Abstract

The limited ability of compilers to find the parallelism in programs is a significant barrier to the use of high performance computers. It forces programmers to resort to parallelizing their programs by hand, adding another level of complexity to the programming task. We show evidence that compilers can be improved, through static and run-time techniques, to the extent that a significant group of scientific programs may be parallelized automatically. Symbolic dependence analysis and array privatization, plus run-time versions of those techniques are shown to be important to the success of this effort. If we can succeed to parallelize programs automatically, the acceptance and use of large-scale parallel processors will be enhanced greatly.

Keywords: compiler, parallelization, Fortran, dependence analysis, privatization, symbolic, run-time, Polaris

Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing *

William Blume Rudolf Eigenmann Jay Hoeflinger David Padua
Paul Petersen Lawrence Rauchwerger
Peng Tu
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

1 Introduction

For several decades, the most powerful computers have been those capable of exploiting parallelism at one or more levels of granularity ranging from instruction-level to task parallelism. This will probably continue to be the case, as it is unlikely that hardware technology alone will satisfy the ever increasing demand for computational power.

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. Usually that means they have to specify in their programs how parallel processors cooperate in the execution of the program and how data is mapped onto the memory system. Such hand-coding is error-prone and often not portable to different machines.

We believe that many users want to write sequential programs in conventional languages, leaving the machine-specific work to a compiler. Conventional languages, such as Fortran and C, provide a familiar programming environment and, as a consequence, would facilitate the acceptance of high-performance machines. Explicitly parallel programs are more difficult to develop, debug, and maintain than sequential programs. For example, task-parallel programs could exhibit intermittent errors due to misplaced synchronization operations. This kind of problem is very hard to find and fix. Also, to effectively exploit instruction-level parallelism, it is necessary to reorder elementary operations, some of which are hidden from the high-level language programmer. This reordering must be done by coding in assembly language, which is clearly undesirable. Furthermore, writing sequentially, without machine-specific constructs, makes it possible to port programs to a variety of high-performance computers. Portability is particularly important because high-performance machines are evolving rapidly. Software houses and end-users are understandably reluctant to develop parallel code that could be made obsolete by the rise and widespread acceptance of a radically new machine organization.

Unfortunately, today's compilers cannot do a totally accurate job of detecting parallelism in programs written in conventional languages. Sometimes the reason is that the information necessary for such a detection is not available at compile-time, but most often it is because the analysis algorithms used by the compiler are not sufficiently sophisticated.

Because of compiler limitations, the conventional programming languages are usually extended with directives to supply the compiler with the information that it is unable to obtain by itself. For example, most compilers accept directives indicating that the iterations of a `do` loop can be executed in parallel.

*Research supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

Some compilers also accept assertions about the values of variables or arrays which can be useful to identify parallelism or to determine how a section of the code should be mapped onto the target machine.

The preceding discussion applies to all high-performance computers including the new breed of massively parallel processors (MPPs). For these machines, Fortran extensions, HPF, and its precursor Fortran D, have been designed to provide a familiar environment to spare the programmer the need to work with low-level message-passing primitives. To circumvent the limitations of current compilers, these Fortran extensions include directives to specify how the data is to be distributed (the `align`, `distribute`, and `decomposition` directives) and whether a `do` loop can be executed in parallel (the `independent` directive). The success of MPP compilers rests heavily on the effectiveness of their techniques for automatic detection of parallelism. Techniques for dependence analysis, privatization, and symbolic analysis have all been mentioned as central to Fortran compilers for MPPs [HKT92]. For this reason, it is pertinent to discuss here the effectiveness of today's techniques for the automatic detection of parallelism and what we believe is necessary to improve their effectiveness.

2 Fundamental Techniques

We now present a brief description of automatic parallelization techniques. Due to lack of space we will omit many details. The interested reader is referred to [BENP93], a recent survey that includes an extensive list of references.

We begin with a discussion of data dependence analysis, whose purpose is to determine whether two statement instances[1] must execute in the order specified in the source program to guarantee correct results. Two statement instances may execute in any order or even in parallel when there is no chain of dependence relations connecting them.

In a sequential program, we say that a statement S_2 is *flow-dependent* on a statement S_1 if, in some execution of the program, an instance of S_2 could read from a memory location previously written to by an instance of S_1. This type of dependence arises when there is a producer-consumer relation between instances of S_1 and S_2. We say that statement S_2 is *anti-dependent* (resp. *output-dependent*) on a statement S_1 if, in some execution of the program, an instance of S_2 could write to a memory location previously read (resp. written) by an instance of S_1. Output- and anti-dependences are also known as *memory-related dependences*. They occur whenever a memory location is rewritten.

Dependences can be determined statically at compile-time or dynamically at run-time. We will next discuss static dependence analysis which, for all practical purposes, is the only method used today. We will briefly discuss dynamic analysis in a later section.

When only scalar variables are involved, static dependence analysis is simple. For example, in the statements

S_1 : A = B + C
S_2 : D = E + F
S_3 : G = A + C
S_4 : E = H + C

it is easy to determine that S_3 is flow-dependent on S_1 (because of A) and therefore, the two statements have to execute in the order they appear. It is also easy to determine that there is an anti-dependence from S_2 to S_4 (because of E). Clearly, there are no other dependence relations and therefore the sequence S_1;S_3 can execute in parallel with the sequence S_2;S_4 without affecting the outcome of the original sequential code.

In the presence of array references, computing the dependence relation accurately is considerably more difficult. Consider for example the loop:

[1] Because of iteration or recursion, program statements are often executed more than once. We call each execution a statement instance.

```
    do I = 1, N
S1 :    X(a * I + b) = ...
S2 :    ... = X(c * I + d)
    end do
```

To determine whether there is a dependence between statements S_1 and S_2, it is necessary to determine whether the equation $a * i_1 + b = c * i_2 + d$ has a solution in i_1 and i_2, both within the loop limits (i.e. within the interval $[1 : N]$). S_1 is flow-dependent on S_2 if there is a solution satisfying the constraint $i_1 \leq i_2$, and S_2 is anti-dependent on S_1 if there is a solution satisfying $i_2 < i_1$. In other words, determining the existence of a dependence in a singly-nested loop is equivalent to determining the existence of a solution to a system consisting of an equation and several inequalities. For multiply-nested loops and multi-dimensional arrays, the system would include several equations.

Cross-iteration dependences are the dependences between statement instances executing in different iterations of a `do` loop. In the previous example, there will be a cross-iteration dependence between S_1 and S_2 if there is a solution to the equation where $i_1 \neq i_2$. A loop can be executed in parallel without the need for any synchronization (except for the barriers at the beginning and end of the loop) if there are no cross-iteration dependences.

Because of its importance, the dependence analysis problem has been studied extensively and many techniques have been developed to determine automatically whether or not there are solutions to the associated systems of equations and inequalities. Practically all the techniques implemented in today's compilers, such as the GCD test and Banerjee's test, solve the problem numerically under the assumption that the subscript expressions are linear combinations of the loop indices. For these numerical techniques to work accurately, it is often necessary to know at compile time the values of the coefficients in the subscript expressions. The values of the loop limits is also necessary even though accurate results can sometimes be obtained by conservatively assuming that the upper limit is the largest possible integer value in the target machine [PP93]. If a compiler relies only on numerical techniques, as often is the case, it has to assume a dependence when the values of the coefficients or loop limits are not known. This is one of the main reasons why today's compilers fail to detect parallelism in sequential programs. This limitation can be overcome by using symbolic analysis. We will discuss symbolic analysis together with run-time techniques in a later section.

To reduce the number of dependences and increase parallelism, compilers try to apply several transformations. The two most important are *privatization* and *idiom replacement*. The objective of privatization is to determine which variables can be replicated across loop iterations to eliminate cross-iteration memory-related dependences. For example, variable `A` in the loop

```
    do I = 1, N
S1 :    A = X(I) + 2
S2 :    Y(I) = A + 1
    end do
```

is used to carry a value from S_1 to S_2. Cross-iteration dependences arise because there is only one copy of `A` for the loop. Replicating `A` to create a private copy per iteration would eliminate the cross-iteration dependences. Today's compilers do a good job of privatizing scalar variables, but as we discuss below, they are less successful in privatizing arrays. This is perhaps the main reason that they fail to parallelize many outer loops and thus are limited in their effectiveness. Privatization is not only important to detect parallelism, but also to increase the quality of the code generated by distributed-memory Fortran compilers [TP92] using the *owner computes rule*.

Another important transformation to eliminate dependences is the recognition and replacement of idioms, usually simple recurrences. One recurrence found frequently is *induction*. An induction statement in a loop uses the previous value of the *induction variable* to compute a new value, usually by adding or

multiplying a scalar expression. This dependence on the value from a previous iteration can prevent a loop from being parallelized.

If we can produce an expression for the induction variable which does not refer to its previous value, then the dependence is removed. The expressions which can be produced for induction variables in a loop nest become functions of the loop indices. This meshes well with dependence analysis when the induction variable is used to index arrays, since dependence analysis techniques require that the subscripts be expressed as functions of the loop indices. For example, in the loop

```
      J=0
      do I = 1, N
S1 :      J = J + 2
S2 :      Y(J) = X(J) + 1
      end do
```

statement S_1 can be eliminated, and all occurrences of J in S_2 can be replaced with the expression $2 * I$. In this way the cross-iteration dependences caused by S_1 disappear and the dependences caused by S_2 can be analyzed using conventional techniques. Today's compilers can do a good job at replacing simple induction variables, but often fail when the induction variables are updated at several points in a loop, especially in multiply-nested loops where inner loop bounds depend on outer loop indices.

Another recurrence that often arises is *reduction*. Reductions of the form $S = S + V(I)$ are very common. Such recurrences can also be detected and their dependences eliminated if the programmer is willing to accept a reordering of the computation. Today's compilers recognize many of these idioms.

Many other techniques to analyze and expose implicit parallelism have been studied. Some are applicable in real situations while others are only of theoretical interest because they only occur in the synthetic loops created for describing them.

We feel that it is important to advance the state of the art in parallelization by studying real programs, and finding a set of techniques that are necessary to parallelize them. In our study of real programs, we have found dependence analysis, privatization and idiom recognition to be the most important and widely applicable techniques.

3 The Need for Improvement in Current Techniques

One of our goals at Illinois is to improve the effectiveness of compilers in the detection of parallelism. To determine exactly what kinds of improvements are needed, it is necessary to know the strengths and weaknesses of current technology in automatic restructurers. Thus, in the late 80's, we performed a study on the effectiveness of parallelizing compilers using the Perfect Benchmarks, a suite of programs representing commonly used applications.

Our study showed that current automatic restructurers seldom achieve good speedups. Although restructurers can achieve significant gains for small kernels or benchmarks, the typical gain for real programs is small [EHJ+93]. Our experience has been only with coarse-grain loop parallelism, but a more accurate analysis of programs could be useful to detect vector and instruction-level parallelism.

In response to these poor results, we began to search for improvements that would increase the effectiveness of automatic restructurers on real programs. We did this by hand-transforming these programs into an efficient parallel form. With few exceptions, we have applied only transformations that can be potentially implemented in a parallelizing compiler. That is, we have restricted ourselves to transformations that do relatively small code changes as opposed to reorganizing large sections of the code. Furthermore, we have applied transformations that can be derived from the program text, rather than from knowledge of the application. The results of this effort are displayed in Figure 1. In all programs we inspected, we found that our transformations could improve the program performance

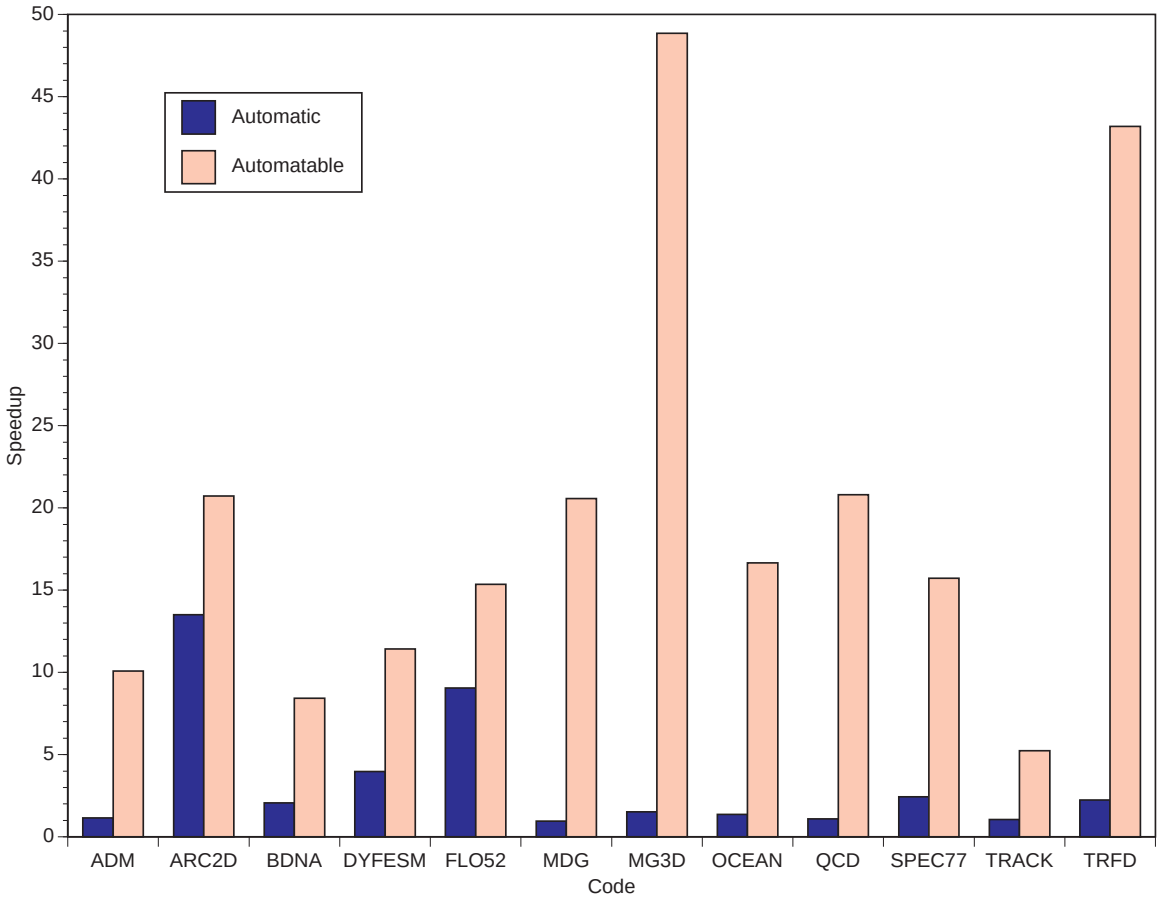

Figure 1: Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on the Cedar machine

by a significant factor. In fact, in most programs this was close to or matching the performance that resulted from the best reported manual effort [Poi90].

Having identified several techniques that can greatly improve the effectiveness of automatic parallelization, we are now implementing these techniques in the Polaris compiler. So far, a preliminary implementation of Polaris is able to parallelize half the programs shown in Figure 1 to the extent of the manually parallelized versions. Hence, we now have evidence that half of the suite of representative high-performance computer applications is amenable to automatic parallelization. This has been achieved by implementing new techniques for dependence analysis, privatization, and idiom recognition. In the next section we present an overview of our new techniques: dependence analysis and privatization, which we have already implemented, plus run-time techniques which we will implement in the near future.

As we have noted earlier, the choice of a benchmark set is critical. The results presented above are based on the code in the Perfect Benchmarks. Although these codes are perhaps the most widely accepted suite that represents a supercomputer workload, there is the open question of whether our findings carry over to other programs. To answer this question, we have collected an additional suite of programs that are currently being used by researchers at NCSA[2] and inspected them. So far, we have carefully examined two of these six codes, and found that our newly-implemented techniques are capable of parallelizing them.

Table 1 lists the most important loops of the suite of benchmark programs that we have chosen as a first yardstick for our Polaris compiler. Column 2 shows the percentage of program serial execution time of each loop. Columns 3 and 4 show which loops were parallelized in our hand-optimized program and in the automatically translated versions, respectively. Column 5 indicates the loops to which the new compiler technology can be credited for finding the parallelism.

The table shows that, with very few exceptions, Polaris is able to parallelize the loops which were parallelized in our previous hand-experiments. The non-parallelized cases include some inherently serial loops (indicated by "S" in hand-optimized column) and some rare patterns that are automatable but are not yet being implemented ("A" in the Polaris-optimized column). The mark ("I") indicates that the implementation is not yet complete, but the work will be done soon.

The loops marked in column 5 could not be parallelized by the compilers we used to compute the values in Figure 1. In all these programs the loops listed are the most time-consuming. Many of the other loops that we have not listed in the table can be parallelized successfully as well. For space reasons we have not shown all of them, although in some programs they contribute significantly to the execution time.

The serial loops in this table are of particular concern because they limit the achievable speedup on massively parallel processors. In ARC2D this problem is not as severe because there are inner parallel loops in the serial outer loops. However, BDNA is limited to a 30-fold speedup. To deal with BDNA, we need to develop transformations that work on I/O operations - an area we have not yet studied.

4 Improving Current Techniques: Run-Time and Symbolic Analysis

The manual parallelization effort of the Perfect Benchmarks has shown us that it is possible to automatically parallelize real programs effectively if parallelizing compilers are given a few enhanced techniques. These techniques, including dependence analysis, array privatization, and idiom recognition, need to be done symbolically to be effective. Doing these techniques symbolically means that the analysis manipulates or propagates symbolic expressions, equations, and inequalities that contain program variables.

Polaris implements two mechanisms for symbolic propagation. One is based on the techniques discussed in [CH78] for the forward substitution of symbolic equations and inequalities. The other is a demand-driven mechanism that backtracks symbolic variables and their relations as they are needed

[2] The National Center for Supercomputing Applications, Champaign, IL

PROGRAM-routine/loop	% program execution time	hand-optimized P=parallel S=serial	Polaris-optimized P=parallel S=serial	new technology is crucial
ARC2D-filerx/15	7.9	P	P (I)	X
ARC2D-stepfx/230	7.6	P	P	
ARC2D-filery/39	7.4	P	P (I)	X
ARC2D-stepfy/435	6.8	P	P	
ARC2D-tkinv/1	5.6	P	P	
ARC2D-stepfx/210	5.5	P	P	
ARC2D-stepfy/428	5.4	P	P	
ARC2D-tk/1	5.3	P	P	
ARC2D-ninver/1	4.7	P	P	
ARC2D-eigval/100	4.6	P	P	
ARC2D-xpenta/3	3.5	P	P	
ARC2D-ypenta/1	3.0	S (R)	S	
ARC2D-ioall/501	2.7	P	P	
ARC2D-xpent2/3	2.3	S (R)	S	
ARC2D-update/600	2.3	P	P	
ARC2D-rhsx/400	2.3	P	P	
ARC2D-rhsy/30	2.2	P	P	
BDNA-actfor/500	60.8	P	P	X
BDNA-actfor/240	33.6	P	P	X
BDNA-restar/15	2.1	S (IO)	S	
BDNA-actfor/320	1.8	P	P	
BDNA-actfor/700	0.3	P	P	X
FLO52-dflux/30	10.3	P	P	
FLO52-eflux/10	10.0	P	P	
FLO52-eflux/30	9.8	P	P	
FLO52-psmoo/40	8.7	P	P	
FLO52-psmoo/80	8.4	P	P	
FLO52-euler/50	8.0	P	P	
FLO52-dflux/60	6.0	P	P	
FLO52-eflux/40	5.0	P	P	
FLO52-step/20	4.8	P	P	
MDG-interf/1000	91.9	P	P	X
MDG-cshift/100	29.2	P	P	
MDG-poteng/2000	6.8	P	P	X
MDG-predic/1000	0.9	P	S (A)	
OCEAN-ftrvmt/109	42.7	P	P	X
OCEAN-in/10	14.5	P	P	
OCEAN-out/10	12.8	P	P	
OCEAN-ftrvmt/116	4.7	P	S (A)	
OCEAN-csr/20	4.6	P	P (I)	X
OCEAN-ocean/340	3.6	P	P (I)	X
OCEAN-acac/30	3.1	P	P (I)	X
OCEAN-ocean/420	2.9	P	P (I)	X
OCEAN-ocean/460	2.7	P	P (I)	X
OCEAN-ocean/440	2.7	P	P (I)	X
TRFD-olda/100	71.1	P	P (I)	X
TRFD-olda/300	27.8	P	P (I)	X
TRFD-intgrl/140	0.6	P	S (A)	X
CLOUD3D-kessler/1000	17.2	P	P (I)	X
CLOUD3D-s_advect/	7.7	P	P	X
CLOUD3D-s_advect/1	7.6	P	P (I)	X
CLOUD3D-s_advect/2	7.5	P	P (I)	X
CLOUD3D-kmsource/5	6.3	P	P (I)	X
CLOUD3D-s_mix/3	2.2	P	P	X
CLOUD3D-filter4/1	2.2	P	P	X
CLOUD3D-s_mix/1000	1.8	P	P	X
CLOUD3D-s_mix/2000	1.8	P	P	X
CLOUD3D-padvect/1	1.2	P	P	
CLOUD3D-wadvect/1001	1.2	P	P	
CLOUD3D-vadvect/1001	1.2	P	P	
CLOUD3D-cloud3d/14	1.0	P	P	
CMHOG-solvex2/200	34.8	P	P (I)	X
CMHOG-solvex1/300	16.4	P	P	X
CMHOG-solvex1/3000	11.2	P	P	X
CMHOG-pgas/10	8.7	P	P	
CMHOG-solvex2/1000	2.4	P	P	
CMHOG-solvex1/4000	2.0	P	P	X
CMHOG-nudt/100	1.8	P	P	
CMHOG-solvex1/110	1.6	P	P	
CMHOG-solvex1/100	0.9	P	P	
CMHOG-nudt/200	0.7	P	S (A)	
CMHOG-setup/70	0.7	P	P	
CMHOG-maxmin/10	0.3	P	S (A)	
CMHOG-hdfall/800	0.2	P	P	X

Notes:
(R)=true recurrence; (IO)=input/output operations;
(I)=not yet fully implemented; (A)=Automatable technique not being implemented

Table 1: Transformation of the time-critical loops of our evaluation suite

[TP93b]. We have used the former to support dependence analysis and the latter to support array privatization and idiom recognition. Even though, in theory at least, either approach could support all three analysis techniques, the demand-driven approach is potentially more efficient because it derives only the information that is needed.

In this section, we will discuss two ways that symbolic analysis can improve the effectiveness of automatic parallelization. First, we will discuss how symbolic data dependence tests can identify an important subclass of loops as parallel, which conventional data dependence tests cannot. Then, we will discuss how symbolic analysis can be used to improve array privatization, which is one of the most important techniques needed for effectively parallelizing programs. We will not discuss idiom recognition further in this paper. Some issues regarding this topic are discussed in [EHJ+93].

Even the most powerful symbolic analysis techniques cannot detect parallelism if the information is unavailable at compile time. In Section 4.3 we will describe techniques that perform run-time analysis in such situations.

4.1 Symbolic dependence analysis

4.1.1 Motivation for symbolic dependence analysis

Much research has been conducted in the area of data dependence analysis. Because of this, modern day data dependence tests have become very accurate and efficient [PP93]. However, these tests place constraints upon loop bounds and array subscript expressions of the loops that they examine. If these constraints are not met, these tests fail, thus preventing the loop from being fully parallelized.

Most data dependence tests require their loop bounds and array subscripts to be represented as a linear (affine) function of loop index variables. That is, the expressions must be in the form

$$\mathtt{c}_0 + \sum_{j=1}^{n} \mathtt{c}_j * \mathtt{I}_j$$

where $\mathtt{c}_j$ are integer constants and $\mathtt{I}_j$ are loop index variables. Expressions not of this form are called *nonlinear*. Most often, nonlinearity arises because the value of at least one coefficient is not known at compile-time. Because nonlinear expressions prevent the application of dependence tests, parallelizing compilers perform several analyses and optimizations to eliminate nonlinear expressions. Transformations such as constant propagation and induction variable substitution are used to remove loop variant variables. Other techniques have also been developed to handle additive loop invariant terms or to eliminate unwanted operations such as divisions [Pug92][HP91].

Unfortunately, not all nonlinear expressions can be removed. It was believed that this would not affect dependence testing in real programs since nonlinear expressions would be rare in real programs. However, our manual parallelization effort of the Perfect Benchmarks has shown us that this is not the case. In fact, four of the twelve codes (i.e. DYFESM, QCD, OCEAN, and TRFD) that we parallelized by hand would exhibit a speedup of at most two if we could not parallelize loops with nonlinear array subscripts [BE94b]. For some of these loops, nonlinear expressions occurred in the original program text. For other loops, nonlinear expressions were introduced by the compiler.

Two common compiler transformations can introduce nonlinearities into array subscript expressions: induction variable substitution and array linearization. As discussed in Section 2, induction variable substitution replaces variables that are incremented by a constant value for each loop iteration with a closed form expression composed of only loop invariants and loop indices. However, when induction variable substitution is performed upon multiply-nested loops, the resulting closed form expression may be nonlinear. For example, performing induction variable substitution on the loop nest in Figure 2 introduces a nonlinear expression into the subscript of array `A` if the value of `N` is not known at compile-time.

Array linearization transforms two or more dimensions of an array into a single dimension. Array linearization may be needed for interprocedural analysis when an array is dimensioned differently across

```
K = 0
do J = 1, M
  do I = 1, N
    K = K + 1
    A(K) = ...
  end do
end do
```

$\Longrightarrow$

```
do J = 1, M
  do I = 1, N
    A(I + N*(J-1)) = ...
  end do
end do
```

Figure 2: Before and after induction variable substitution

procedure boundaries. If the declared dimensions of a multidimensional array are symbolic expressions, the resulting linearized array may be nonlinear. For example, if the array `A`, which was originally dimensioned as `A(N,M)`, was linearized, its declaration will be changed to `A(N*M)`, and a reference `A(I,J)` will be changed to `A(I + N*J)`.

4.1.2 Symbolic dependence analysis in Polaris

To handle the nonlinear expressions that we have seen in the Perfect Benchmarks, we have implemented a symbolic dependence test in Polaris, called the *range test* [BE94a]. In the range test we mark a loop as parallel if (1) there are no cross-iteration dependences caused by scalars and (2) for all arrays `A` we can prove that the range of elements of `A` accessed by an iteration of that loop do not overlap with the range of elements accessed by other iterations. We prove this last condition by determining whether certain symbolic inequality relationships hold. Variable constraint propagation and symbolic simplification techniques are necessary to determine such constraints. For example, the range test can identify both loops in Figure 2 as parallel because the span of the range of elements of `A` generated by the `I` loop, which equals $\mathtt{N}-1$, fits within the stride of access to `A` due to the `J` loop, which equals `N`.

In general, the range test proves independence in a loop L by determining that for each array `A` the range of values that can be accessed within L fits within the absolute value of the stride of L. As illustrated next, to maximize the number of loops found parallel, we apply the range test upon permutations of the loop nest.

```
do I = 1, N
  do J = 0, (64 - I) / (2*N)
    do K = 1, 129
      L = 258*N*J + 129*I + K - 129
      A(L) = A(L) + A(L + 129*N)
      A(L + 129*N) = H * E
    end do
  end do
end do
```

Figure 3: Simplified version of loop nest **ftrvmt/109** from OCEAN

An example of an important loop nest that contained non-linear subscripts is shown in Figure 3. This is a simplified version of a loop which accounts for 43% of OCEAN's sequential execution time. Interprocedural constant propagation and loop normalization were needed to transform the loop nest into the form shown. Traditional data dependence tests would not be able to parallelize any of the loops in this loop nest because of the nonlinear term `258*N*J`. The range test can prove all three loops are parallel. This can be seen by examining Table 2, which displays the spans and strides of a permutation

Loop index	Span	Sum of inner spans	Stride
K	128	128	1
I	$129*N-129$	$(129*N-129)+128=129*N-1$	129
(OFFSET)	$129*N$	$129*N+(129*N-1)=258*N-1$	$129*N$
J	...	...	$258*N$

Table 2: Spans and strides of permuted loops in Figure 3

of the loop nest. The range test treats the two accesses `A(L)` and `A(L + 129*N)` as a single access of the form `A(L + OFFSET)`, where `OFFSET` is pseudo-loop of the form "`do OFFSET = 0, 129*N, 129*N`." In Table 2, it can be seen that, for the permutation shown, the sum of the inner spans of a loop always fits within the stride of the next loop in the permutation.

Although the range test was developed to complement rather than replace conventional dependence analysis, it was the only test needed to parallelize the loops listed in Table 1.

4.2 Array privatization

Although symbolic dependence analysis will allow us to prove that more references in a loop nest are independent from each other, it will not allow a significantly greater number of important loops to be parallelized without additional transformations. In our experience, the most important of these transformations is *array privatization* [TP93a].

As mentioned in Section 2, array privatization is used to eliminate memory-related dependences. That is, array privatization identifies scalars and arrays that are used as temporary work spaces by a loop iteration, and allocates a local copy of those scalars and arrays for that iteration so as to eliminate any cross-iteration anti-dependences or output-dependences caused by storage reuse.

To prove that a variable is privatizable, every use of that variable must be dominated by a definition of the variable in the same loop iteration. The definition of a variable *dominates* a use if and only if all control flow paths, from the start of the loop iteration to the statement containing the use, pass through the statement making the definition. If a definition dominates a use, then we may say that the definition *covers* the use.

Determining the dominating definition for a use of a scalar variable is straightforward, since the scalar is an atomic object which can only be read and written as a whole. However, since an array variable is a composite object that can be partially read and written, determining whether an array assignment covers an array use needs an elaborate analysis of the array ranges. More specifically, the array privatizer must prove that the region of array elements referenced by the use is a subset of the region of array elements defined by the assignment to determine that the use is dominated by the assignment. Symbolic analysis techniques are often required for these region comparisons, since the regions often contain symbolic expressions. As mentioned in Section 2, because of the difficulty of analyzing array references, most of today's parallelizing compilers only privatize scalars.

In many cases, determining the ranges in the definitions and use of arrays and whether one covers the other can be done using information immediately available at the points of definition and use. However, a more elaborate analysis requiring global information is necessary in many other cases.

An example where global information is necessary for array privatization is shown in Figure 4. To parallelize the `I` loop, the equivalenced arrays `A` and `AA` must be privatized. Loop `J` defines the region `AA(1:MP)`, while loop `K` uses region `A(1:M, 1:P)`. Thus, to prove that `A` (and therefore `AA`) are privatizable, we only need to prove that $MP \geq M*P$. To prove this, we must use information from outside the loop. As mentioned above, we use a demand-driven algorithm, based on a Static Single Assignment (SSA) representation, to obtain global information. To obtain the SSA form, program variables are renamed such that each time the variable is defined it is given a new name. Then, each time a variable is used, it is named according to which definition reaches it. In the program shown in Figure 4, each

```
      equivalence A(1,1), AA(1)
      ...
S1 :  M = ...
      ...
S2 :  MP = M * P
      ...
      do I = 1, N
          do J = 1, MP
              AA(J) = ...
          end do
          ...
          do K = 1, M
              do L = 1, P
                  ...= A(K,L) ...
              end do
          end do
      end do
```

Figure 4: Example for array privatization

variable is assigned only once, so no renaming is necessary to obtain the SSA form. Our demand-driven algorithm proceeds backwards from use to definition. To prove that $\texttt{MP} \geq \texttt{M} * \texttt{P}$, the algorithm starts at loop `J` and backward-substitutes `MP` with `M*P` as defined in statement $\texttt{S}_2$. Because the goal is satisfied, the algorithm stops at this point and no further replacements are performed.

Another example of the need for global information is shown in Figure 5, taken from BDNA. Several intermediate variables need to be privatized to parallelize the outermost loop in Figure 5. They are the scalar variables `R, P,` and `M`, and the arrays `IND,` and `A`. Except for array `A`, it is easy to determine that these intermediate variables are privatizable.

To determine whether `A` is privatizable in loop `I`, it is necessary to determine the range of the use of `A` in loop `L`. By analyzing the subscript and the range of the loop `L`, it is easy to determine that the range is $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$. The possible dominating definition for `A` is in loop `J`, where `A` is defined for the range `A(1:I-1)`. To prove that the definition in loop `J` dominates all the uses in loop `L`, we need to prove that $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$ falls in the range of `A(1:I-1)`.

A demand-driven strategy works well in situations like this where it is necessary to propagate values from complicated control structures with conditional assignments. The demand-driven analysis determines how many elements of `IND` are defined in loop `K` making use of the fact that the subscript `P` for the assignment to `IND(P)` is a monotonically increasing variable with an initial value of 1 and step of 1. Using a monotonic variable identification technique similar to induction variable identification, the algorithm determines that all the elements in $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(L)}\}$ are assigned in loop `K`.

Now that the algorithm knows the definition point for $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(P)}\}$, it can substitute the loop variant terms in $\{\texttt{A(IND(1))}, \texttt{A(IND(2))}, \ldots, \texttt{A(IND(P))}\}$ with their values. Each of them takes on a value of loop index `K`. Because the value of `K` falls in the range `[1:I-1]`, $\{\texttt{IND(1)}, \texttt{IND(2)}, \ldots, \texttt{IND(P)}\}$ will also fall in the same range. Hence all the uses of `A` fall within the range `[1:I-1]` and are therefore dominated by the definition `A(1:I-1)`. Thus, the algorithm determines that the array `A` is privatizable in loop `I`.

```
do I = 2,N
    do J = 1, I - 1
        IND(J) = 0
        A(J) = X(I,J) - Y(I,J)
        R = A(J) + W
        if (R .LT. RCUTS) IND(J) = 1
    end do
    P = 0
    do K = 1,I - 1
        if (IND(K) .NE. 0) then
            P = P + 1
            IND(P) = K
        end if
    end do
    do L = 1,P
        M = IND(L)
        X(I,L) = A(M) + Z
    end do
end do
```

Figure 5: Example from BDNA

4.3 Run-Time Techniques

Although compiler techniques such as those discussed above can often detect parallelism, it has become clear that, for a class of programs, compile-time analysis must be complemented with run-time techniques to obtain good speedups.

The reason for this is that the access pattern of some programs cannot be determined statically, either because of limitations in the current analysis algorithms or because the access pattern is a function of the input data. For example, compilers usually conservatively assume data dependences in the presence of subscripted subscripts. Although more powerful analysis techniques could remove this 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. Therefore, if data dependences such as these are to be detected, the analysis must occur at run-time. Because of the overhead involved, it is very important that run-time techniques be fast, in addition to being effective.

Another situation in which compilers have thus far been unable to generate parallel code is when the iteration space of a loop is not known at compile-time, as in `while` loops or `do` loops with conditional exits. Run-time techniques which are fast and effective are needed for these loops.

4.3.1 Detecting data dependences at run-time

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 executing the loop sequentially, the compiler could decide to speculatively execute the loop as a `doall`, and generate code to determine at run-time whether the loop was, in fact, fully parallel. If the subsequent test finds that the loop was not fully parallel, then it will be re-executed sequentially.

In order to implement such a strategy, we have developed a run-time technique, called the *Privatizing Doall test* (*PD test*), for detecting the presence of cross-iteration dependences in a loop [RP94]. If there are any such dependences, this test does not identify them; it only flags their existence. In addition,

```
do I = 1, n
   ...        = A(T(I))
   A(U(I))    = ...
   ...        = A(V(I))
end do
```

```
T(1:8) = [ 2 2 2 10 8 8 8 10]
U(1:8) = [ 1 3 5  4 7 3 6 12]
V(1:8) = [ 1 3 2 10 7 3 8 12]
```

	Position in shadow arrays													
	1	2	3	4	5	6	7	8	9	10	11	12	w_A	m_A
A_w	1	0	1	1	1	1	1	0	0	0	0	1	8	7
A_r	0	1	0	0	0	0	0	1	0	1	0	0		
A_{np}	0	0	0	0	0	0	0	0	0	0	0	0		
$A_w \wedge A_r$	0	0	0	0	0	0	0	0	0	0	0	0		

Figure 6: PD Test - PASSED

if any variables were privatized for speculative parallel execution, this test determines whether those variables were, in fact, validly privatized. Our interest in identifying fully parallel loops is motivated by the fact that they arise frequently in real programs.

The PD test

The PD test is applied to each shared variable referenced during the loop whose accesses cannot be analyzed at compile-time. For convenience, we discuss the test as applied to only one shared array, say A. Briefly, the test traverses and marks shadow array(s) during speculative parallel execution using the access pattern of A, and after loop termination, performs a final analysis to determine whether there were cross-iteration dependences between the statements referencing A. The first time an element of A is written during an iteration, the corresponding element in the write shadow array A_w is marked. If, during any iteration, an element in A is read, but never written, then the corresponding element in the read shadow array A_r is marked. Another shadow array A_{np} is used to flag the elements of A that *cannot* be privatized: an element in A_{np} is marked if the corresponding element in A is both read and written, and is read first, in any iteration.

A post-execution analysis determines whether there were any cross-iteration dependences between statements referencing A as follows. If $\mathtt{any}(A_w(:) \cap A_r(:))$[3] is true, (i.e., if the marked areas are common anywhere), then there is at least one flow- or anti-dependence that was not removed by privatizing A (some element is read and written in different iterations). If $\mathtt{any}(A_{np}(:))$ is true, then A is not privatizable (some element is read before being written in an iteration). Let w_A be the total number of writes that were marked in A_w by all iterations (computed during the parallel execution), and let m_A be the total number of marks in A_w (computed after the parallel execution). If $w_A \neq m_A$, then there is at least one output dependence (some element is overwritten); however, if A is privatizable (i.e., if $\mathtt{any}(A_{np}(:))$ is false), then these dependences were removed by privatizing A. The PD test is fully parallel and requires time $O(a/p + \log p)$, where p is the number of processors, and a is the total number of accesses made to A in the loop.

The PD test is illustrated using the loop shown in Figure 6. The access pattern is given by the subscript arrays T, V and U. Since $A_w(:) \wedge A_r(:)$ and $A_{np}(:)$ are zero everywhere, the loop was a doall, but only after privatizing A since $w_A \neq m_A$.

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

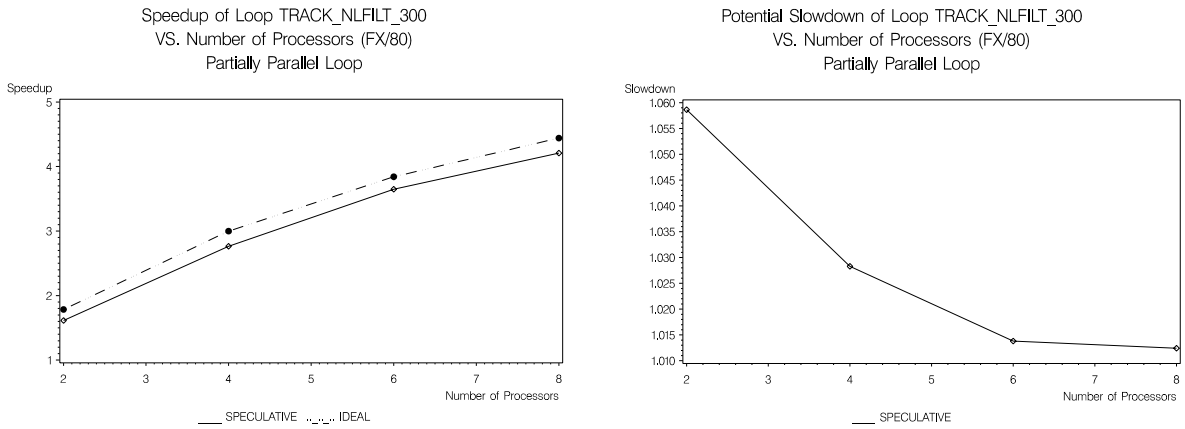

Figure 7:

4.3.2 `While` loops

We have also developed techniques for concurrently executing loops with unknown iteration spaces (i.e., `while` loops and `do` loops with conditional exits) [?]. For simplicity, we assume here that the `while` loop has *no cross-iteration dependences* except for those necessary to control the loop. If the dependence relations among the iterations of the loop are unknown, then, with some care, the PD test can be incorporated into the techniques discussed below.

Such a `while` loop can be considered as a sequence of independent iterations ordered by some underlying recursion. If the recursion has a closed form solution (e.g., a `do` loop with a conditional exit), then the loop can be executed in parallel. If each processor keeps track of the lowest iteration it executes that meets the termination condition, then the last valid iteration of the sequential version of the loop can be found by taking the minimum of the processor-wise minima. The values overwritten during executed iterations found to be invalid can be restored if we checkpoint prior to the parallel execution and maintain a record of *when* (iteration number) a memory location is written.

If the recursion does not have a closed form solution (e.g., a loop that traverses a linked list), then the iterations of the loop cannot be initiated simultaneously. In this case significant speedups may still be obtained by computing the values of the recursion serially, and performing the rest of the loop's work in parallel. One obvious method is to serialize the operations that update the values of the recursion (e.g., $next()$). A method that avoids explicit serialization is to compute all values of the recursion in each processor, but to assign each value to only one processor for processing, e.g., assign to processor i the iterations which are congruent to $i \bmod p$, where p is the total number of processors. Invalid iterations can be undone in the same way as when the recursion has a closed form solution.

4.3.3 Performance of run-time techniques

It can be shown that if the PD test passes (i.e., the loop is in fact fully parallel), then a significant portion of the ideal speedup of the loop is obtained. In particular, the speedups obtained range from nearly 100% of the ideal in the best case, to *at least* 25% of the ideal in the worst case. On the other hand, if the PD test fails (i.e., the loop is not fully parallel), then the slow-down incurred is proportional to $\frac{1}{p}T_{seq}$, where T_{seq} is the sequential execution time of the loop. If the target architecture is a MPP with *hundreds*, or in the future *thousands*, of processors, then the worst case potential speedups reach into the hundreds, and the cost of a failed test becomes a very small fraction of sequential execution time. Thus, speculating that the loop is fully parallel has the potential to offer large gains in performance, while at the same time risking only a small increase in the sequential execution time.

In Figure 7, we show experimental results of a Fortran implementation of the PD test on loop

nlfilt/300 from the Perfect Benchmark program TRACK. The measurements were made on the Alliant FX/80, a modestly parallel machine with 8 processors. The access pattern of the shared array in this loop cannot be analyzed by the compiler since the array is indexed by a subscript array that is computed at run-time. In addition, this loop 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.

Our experimental results indicate that our techniques for loops with unknown iteration spaces usually yield significant speedups when compared to the available parallelism in the original loop. The experiments have also shown that the overhead associated with these techniques is generally very small.

5 Conclusions

The last decade has seen a dramatic increase in the use of parallelism in all classes of machines. At the lower end of the spectrum, most new microprocessors exploit functional unit parallelism and, at the upper end, new MPP machines with thousands of processors have recently been developed. There is also an increasing presence of multiprocessor workstations and mainframes.

However, a major stumbling block for the widespread acceptance of parallelism is the difficulty of writing effective parallel programs. Better compilers with more powerful techniques for the detection and exploitation of parallelism are clearly needed. Even though today's compiler techniques are limited, it seems clear to us that compilers that accurately detect and effectively exploit parallelism can be developed. Such compilers will need to use a combination of static and dynamic techniques and include symbolic algebra and analysis methods. In the development of new techniques, researchers should devote a substantial effort to the analysis of real programs and program patterns to help them focus their attention where it is needed and to evaluate the effectiveness of the new methods. The problem of automatic detection of parallelism is certainly not trivial, but it is not insurmountable either, and the reward for success will more than compensate for the effort.

References

[BE94a] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., April 1994. CSRD Report No. 1345.

[BE94b] William Blume and Rudolf Eigenmann. Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1994. CSRD Report No. 1332.

[BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), February 1993.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[EHJ+93] R. Eigenmann, J. Hoeflinger, G. Jaxon, Zhiyuan Li, and D. Padua. Restructuring Fortran Programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–573, October 1993.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HP91] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. In A. Nicolau D. Gelernter, T. Gross and

D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 310–330. MIT Press, 1991.

[Poi90] Lynn Pointer. Perfect: Performance Evaluation for Cost-Effective Transformations Report 2. Technical report, University of Illinois at Urbana-Champaign, Center for Supercomputing Res & Dev, March 1990. CSRD Report No. 964.

[PP93] Paul M. Petersen and David A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. In *Proc. of ICS'93, Tokyo, Japan*, July 1993.

[Pug92] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[RP94] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . Technical report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev., January 1994. CSRD Report No. 1329.

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines, ACM SIGPLAN Notices 1993*, September 1992.

[TP93a] Peng Tu and David Padua. Automatic array privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 500–521, Portland, OR, August 1993. Springer Verlag.

[TP93b] Peng Tu and David Padua. Demand-driven symbolic analysis. CSRD Report 1336, University of Illinois at Urbana-Champaign, Center for Supercomp. R&D, Dec 1993.