitor the access pattern 'on-the-fly', i.e., as it occurs, and check it against some previously collected access history. If an out-of-order access has occurred then various actions are undertaken to insure proper parallel execution. The implementation of these methods, regardless of their type and goal, have, with few exceptions ([37]), relied on exclusive access (mutually exclusive) to some shared structure.

Based on the type of loop (`doall` or partially parallel) and how the access history is collected we distinguish the following run-time techniques:

1. Techniques for parallelizing partially parallel loops
   - Inspector/Executor methods
   - Speculative methods
2. Techniques for parallelizing fully parallel (`doall`) loops.
   - Inspector/Executor methods
   - Speculative methods

In an inspector/executor method the access pattern is collected by first extracting an 'inspector' loop from the loop under test. This inspector loop traverses the access pattern without modifying the shared data (wihtout side effects) before the actual computation takes place. On the other hand, speculative methods perform the inspection of the shared memory accesses during the parallel execution of the loop under test. It should be noted here that there are no known speculative techniques for extratcing the parallelism from a partially parallel loop.

All run-time techniques must satisfy the requirement that their associated overhead can be amortized and speedups can be obtained. Most methods cited in the literature which rely on the inspector/executor strategy amortize their cost by reusing their collected parallelism information (execution schedule) over many instantiations of the tested loop. This method, called *schedule reuse* has been successfully applied by Joel Saltz em et al. for many applications where only a few data access patterns are generated during program execution.

In the following sections we present an overview of the most representative techniques that have appeared in the literature for partially parallel loops (Section 4) and fully parallel loops (Section 5). In Section 6 we describe an overall framework for applying run-time parallelization in real applications.

Because the author believes that speculative methods are more general and can be applied to any program, regardless of dynamic of its access patterns, this paper will describe them in more detail.

4 Run–Time Techniques for Partially Parallel Loops

Simple run-time techniques for using multi-version loops have been in production compilers for quite some time. Many of today's parallelizing compilers postpone part of their analysis for the run–time phase by generating two-version loops [11]. These consist of an `if` statement that selects either the original serial loop or its parallel version. The boolean expression in the `if` statement typically tests the value of a scalar variable.

As mentioned in Section 2.3, most previous approaches to run–time parallelization have concentrated on developing methods for constructing execution schedules for partially parallel loops, i.e., loops whose

parallelization requires synchronization to ensure that the iterations are executed in the correct order. Briefly, run–time methods for parallelizing loops rely heavily on global synchronizations (communication) [13, 21, 26, 31, 35, 41, 43, 49], are applicable only to restricted types of loops [26, 41, 43], have significant sequential components [35, 41, 43], and/or do not extract the maximum available parallelism (they make conservative assumptions) [13, 26, 35, 41, 43, 49]. The only method that manages to combine the most advantageous features is that of [37]. It does however rely on the availability of an inspector loop, which is not a generally applicable technique.

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

Method	obtains optimal schedule	contains sequential portions	requires global synchron	restricts type of loop	privatizes or finds reductions
Rauchwerger/Amato/Padua [37]	Yes	No	No	No	P,R
Zhu/Yew [49]	No[1]	No	Yes[2]	No	No
Midkiff/Padua [31]	Yes	No	Yes[2]	No	No
Krothapalli/Sadayappan [21]	No[3]	No	Yes[2]	No	P
Chen/Yew/Torrellas [13]	No[1,3]	No	Yes	No	No
Saltz/Mirchandaney [41]	No[3]	No	Yes	Yes[5]	No
Saltz *et al.* [43]	Yes	Yes[4]	Yes	Yes[5]	No
Leung/Zahorjan [26]	Yes	No	Yes	Yes[5]	No
Polychronopoulous [35]	No	No	No	No	No
Rauchwerger/Padua [38, 39]	No[6]	No	No	No	P,R

Table 2: A comparison of run–time parallelization techniques for `do` loops. 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.

4.1 Methods Utilizing Critical Sections

One of the first run–time methods for scheduling partially parallel loops was proposed by Zhu and Yew [49]. It computes the wavefronts one after another using the following simple strategy. During a phase, an iteration is added to the current wavefront if none of the data accessed in that iteration is accessed by any lower unassigned iteration; the lowest unassigned iteration to access any array element is found using atomic *compare-and-swap* synchronization primitives and a shadow version of the array. Midkiff and Padua [31] extended this method to allow concurrent reads from a memory location in multiple iterations. Due to the compare–and–swap synchronizations, this method runs the risk of a severe degradation in performance for access patterns containing *hot spots* (i.e., many accesses to the same memory location). However, when there are no hot spots and the critical path length is very small, this method should perform well. An advantage of this method is reduced memory requirements: it uses only a shadow version of the shared array under scrutiny whereas all other methods (except [35, 38, 39]) unroll the loop and store all the accesses to the shared array.

Krothapalli and Sadayappan [21] proposed a run–time scheme for removing anti and output dependences

from loops. Their scheme includes a parallel inspector that determines the number of accesses to each memory location using critical sections as in the method of Zhu and Yew (and is thus sensitive to hotspots). Using this information, for each memory location, they place all accesses to it in a dynamically allocated array and then sort them according to iteration number. Next, the inspector builds a dependence graph for each memory location, dynamically allocates any additional global storage needed to remove all anti and output dependences (using renaming), and explicitly constructs the mapping between all the memory accesses in the loop and the storage, both old and new, thereby inserting an additional level of indirection into all memory accesses. The loop is executed in parallel using synchronization (full/empty bits) to enforce flow dependences. To our knowledge, this is the only run–time privatization technique except [38, 39].

Chen, Yew, and Torrellas [13] proposed an inspector that has a private phase and a merging phase. In the private phase, the loop is chunked and each processor builds a list of all the accesses to each memory location for its assigned iterations; read accesses to a memory location are serialized. Next, the lists for each memory location are linked across processors using a global Zhu/Yew algorithm [49]. Their scheduler/executor uses `doacross` parallelization [41], i.e., iterations are started in a wrapped manner and processors busy wait until their operands are ready. Although this scheme potentially has less communication overhead than [49], it is still sensitive to hot spots and there are cases (e.g., `doalls`) in which it proves inferior to [49]. For example, consider a loop with $cpl = p$ and dependence distance p as well, i.e., in which each processor's iterations access the same set of n/p distinct memory locations. In this case the new inspector requires time $O(np)$, and the original Zhu/Yew scheme uses time $O(p^2 + n)$. Although this example may appear a bit contrived, it is actually a quite realistic possibility. Consider for example a nested loop in which the inner loop is fully parallel and moreover, it always accesses the same memory locations. Thus, if, as would be natural, each processor were assigned one iteration of the outer loop, we would have precisely the situation described above.

4.2 Methods for Loops Without Output Dependences

The problem of analyzing and scheduling loops at run–time has been studied extensively by Saltz *et al.* [7, 41, 42, 43, 48]. 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 [43], the inspector constructs stages that respect the flow dependences by performing a sequential topological sort of the accesses in the loop. The executer 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 [41], 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).

Leung and Zahorjan [26] 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.*

4.3 A Scalable Method for Loop Parallelization

All previously described methods either perform accesses in critical sections or have a purely sequential phase (e.g., a topological sort), which degrade the desired scalability characteristics of any run-time parallelization technique. In [36, 37] a technique is presented in which all its 3 phases, inspector, scheduler and executor, are parallel and scale well with the number of processors.

The inspector phase records the memory references to each array element under test in a per processor private data structure and sorts them according to iteration number and type of access (read or write). The parallel sorting operation is implemented as a bucket sort because the range of the accesses is known (they are indexes in an array). The obtained per processor information is then merged to form a global structure containing the dependence graph (DAG) for each memory element referenced in the loop.

In the next phase a quick parallel inspection of the individual DAG's finds all read-only, independent, privatizable and reduction operands. Since these locations do not introduce any loop-carried data dependences they are removed form the global data structure.

Finally, the scheduler cross-references data address information with iteration number in topological order and generates sets of independent iterations (wavefronts). The executor then schedules these independent iterations on the machine's processors. The scheduler and executor can work in tandem or in an interleaved mode increasing the efficiency of the overall process. Alternatively, the method provides sufficient information to generate independent *threads* of execution (chains of dependent iterations) that may need only occasional synchronizations.

This method incorporates two powerful transformations, privation and reduction parallelization, performs all phases in a scalable manner and can generate either wavefronts or threads. The iteration schedule can be either pre-computed or overlapped (pipelined) with the actual loop execution. We should remark however that this technique can be applied only to loops from which a side-effect free inspector can be extracted.

4.4 Other methods

In contrast to the above methods which place iterations in the lowest possible wavefront, Polychronopolous [35] gives a method where wavefronts are maximal sets of contiguous iterations with no cross-iteration dependences. Dependences are detected using shadow versions of the variables, either sequentially, or in parallel with the aid of critical sections as in [49].

Another significant contribution to this field is the work of Nicolau [32]. Run-time disambiguation has been recently used in optimizing codes for instruction level parallelism [18, 17]. Their idea is to speculatively execute code very aggressively (out of order) despite the fact that some memory locations (few) could cause unsatisfied data dependences. The offending addresses which are used out of order are stored until all potential hazards have been cleared. If an error is detected repair code will backtrack and restart from that point.

5 Run-Time Techniques for Fully Parallel (`Doall`) Loops

Most approaches to run–time parallelization have concentrated on developing methods for constructing execution schedules for partially parallel loops, i.e., loops whose parallelization requires synchronization to ensure that the iterations are executed in the correct order. Note that the parallelism available in partially parallel loops does not necessarily scale with increased data set size. Indeed, there is anecdotal evidence that the critical path length actually increases with data size which means that scalable speedups may not be achievable, that is, using more processors may not always yield higher speedups.

The only class of loops that can almost always offer scalable speedups are the fully parallel loops. The more work (iterations) a `doall` loop executes, the more processors can be employed and a higher speedup can be obtained. Because there is experimental evidence that many important loops are indeed fully parallel [8], we believe that they represent the biggest potential payoff in program parallelization. As mentioned before, exploiting the parallelism in irregular applications requires the application of run-time techniques whose overhead needs to be minimized. Building an execution schedule for a partially parallel loop requires fundamentally much more run-time collected information and analysis than does detecting full parallelism. From this fact alone we can conclude that the run-time overhead of `doall` detection will be lower than that of partially parallel loops.

All the techniques surveyed in the previous section use some form of the inspector/executor model. Of these, only the method proposed by [37] does not make use of critical sections. Unfortunately the distribution of a loop into an inspector and an executor is often not advantageous (or even possible). If the address computation of the array under test depends on the actual data computation (they form a dependence cycle), as exemplified in Fig. 1(a), then the inspector becomes both computationally expensive and has side–effects. This means that shared arrays would be modified during the execution of the inspector loop and saving the state of these variables would be required – making the inspector equivalent to the loop itself. In addition, the desirable goal of exploiting coarse–grain parallelization, i.e., at the level of large complex loops, makes it even less likely that an appropriate "inspector" loop can be extracted. Furthermore, if the goal is to construct an execution schedule for a partially parallel loop containing address-data cycles (as opposed to detecting full parallelism) then the inspection/collection of its access pattern cannot possibly be executed in parallel because it may contain loop-carried dependences. Thus, the inspector/executor approach is not a generally applicable method, i.e., it is limited to special cases.

We now describe the only technique known to us that is designed to detect only `doall` loops and can be used both in the inspector/executor framework as well as in a speculative mode and can be therefore applied to any loop type. Because of the author's opinion that speculative `doall` parallelization can be the basis of run-time parallelization we present it in more detail the previously described methods.

First we describe the LRPD Test [40, 38], an algorithm that can detect fully parallel loops and that can validate two of the most effective parallelism enabling transformations: privatization and reduction parallelization. Then we show how the test can be used both speculatively as well as an inspector for parallelizing loops in real applications. The section concludes with a few experimental results.

5.1 The LRPD Test

In this section we describe an efficient run–time technique that can be used to detect the presence of cross–iteration dependences in a loop access pattern. If there are any such dependences, then this test will not identify them, it will only flag their existence. In addition, the test can check if a shared variable is safely privatizable or if it is participating in a reduction operation. We note that the test need only be applied to those scalars and arrays that cannot be analyzed at compile–time.

An important source of ambiguity that cannot be analyzed statically and can potentially generate 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 LRPD test eliminates such false dependences by checking only the dynamic data dependences caused by the actual cross–iteration flow of values stored in the shared arrays, thus potentially increasing the number of loops found parallel. This is accomplished using a technique called *dynamic dead reference elimination*

which is explained in detail following the description of the test.

In the following description, the test is applied to the access pattern of a shared array A whose dependence behavior could not be computed at compile time.

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

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

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

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

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

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

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

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

   (b) If $any(A_w[:] \wedge A_r[:])$,[1] i.e., if the marked areas are common *anywhere*, then the loop *is not* a `doall` and the phase ends. (Since we define (write) and use (read, but do not define) values stored at the same location in different iterations, there is at least one flow or anti dependence.)

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

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

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

Dynamic dead reference elimination. We now describe how the marking of the read and private shadow arrays, A_r and A_{np}, can be postponed until the value of the shared variable is actually used (Step 1(b)). More formally, the references we want to identify are defined as follows. As mentioned above, if a value stored in a shared–memory location is read into a private variable and only conditionally used then it may not introduce a data dependence if the use does not materialize at run–time. In order to recognize statically which read references may not affect the data flow at run–time we first define a dynamic dead read reference:

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

```
do i=1, 5
   z = A(K(i))
   if (B1(i) .eq. .true.) then
      A(L(i)) = z +C(i)
   endif
enddo
```

B1(1:5) = (1 0 1 0 1)
K(1:5) = (1 2 3 4 1)
L(1:5) = (2 2 4 4 2)

(a)

```
do i=1, 5
   markread(K(i))
   z = A(K(i))
   if (B1(i) .eq. .true.) then
      markwrite(L(i))
      A(L(i)) = z +C(i)
   endif
enddo
```

(b)

```
do i=1, 5
   z = A(K(i))
   if (B1(i) .eq. .true.) then
      markread(K(i))
      markwrite(L(i))
      A(L(i)) = z +C(i)
   endif
enddo
```

(c)

original PD test	shadow arrays 1	2	3	4	5	tw	tm
A_w	0	1	0	1	0	3	2
A_r	1	1	1	1	0		
A_{np}	0	0	0	0	0		
$A_w(:) \wedge A_r(:)$	0	1	0	1	0		
$A_w(:) \wedge A_{np}(:)$	0	1	0	1	0		

(d)

new LPD test	shadow arrays 1	2	3	4	5	tw	tm
A_w	0	1	0	1	0	3	2
A_r	1	0	1	0	0		
A_{np}	0	0	0	0	0		
$A_w(:) \wedge A_r(:)$	0	0	0	0	0		
$A_w(:) \wedge A_{np}(:)$	0	0	0	0	0		

(e)

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

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

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

As can be observed from the example in Fig. 2, this method allows the LPD test to qualify more loops for parallel execution then would be otherwise possible by just marking all memory references. It is important to note that this technique *may* be applicable in an inspector if it contains all the necessary predicates but is *always* possible in a speculative execution.

In particular, after marking and counting we obtain the results depicted in the tables. The loop fails the test without dynamic dead reference elimination since $A_w(:) \wedge A_r(:)$ is not zero everywhere (Step 2(b)). However, the loop passes the LPD test as $A_w(:) \wedge A_r(:)$ is zero everywhere, but only after privatization, since $tw(A) \neq tm(A)$ and $A_w(:) \wedge A_{np}(:)$ is zero everywhere.

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

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 describe how potential reductions can be validated at run-time. Techniques for statically finding more potential reductions are described in more detail in [40].

A 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 [40]. The reduction can be validated by checking at run-time that the reduction variable x satisfies the definition given in Section 2, i.e., that x is only accessed in the reduction statement, and that it does not appear in exp. In addition, if x is a reduction variable in several potential reduction statements in the loop, then it must also be verified that each of these reduction statements have the same operator. The need for such a test could arise if the reduction variable is an array element accessed through subscripted subscripts and the subscript expressions are not statically analyzable. For example, although statement S3 in the loop in Fig. 3(a) matches a reduction statement, it is still necessary to prove that the elements of array A referenced in S1 and S2 do not overlap with those accessed in statement S3, i.e., that: $K(i) \neq R(j)$ and $L(i) \neq R(j)$, for all $1 \leq i, j \leq n$. Thus, the LRPD test must check at run–time that there is no intersection between the references in S3 and those in S1 and/or S2; in addition it will be used to prove, as before, that any cross–iteration dependences in S1 and S2 are removed by privatization. To test this new condition we use another shadow array A_{nx} to flag the array elements that *are not* valid reduction variables. Initially, all array elements are assumed to be valid reduction variables, i.e., $A_{nx}[:] = false$. In the marking phase of the test, i.e., during the speculative parallel execution of the loop or in the inspector 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 or after the inspector loop is executed, the analysis phase of the test is performed. An element of A is a valid reduction variable if and only if it was not invalidated during the marking phase, i.e., it was not marked in A_{nx} as not a reduction variable for any iteration. The other shadow arrays A_{np}, A_w and A_r are initialized, marked, and interpreted just as before.

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

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

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

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

(a)

```
      doall  i = 1, n
         markwrite(K(i))
         markredux(K(i))
S1:      A(K(i)) = .....
         markread(L(i))
         markredux(L(i))
S2:      ..... = A(L(i))
         markwrite(R(i))
S3:      A(R(i)) = A(R(i)) + exp()
      enddoall
```

(b)

```
      do i = 1, n
S1:      A(S(i)) = A(S(i)) + exp(X(i))
S2:      A(R(i)) = A(R(i)) + exp()
      enddo
```

(c)

```
      A_nx (:) = 0
      doall  i = 1, n
         markwrite(R(i))
         if (A_nx(R(i)) .ne. 0)  then
            if (A_nx(R(i)) .ne. '*') markredux(R(i))
         else
            A_nx(R(i)) = '*'
         endif
         markread(X(i))
         markredux(X(i))
S1:      A(R(i)) = A(R(i)) + exp(A(X(i)))
         markwrite(S(i))
         if (A_nx(S(i)) .ne. 0)  then
            if (A_nx(S(i)) .ne. '+') markredux(S(i))
         else
            A_nx(S(i)) = '+'
         endif
S2:      A(S(i)) = A(S(i)) + exp()
      enddoall
```

(d)

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

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

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

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

5.3 Complexity and Implementation Notes

Private shadow structures. The LRPD 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 (conceptually) into the global shadow structures. In fact, using private shadow structures enables some additional optimization of the

LRPD 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 re–initialization of the shadow structures is required between successive iterations, and checks such as "has this element been written in this iteration?" simply require checking if the corresponding element in A_w is marked with the iteration number. Another benefit of the iteration number "marks" is that they can double as time–stamps, which are needed for performing the last–value assignment to any shared variables that are live after loop termination.

A processor–wise version of the LRPD test. The LRPD 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 LRPD test so that all iterations assigned to a processor are considered as one "super–iteration" by the test. Note that a loop that is not fully parallel could potentially pass the processor–wise version of the LRPD test because data dependences among iterations assigned to the same processor will not be detected. This is acceptable (even desirable) as long each processor executes its assigned iterations in increasing order using static scheduling. The storage requirements for the shadow structures are at most 4 bits per element.
Complexity. If the test uses private shadow structures then the time complexity of the LRPD test is $T(n, s, a, p) = O(na/p + \log p)$, where p is the number of processors, n is the total iteration count of the loop, s is the number of elements in the shared array, and a is the (maximum) number of accesses to the shared array in a single iteration of the loop.

The space requirements of the test are $O(ps)$ (more precisely, 4 fields per shadowed element). In the processor-wise version the test requires at most 4 bits per elements on each processor. 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). Using "shadow" hash tables (instead of arrays), each processor will only have private shadow copies of the array *elements* accessed in iterations assigned to it. The storage requirements become in this case $O(na/p)$ for each processor or $O(na)$ in total and the time complexity of the test becomes $O(na/p + \log p)$. A more precise analysis can be found in [40].

5.4 Inspector/Executor vs. Speculative Strategies

So far the only run-time test that can be performed in either inspector/executor or speculative mode is the LRPD test which checks exclusively for full parallelism. Each strategy has its own advantages and disadvantages and the choice between the two is dictated by the particular case at hand.

In an inspector/executor strategy a proper inspector loop must be extracted from the original loop, which is then instrumented for marking, i.e., for the collection of the data's access history. An inspector exists only if there are no cycles between data and address computation - which is not the case for many dynamic applications. After executing the inspector (in parallel, at run-time) an analysis of the shadow structures reveals whether the actual loop is fully parallel. A compiler–generated two version loop can then be executed accordingly.

If a proper inspector cannot be found then the only choice is speculative execution. This approach requires the compiler to generate a marking instrumentation of the original loop in order to collect the access history in the shadow structures. The analysis phase will be performed after the speculative execution of the parallelized loop under test. However, because the speculative parallel execution may fail and shared data may have been modified, the compiler must also generate code for saving state (before execution) and restoring state (after failure). Just as in the inspector/executor mode, a serial version is available if the loop is found not to be a `doall`.

Benchmark[2] Subroutine Loop	Experimental Results Technique	Speedup	Tested	Description of Loop (% of sequential execution time of program)	Inspector (computation)
MDG	14 processors				privatization
INTERF	speculative	11.55	doall	accesses to a privatizable vector guarded	data accesses
loop 1000	insp/exec	8.77	privat	by loop computed predicates (92% T_{seq})	branch predicate
BDNA	14 processors				privatization
ACTFOR	speculative	10.65	doall	accesses privatizable array indexed by a	data accesses
loop 240	insp/exec	7.72	privat	subscript array computed inside loop (32% T_{seq})	subscript array
TRACK				accesses array indexed by subscript array	
NLFILT	8 processors		doall	computed outside loop, access pattern	not applicable
loop 300	speculative	4.21		guarded by loop computed predicates (39% T_{seq})	
ADM				accesses privatizable array through aliases,	
RUN	14 processors		doall	array re–dimensioned, access	not applicable
loop 20	speculative	9.01	privat	pattern control flow dependent (44% T_{seq})	

Table 3: Summary of Experimental Results.

The decision of whether to use an inspector/executor (if at all possible) or a speculative approach is dictated by the overall potential performance gains. In Section 6 we will sketch a general performance driven strategy for run-time parallelization.

5.5 Experimental Results

This section describes some experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [3]) and 14 processors (Alliant FX/2800 [4]) using a Fortran implementation of the LRPD test.

Four `do` loops from the PERFECT Benchmarks [6] that could not be parallelized by any available compiler were considered. The results are summarized in Table 3. For each loop, the type of test applied is noted: *doall* indicates cross–iteration dependences were checked (Lazy Doall (LD) test), *privat* indicates privatization was checked (LRPD test). For each method applied to a loop, the speedup obtained is reported. 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.

In addition to the summary of results given in Table 3, Figures 4 through 5 show the speedup 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. In cases where extraction of a reduced inspector loop was impractical because of complex control flow and/or inter–procedural problems, we only applied the speculative methods.

The graphs show that the speedups scale fairly well with the number of processors and are a very significant percentage of the ideal speedup. We note that in the examples shown the speculative strategy gives superior speedups versus the inspector/executor method.

It should be noted that the loop from TRACK is parallel for only 90% of its invocations. In the cases when the test failed, we restored state, and re–executed the loop sequentially. The speedup reported includes both the parallel and sequential instantiations (Fig. 5).

[2]All benchmarks are from the PERFECT Benchmark Suite

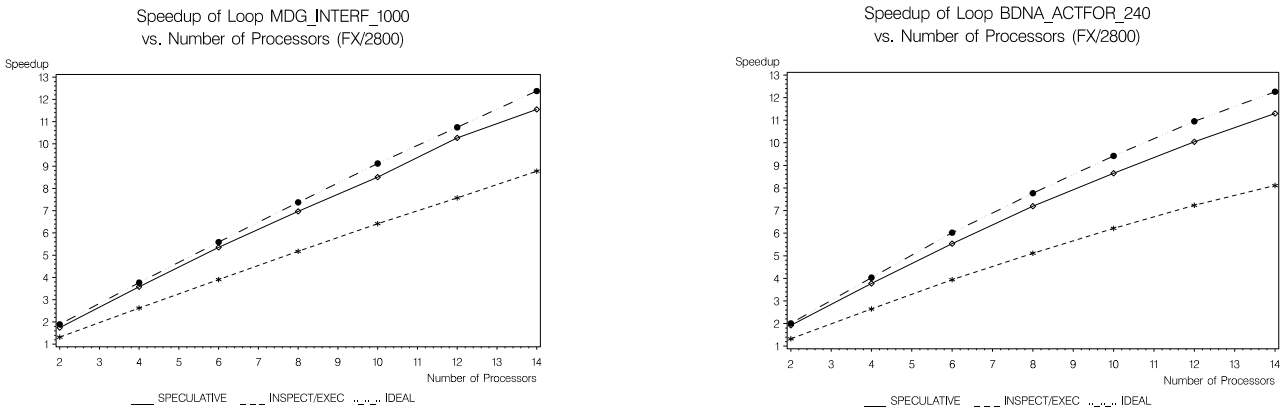

Figure 4: Exeprimental Speedup.

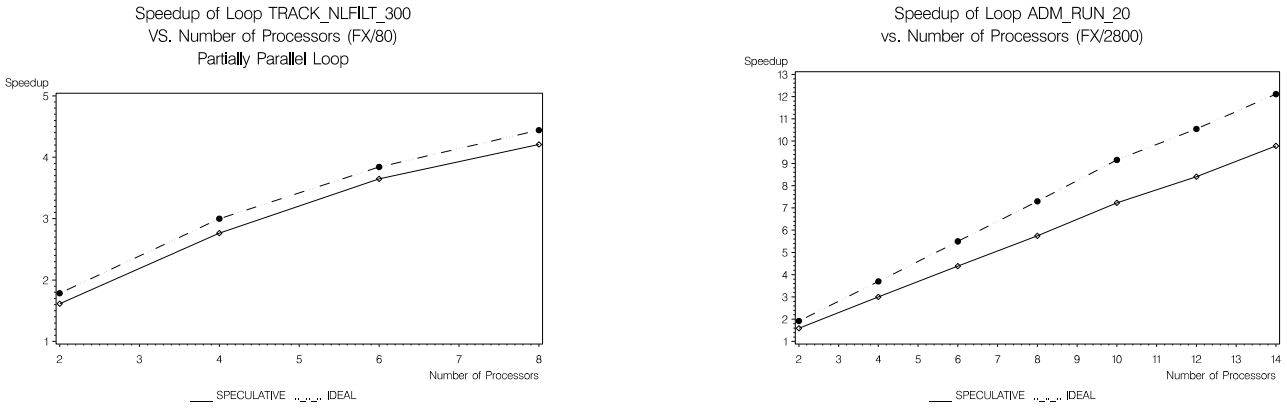

Figure 5: Exeprimental Speedup.

6 A Framework for Implementing Run-time Testing

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 has any intrinsic parallelism the compiler can choose to use a run-time parallelization technique in the hope that it will pay off. It should be noted that regardless of the method chosen the decision of applying a run-time test is in itself speculative: there is no assurance that a performance gain (parallelism) can be obtained and there is always an associated run-time overhead. The only way to minimize the potential loss is to: *(a)* improve the odds of predicting correctly and *(b)* reduce the run-time overhead.

In this section we give a brief outline of how a compiler might proceed when presented with candidate loop for run-time parallelization.

1. *At Compile Time.*

   (a) A cost/benefit analysis is performed using both static analysis and run–time collected statistics to determine whether the loop should be:

   (i) executed serially (because of the little probable parallelism)

   (ii) use a run-time technique for partially parallel loops (if statistics or some other analytical heuristics indicates moderate parallelism)

   (iii) test only for full parallelism

(b) Decide based on (approximate) statical analysis whether privatization, reduction or any other loop transformation may increase the probability for successful parallelization.

(c) Code is generated for:

- the collection of the access history as required for the chosen technique
- the potential sequential re–execution of the loop (in the case of speculative execution)
- any necessary checkpointing/restoration of program variables (in the case of speculative execution)

2. *At Run–Time.*

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

(b) Execute selected run-time technique in parallel (which may require a sequential re-execution of the loop)

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

As we have previously mentioned, run-time tests are always a 'gamble'. Collecting statistics will become part and parcel of this technique just as branch-prediction has become a classic technology. Complementary to the use of statistics is the reduction of the intrinsic run-time overhead of the employed methods. This can be achieved in two ways:

- Choice of the appropriate run-time technique based on code specifics and machine architecture, and
- Use of the partial information obtained by the compiler (by itself insufficient for static parallelization) to reduce the amount of information that must be collected at run time.

6.1 An Example

Because of its relative simplicity we have chosen to illustrate in Fig. 6 the code generated by a compiler for a simplified version of the speculative LRPD test. In this example, iteration numbers are used as "marks" in private shadow arrays. If the speculative execution of the loop passes the analysis phase, then the scalar reduction results are computed by performing a reduction across the processors using the processors' partial results. Otherwise, if the test fails, the loop is re–executed sequentially.

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. Padua *et al.* [2, 16] 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 [44, 14, 33]. In fact, a run–time anomaly detection method proposed by Snir, and optimized by Schonberg [44] and later by Nudler [33] bears similarities to a simplified version of the LRPD test presented in Section 5.1 (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

```
C        original loop
         dimension   A(1:m)
         do  i=1, n
S1:          A(R(i)) = A(R(i)) + exp()
S2:            .....    = A(L(i))
         enddo
```

(a)

```
C        Marking Phase
         dimension   A(m), pA(m,procs)
         dimension    A_w(m), pA_w(m,procs)
         dimension    A_r(m), pA_r(m,procs)
         dimension    A_nx(m), pA_nx(m,procs)
          Initialize(pA, pA_w, pA_r, pA_nx)
         doall  i=1,n
            private  p
            p = get_proc_id()
            pA_w(R(i), p) = i
S1:          pA(R(i), p) = pA(R(i), p) + exp()
            if  (pA_w(L(i), p) .ne. i)
               pA_r(L(i), p) = i
            pA_nx(L(i), p) = .true.
S2:           .....       = pA(L(i), p)
         enddoall
```

(b)

```
C        Analysis Phase
         doall  i=1,n
            A_w(1:m) = pA_w(1:m,i)
            A_r(1:m) = pA_r(1:m,i)
            A_nx(1:m) = pA_nx(1:m,i)
         enddoall
         result = test(A_w, A_r, A_nx)
         if  (result .eq. pass)  then
C         compute reduction
            doall  i=1, m
               if  (A_nx(i) .eq. .false.)
                  A(i) =  sum (pA(i, 1:procs))
            enddoall
         else
C         execute the loop sequentially
         endif
```

(c)

Figure 6: 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

detection mechanism maintains an access history of the shared locations and is checked 'on the fly' for an illegal 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 [29] improved this technique by dramatically reducing the memory requirements; the maximum accesses history storage is O(N), where N is the maximum level fork-join nesting. Since, in current systems at most 2 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 [30] 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 [19] and defined as " ... 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 Future Directions

In this paper we have considered the problem of parallelizing statically intractable loops at run–time and have summarized some of the major approaches to solve it. It is clear that the rate at which new run-time techniques have been proposed has been constantly increasing. As mentioned in Section 1.1, the motivation is, paradoxically, the increased static analysis power of parallelizing compilers. As modern, sparse codes are added to the various collected benchmark suites it has become more evident that static compilation has fundamental limits and that new techniques are needed if parallel processing is to succeed. Because the hardware for parallel systems has recently become commercially viable, the development of software, and in particular compilers to efficiently use them has become more urgent. We believe that these current circumstances will push for a more aggressive use of run-time optimizations. It is more economical to forgo some efficiency when parallelizing at run-time rather than executing sequentially, which would represent a total waste of resources.

While several promising techniques have been developed, much work remains to be done in order to make run-time parallelization of production quality.

New techniques need to be developed that can speculatively parallelize partially parallel loops. Current methods are either based on inspector/executor approaches that are not generally applicable or make use of synchronizations (spin–locks) that are not efficient in a large system. Run-time and static analysis have to be seemlessly integrated in order to pass all useable and possibly incomplete static information from the compilation to the execution phase. This will insure that only the minimum necessary information needs to be collected and analyzed at run-time, thus lowering the overhead. Additionally it is necessary to obtain more experimental, statistical data about the parallelism profiles of dynamic codes to increase the probability of correct speculations. We believe that, simultaneous with run-time parallelization, run-time data placement analysis and redistribution can be achieved with minimal additional overhead.

Last, but not least, architectural support that can either lower the run-time overhead of any of the known methods or replace them all together may soon become very feasible and attractive.

We believe that the significance of the methods presented here will only increase with the advent of both small scale parallel systems (workstations) as well as that of massively parallel processors (MPPs). Small scale, singly used (as opposed to multiused) parallel machines will not have appeal unless they can speed up single applications by executing them in parallel, regardless of efficiency. At the same time, the inability to partition and parallelize grand-challenge applications would make MPP's useless.

Finally we believe that the true importance of the described run-time parallelization techniques is that they break 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.

References

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

[2] T. Allen and D. A. Padua. Debugging fortran on a shared-memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, St. Charles, IL, 1987.

[3] Alliant Computer Systems Corporation. *FX/Series Architecture Manual*, 1986.

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

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

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

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

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

[9] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

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

[11] Mark Byler, James Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple Version Loops. *Proc. of 1987 Int'l. Conf. on Parallel Processing, St. Charles, IL*, 1987.

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

[13] D. K. Chen, P. C. Yew, and J. Torrellas. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of Supercomputing 1994*, pages 518–527, Nov. 1994.

[14] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proc. of 2-nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, 1990.

[15] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

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

[17] D.M. Gallagher, W. Y. Chen, S. A. Malke, J.G. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proc. 21st Ann. Int'l Symp. Computer Architecture*, pages 183–195, Chicago, IL, April 1994.

[18] A.S. Huang, G. Slavenburg, and J.P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 200–210, Chicago, IL, April 1994.

[19] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[20] Ken Kennedy. Compiler technology for machine-independent programming. *Int. J. Paral. Prog.*, 22(1):79–98, February 1994.

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

[22] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[23] C. Kruskal. Efficient parallel algorithms for graph problems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 869–876, August 1986.

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

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

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

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

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

[29] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing 1991*, pages 24–33, Albuquerque, NM, Nov. 1991.

[30] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proc. if the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, CA, May 1993.

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

[32] A. Nicolau. Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

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

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

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

[36] L. Rauchwerger, N. Amato, and D. Padua. Run–time methods for parallelizing partially parallel loops. In *Proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, July 1995.

[37] L. Rauchwerger, N. Amato, and D. Padua. A scalable method for run-time loop parallelization. *IJPP*, 26(6):537–576, July 1995.

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

[39] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. Technical Report 1390, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., November 1994.

[40] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation, La Jolla, CA*, pages 218–232, June 1995.

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

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

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

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

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

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

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

[48] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 International Conference on Parallel Processing*, pages 26–30. CRC Press, Inc., 1991. Vol. II - Software.

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

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