Automatic Detection of Parallelism

A Grand Challenge for High-Performance Computing

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

The limited ability of compilers to find the parallelism in programs is a significant barrier to the use of high-performance computers. However, a combination of static and runtime techniques can improve compilers to the extent that a significant group of scientific programs can be parallelized automatically.

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.

But to achieve high performance on today's supercomputers, application 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.

In general, explicitly parallel programs are more difficult to develop, debug, and maintain than sequential programs. For example, task-parallel programs can 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.

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. Furthermore, writing sequentially,

1063-6552/93/$4.00 © 1994 IEEE

Basic techniques for automatically detecting parallelism

This description of automatic parallelization techniques necessarily omits many details; a recent survey by Utpal Banerjee et al. includes an extensive list of references on the subject.[1]

Dependence analysis

Because of iteration or recursion, program statements are often executed more than once; each execution is a *statement instance*. We use *data dependence analysis* to determine whether two statement instances 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 *S2* is *flow dependent* on a statement *S1* if, in some execution of the program, an instance of *S2* could read from a memory location previously written to by an instance of *S1*. This type of dependence arises when there is a producer-consumer relation between instances of *S1* and *S2*. We say that *S2* is *antidependent* on *S1* if, in some execution of the program, an instance of *S2* could write to a memory location previously read by an instance of *S1*; similarly, *S2* is *output dependent* on *S1* if an instance of *S2* could write to a memory location previously written by an instance of *S1*. Output and antidependences 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 runtime, but for all practical purposes, static dependence analysis is the only method used today.

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

$S1: A = B + C$
$S2: D = E + F$
$S3: G = A + C$
$S4: E = H + C$

it is easy to determine that *S3* is flow-dependent on *S1* (because of *A*), and therefore that the two statements have to execute in the order they appear. It is also easy to determine that there is an antidependence from *S2* to *S4* (because of *E*). There are no other dependence relations, so the sequence *S1*;*S3* can execute in parallel with the sequence *S2*;*S4* 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 this loop:

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

To determine whether there is a dependence between *S1* and *S2*, we must 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 (that is, within the interval [1:*N*]). *S1* is flow dependent on *S2* if there is a solution satisfying the constraint $i_1 \le i_2$, and *S2* is antidependent on *S1* 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 multidimensional arrays, the system would include several equations.

Cross-iteration dependences are those between statement instances executing in different iterations of a `do` loop. In the previous example, there will be a cross-iteration dependence between *S1* and *S2* if there is a solution to the equation where $i_1 \neq i_2$. A loop can be executed in parallel without 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 there are solutions to the associated systems of equations and inequalities. Practically all the techniques in today's compilers, such as the GCD test and Banerjee's test, solve the problem numerically under the assumption that

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, and both software houses and 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 do not detect parallelism in conventional programs very accurately, sometimes because the necessary information is not available at compile time, but most often because the compiler's analysis algorithms are not sophisticated enough. As a result, conventional programming languages are usually extended with directives that supply the information that the compiler cannot obtain by itself. For example, Fortran extensions — including High Performance Fortran and its precursor, Fortran D — have been designed for the new breed of massively parallel processors to provide a familiar environment that spares the programmer the need to work with low-level message-passing primitives. To circumvent the limitations of current compilers, these extensions include directives to specify how the data is to be distributed (`align`, `distribute`, and `decomposition`) and whether a `do` loop can be executed in parallel (`independent`). However, even when assertions are used, the compiler still needs to do a very accurate analysis of the source program to generate efficient code.

For these reasons, we believe that the success of compilers for MPP languages, such as High Performance Fortran, rests heavily on the effectiveness of their analysis

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 are 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.[2] 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 compilers fail to detect parallelism in sequential programs. This limitation can be overcome by using symbolic analysis (discussed later in the main article).

TRANSFORMATIONS

To reduce the number of dependences and increase parallelism, compilers try to apply several transformations. The two most important are *privatization* and *idiom replacement*. Privatization determines 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
```

carries a value from *S1* to *S2*. 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 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.

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 that 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 *S1* can be eliminated, and all occurrences of *J* in *S2* can be replaced with the expression *2*I*. In this way the cross-iteration dependences caused by *S1* disappear, and the dependences caused by *S2* 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 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.

References

1. U. Banerjee et al., "Automatic Program Parallelization," *Proc. IEEE*, Vol. 81, No. 2, Feb. 1993, pp. 211–243.
2. P.M. Petersen and D.A. Padua, "Static and Dynamic Evaluation of Data Dependence Analysis," *Proc. ICS '93*, ACM Press, New York, 1993, pp. 107–116.
3. P. Tu and D. Padua, "Array Privatization for Shared and Distributed Memory Machines," *Proc. 2nd Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Machines, ACM SIG Plan Notices 1993*, Sept. 1992.

techniques. Dependence analysis, privatization, and idiom recognition are the most important and widely applicable techniques for use on real programs (a sidebar describes these methods).

We have developed new techniques for dependence analysis, privatization, and idiom recognition that are more accurate than the techniques used on most restructuring compilers today. These techniques have successfully parallelized six programs (half) of a suite of representative high-performance applications, whereas commercial restructurers can parallelize only two of these codes. We have implemented these techniques in a source-to-source Fortran compiler called Polaris, and we will soon implement runtime techniques that further support the automatic detection of parallelism. The targets of Polaris are shared-memory multiprocessors and MPPs with a global address space. We chose these machines so that we could focus on the accuracy of the analysis, rather than the complexities of message generation. Improving the accuracy of analysis techniques is an important issue for parallelizing compilers. Our emphasis has been on the automatic detection of parallelism, although our techniques for data dependence and privatization could be used to improve message generation and data distribution.

The need for improvement

To make compilers more effective at detecting parallelism, we must know the strengths and weaknesses of current automatic restructurers. Thus, in the late '80s, we studied the effectiveness of parallelizing compilers

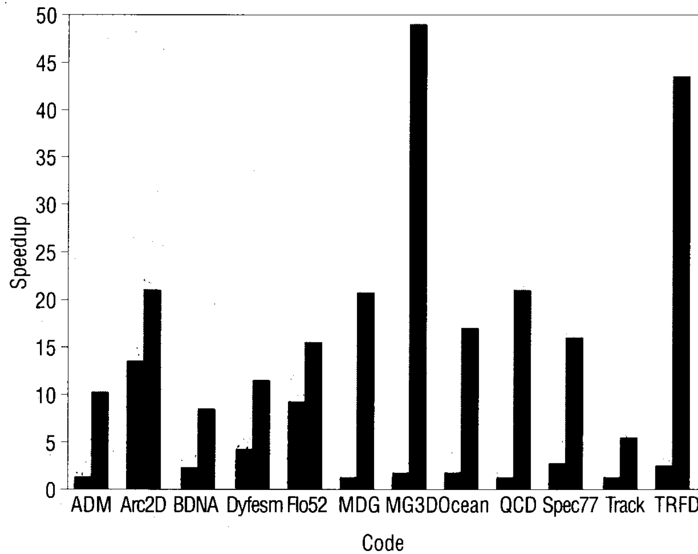


Figure 1. Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on the Cedar machine.

using the Perfect Benchmarks, a suite of programs representing common applications. Our study showed that current automatic restructurers can achieve significant speedups for small kernels or benchmarks, but that the typical gain for real programs is small.[1] (Our experience has been only with coarse-grained loop parallelism; 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 ways to increase the effectiveness of automatic restructurers on real programs.

We first hand-transformed these programs into an efficient parallel form. With a few exceptions, we applied only transformations that could be implemented in a parallelizing compiler. That is, we restricted ourselves to transformations that make relatively small code changes as opposed to reorganizing large sections of the code. Furthermore, we applied transformations that could be derived from the program text, rather than from knowledge of the application.

Our work included dependence analysis, privatization, and idiom recognition. As shown in Figure 1, our transformations significantly improved the performance of all programs we inspected. In fact, the performance of most of the programs was close to or as good as the performance that resulted from the best reported manual effort.[2] (These figures were derived on the Cedar machine, a multiprocessor organized into four clusters of eight processors each. The machine included memory modules accessible to all processors as well as memory shared only by the processors of a single cluster.)

We then began to implement these techniques in our new compiler, called Polaris. Polaris is a source-to-source Fortran restructurer that accepts an extension of Fortran 77 and produces parallel Fortran for global-address-space machines. A preliminary version of Polaris can parallelize half the programs in Figure 1 as well as our hand-transformations did.

Although the Perfect Benchmarks are perhaps the most widely accepted suite that represents a supercomputer workload, the question of whether our findings would carry over to other programs remained an open one. To answer this question, we collected an additional suite of programs that are now being used by researchers at the National Center for Supercomputing Applications. So far, we have carefully examined two of these six codes, and found that our newly implemented techniques can parallelize them.

Table 1 lists the most important loops of the suite of benchmark programs that we chose as a first yardstick for Polaris (Cloud3d and Cmhog are from the NCSA set; the others are from the Perfect Benchmarks). 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 where the new compiler technology can be credited with finding the parallelism.

With very few exceptions, Polaris parallelized the loops that we previously parallelized by hand. The exceptions include some inherently serial loops (indicated by "S" in the "Hand-optimized" column) and some rare patterns that can be automated but are not yet implemented ("A" in the "Polaris-optimized" column). An "I" indicates that the implementation is not yet complete but 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 loops we have not listed can be parallelized successfully as well. We do not have space to show them all here, 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

loop. However, BDNA is limited to a 30-fold speedup. To deal with this loop we need to develop transformations that work on I/O operations — an area we have not yet studied.

Symbolic analysis

Our enhanced techniques need to be done symbolically to be effective. In other words, the analysis must manipulate or propagate symbolic expressions, equations, and inequalities that contain program variables.

Polaris implements two mechanisms for symbolic propagation. One uses 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.[3] We have used the former to support symbolic data dependence tests, which can identify an important subclass of loops as parallel (which conventional data dependence tests cannot). We have used the latter to support array privatization and idiom recognition. Array privatization is one of the most important techniques needed for effectively parallelizing programs. Even though either symbolic propagation mechanism could support all three analysis techniques (at least in theory), the demand-driven approach is potentially more efficient because it derives only the information that is needed.

Symbolic dependence analysis

Many modern dependence tests are accurate and efficient,[4] but they are not applicable in many common situations. For instance, the existence of a variable with an unknown or nonconstant value in a loop bound or a subscript expression coefficient prevents the use of many tests.

Most data dependence tests require their loop bounds and array

Table 1. Transformation of the time-critical loops of our evaluation suite. P = parallel, and S = serial.

Program-routine/loop	% program execution time	Hand-optimized	Polaris-optimized	New techniques are 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	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	X
Cloud3d-s_advect/	7.7	P	P	
Cloud3d-s_advect/1	7.6	P	P	X
Cloud3d-s_advect/2	7.5	P	P	X
Cloud3d-kmsource/5	6.3	P	P	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

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

Figure 2. Before and after induction variable substitution.

```
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 Ocean-ftrvmt/109.

subscripts to be represented as a linear (affine) function of loop index variables. That is, the expressions must be in the form

$$c_0 + \sum_{j=1}^{n} c_j * I_j$$

where c_j are integer constants and 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.[5,6]

Unfortunately, not all nonlinear expressions can be removed. We had believed that this would not affect dependence testing in real programs (since nonlinear expressions would be rare in real programs), but our manual parallelization of the Perfect Benchmarks disproved this. In fact, 4 of the 12 codes (Dyfesm, QCD, Ocean, and TRFD) would have exhibited a speedup of at most 2 if we had not parallelized loops with nonlinear array subscripts.[7] 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 the sidebar, 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; this may be needed for interprocedural analysis when an array is dimensioned differently across procedure boundaries. If the declared dimensions of a multidimensional array are symbolic expressions, the resulting linearized array may contain unknown subscript terms. For example, if the array **A**, which was originally dimensioned as **A**(N,M), were linearized, its declaration will be changed to **A**($N*M$), and a reference **A**(I,J) will be changed to **A**($I + N*J$).

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*.[8] In the range test we mark a loop as parallel if there are no cross-iteration dependences caused by scalars, and if 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 the second 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 the rightmost loop in Figure 2 as parallel because the span of the range of elements of **A** generated by the I loop, which equals $N - 1$, 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.

Figure 3 shows an important loop nest that contained

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

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	...	...	$258*N$

nonlinear subscripts. This is a simplified version of a loop that 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 could not have parallelized 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. We can see this by examining Table 2, which displays the spans and strides of a permutation of the loop nest. The first version of our 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 a pseudoloop of the form `do Offset = 0, 129*N, 129*N`. In Table 2, 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 we developed the range test to complement rather than replace conventional dependence analysis, it was the only test needed to parallelize the loops listed in Table 1.

```
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.

ARRAY PRIVATIZATION

Although symbolic dependence analysis lets us prove that more references in a loop nest are independent from each other, it will not let us parallelize a significantly greater number of important loops without additional transformations. In our experience, the most important of these transformations is *array privatization*.[9]

As mentioned in the sidebar, array privatization eliminates memory-related dependences. That is, it 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 antidependences or output dependences caused by storage reuse.

To prove that a scalar variable can be privatized, every use of that variable must be dominated by a definition of the variable in the same loop iteration. The definition of a scalar 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. For scalars, if a definition dominates a use, then we can say that the definition *covers* the use.

Determining the covering definition for a use of a scalar variable is straightforward, since the scalar is an atomic object that 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 — or a collection of these — covers all array uses requires an elaborate analysis of the array ranges. More specifically, the array privatizer must prove that the region of array elements referenced is a subset of the region of array elements defined to determine that the uses are covered by the assignment. Symbolic analysis techniques are often required for these region comparisons, since the region expressions often contain unknown or nonconstant values. As mentioned in the sidebar, because of the difficulty of analyzing array references, most of today's parallelizing compilers privatize only scalars.

In many cases we can determine the ranges in the definitions and use of arrays, and whether one covers the other, 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.

Figure 4 shows an example where global information is necessary for array privatization. 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.

```
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.

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 Figure 4, each variable is assigned only once, so no renaming is necessary to obtain the SSA form. Our demand-driven algorithm proceeds backward from use to definition. To prove that $MP \geq M*P$, the algorithm starts at loop J and backward-substitutes MP with $M*P$ as defined in statement $S2$. Because the goal is satisfied, the algorithm stops at this point and performs no further replacements.

Another example of the need for global information is shown in Figure5, taken from BDNA. Several intermediate variables need to be privatized to parallelize the outermost loop: 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, we need to determine the range of the use of **A** in loop L. By analyzing the subscript and the range of the loop L, we can easily determine that the range is {**A**(**IND**(1)), **A**(**IND**(2)), ..., **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 {**A**(**IND**(1)), **A**(**IND**(2)), ..., **A**(**IND**(P))} falls in the range of **A**(1:I–1).

A demand-driven strategy works well in situations like this, where we must 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 {**IND**(1), **IND**(2), ..., **IND**(L)} are assigned in loop K.

Now that the algorithm knows the definition point for {**IND**(1), **IND**(2), ..., **IND**(P)}, it can substitute the loop variant terms in {**A**(**IND**(1)), **A**(**IND**(2)), ..., **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], {**IND**(1), **IND**(2), ..., **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.

Runtime techniques

Even the most powerful symbolic analysis techniques cannot detect parallelism if the information is unavailable at compile time. For a class of programs, compile-time analysis must be complemented with runtime 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 arrays subscripted by index arrays. 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 runtime. Because of the overhead involved, it is very important that runtime 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. Such loops require runtime techniques that are fast and effective.

DETECTING DATA DEPENDENCES AT RUNTIME

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 runtime whether the loop was, in fact, fully parallel. If the subsequent test finds that the loop was not fully parallel, then it will be reexecuted sequentially.

To implement such a strategy, we have developed a runtime technique, called the *Privatizing Doall test* (*PD test*), for detecting the presence of cross-iteration dependences in a loop.[10] The test does not identify such dependences; it only flags their existence. In addition, if any variables were privatized for speculative parallel execution, the PD test determines whether those variables were, in fact, validly privatized. We are interested in identifying fully parallel loops because they arise frequently in real programs.

```
do I = 1, n
   ...       = A(T(I))    T(1:8) = [2 2 2 10 8 8 8 10]
  A(U(I))  =...           U(1:8) = [1 3 5  4 7 3 6 12]
   ...       = A(V(I))    V(1:8) = [1 3 2 10 7 3 8 12]
end do
```

	Position in shadow arrays													
	1	2	3	4	5	6	7	8	9	10	11	12	wA	mA
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. The PD test.

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 arrays during speculative parallel execution using the access pattern of **A**. Then, after loop termination, it 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 $\mathbf{A}_w$ is marked. If, during any iteration, an element in **A** is read, but never written, then the corresponding element in the read shadow array $\mathbf{A}_r$ is marked. Another shadow array $\mathbf{A}_{np}$ is used to flag the elements of **A** that *cannot* be privatized: An element in $\mathbf{A}_{np}$ is marked if the corresponding element in **A** is both read and written, and is read first, in any iteration.

The post-execution analysis that determines whether there were any cross-iteration dependences between statements referencing **A** proceeds as follows. In the following, note that *any* returns the OR of its vector operand's elements: $any(v(1{:}n)) = (v(1) \vee v(2) \vee \ldots \vee v(n))$. If $any(\mathbf{A}_w(:) \cap \mathbf{A}_r(:))$ is true (that is, if the marked areas are common anywhere), then there is at least one flow dependence or antidependence that was not removed by privatizing **A** (some element is read and written in different iterations). If $any(\mathbf{A}_{np}(:))$ is true, then **A** is not privatizable (some element is read before being written in an iteration). Let $w_{\mathbf{A}}$ be the total number of writes that were marked in $\mathbf{A}_w$ by all iterations (computed during the parallel execution), and let $m_{\mathbf{A}}$ be the total number of marks in $\mathbf{A}_w$ (computed after the parallel execution). If $w_{\mathbf{A}} \neq m_{\mathbf{A}}$, then there is at least one output dependence (some element is overwritten); however, if **A** is privatizable (that is, if $any(\mathbf{A}_{np}(:))$ 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 loop shown in Figure 6 illustrates the PD test. The access pattern is given by the subscript arrays T, V, and U. Since $\mathbf{A}_w(:) \wedge \mathbf{A}_r(:)$ and $\mathbf{A}_{np}(:)$ are 0 everywhere, the loop was a `doall`, but only after privatizing **A** since $w_{\mathbf{A}} \neq m_{\mathbf{A}}$.

Performance of runtime techniques

If the PD test passes (that is, if the loop is 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 (the loop is not fully parallel), then the slowdown incurred is proportional to $(1/p)T_{seq}$, where T_{seq} is the sequential execution time of the loop. If the target architecture is an MPP with hundreds of processors (and someday, thousands), 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 could offer large gains in performance while risking only a small increase in the sequential execution time.

Figure 7 shows experimental results of a Fortran implementation of the PD test on loop Track-nlfilt/300 from the Perfect Benchmarks. We made these measurements on an Alliant FX/80, a modestly parallel machine with eight processors. The compiler cannot

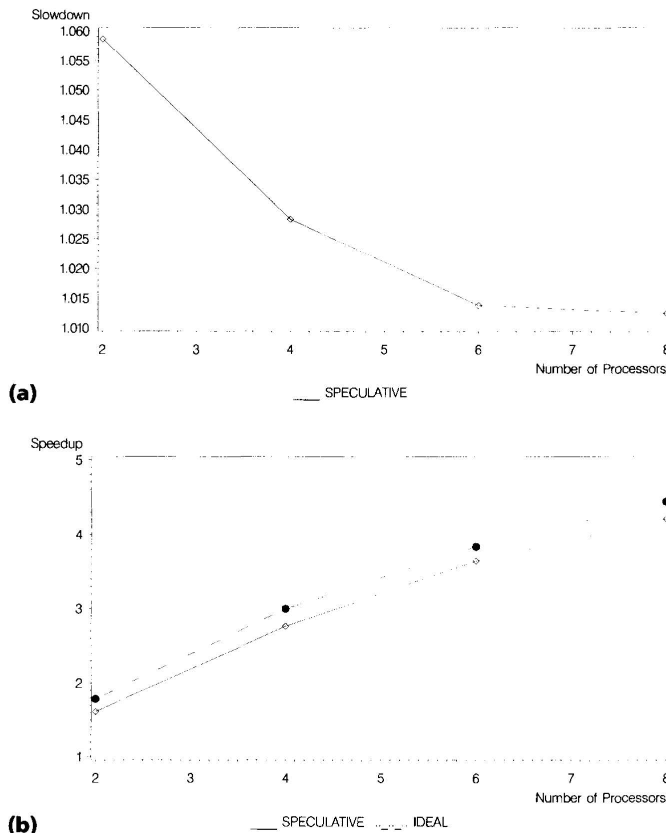


Figure 7. The speedup (a) and potential slowdown (b) of the partially parallel loop Track-nlfilt/300 from the Perfect Benchmark.

analyze the access pattern of the shared array in this loop, since the array is indexed by a subscript array that is computed at runtime. In addition, this loop is parallel for only 90% of its invocations. In the cases when the test failed, we restored state and reexecuted the loop sequentially. The speedup reported includes both the parallel and sequential instantiations.

Our 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.

Even though today's compiler techniques are limited in their effectiveness, it seems clear 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. When developing new techniques, researchers should devote a substantial effort to analyzing 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 automatic detection of parallelism is not a trivial problem, but it is not insurmountable either, and the reward for success will more than compensate for the effort.

Acknowledgments

This research is supported in part by Army contract DABT63-92-C-0033. This work does not necessarily represent the positions or policies of the US Army or the Government.

References

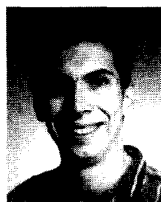
1. R. Eigenmann et al., "Restructuring Fortran Programs for Cedar," *Concurrency: Practice and Experience*, Vol. 5, No. 7, Oct. 1993, pp. 553–573; also CSRD Report No. 1338, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1990; available by anonymous ftp at ftp.csrd.uiuc.edu.
2. L. Pointer, "Perfect: Performance Evaluation for Cost-Effective Transformations, Report 2," CSRD Report No. 964, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1990; available by anonymous ftp at ftp.csrd.uiuc.edu.
3. P. Tu and D. Padua, "Demand-Driven Symbolic Analysis," CSRD Report No. 1336, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1994; available by anonymous ftp at ftp.csrd.uiuc.edu..
4. P.M. Petersen and D.A. Padua, "Static and Dynamic Evaluation of Data Dependence Analysis," *Proc. ICS '93*, ACM Press, New York, 1993, pp. 107–116.
5. W. Pugh, "A Practical Algorithm for Exact Array Dependence Analysis," *Comm. ACM*, Vol. 35, No. 8, Aug. 1992, pp. 102–114.
6. M. Haghighat and C. Polychronopoulos, "Symbolic Dependence Analysis for High-Performance Parallelizing Compilers," in *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau et al., eds., MIT Press, Cambridge, Mass., 1991, pages 310–330.
7. W. Blume and R. Eigenmann, "Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks," CSRD Report No. 1332, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1994; available by anonymous ftp at ftp.csrd.uiuc.edu.; also presented at the 1994 Int'l Conf. Parallel Processing.

8. W. Blume and R. Eigenmann, "The Range Test: A Dependence Test for Symbolic, Nonlinear Expressions," CSRD Report No. 1345, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1994; available by anonymous ftp at ftp.csrd.uiuc.edu.; also to be presented at Supercomputing '94.

9. P. Tu and D. Padua, "Automatic Array Privatization," *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Vol. 768, Aug. 1993, pp. 500–521.

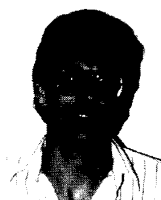
10. L. Rauchwerger and D. Padua, "The Privatizing Doall Test: A Runtime Technique for Doall Loop Identification and Array Privatization," CSRD Report No. 1329, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1994; available by anonymous ftp at ftp.csrd.uiuc.edu

David Padua is an associate professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He was an associate director of the Center for Supercomputing Research and Development from 1990 to 1993. His current research focuses on the experimental analysis of parallelizing compilers, and techniques to make them more effective. Padua is coeditor of the *International Journal of Parallel Programming*, and he is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, and the *Journal of Programming Languages*. Padua received his PhD in 1980 from the University of Illinois at Urbana-Champaign, and his Licenciatura in computer science in 1973 from the Universidad Central de Venezuela. He is a senior member of IEEE.



William Blume is a PhD candidate in computer science at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interests include parallelizing compilers, parallel architectures, and software engineering. He received his MS and BS in computer science in 1992 and 1989 from the University of Illinois at Urbana-Champaign.

Paul Petersen is a member of the compiler group at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interests include parallelizing compilers, dependence analysis, and dependence profiling techniques for optimization. He received his PhD and MS in 1993 and 1989 from the University of Illinois at Urbana-Champaign, and his BS in computer science in 1986 from the University of Nebraska-Lincoln.



Rudolph Eigenmann is a visiting research associate professor at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He led the Cedar Fortran compiler group at CSRD during the Cedar project, and he represents the university at the High-Performance Steering Committee of the Standard Performance Evaluation Corporation. His research interests include parallelizing compilers, parallel software engineering, performance evaluation, and parallel architectures. He received his PhD in computer science and his diploma in electrical engineering from ETH Zurich.

Lawrence Rauchwerger is pursuing a PhD in computer science at the University of Illinois at Urbana-Champaign. His research interests include parallelizing compilers, instruction-level parallelism, and performance evaluation. He received his MS in electrical engineering from Stanford University, and his Diploma Engineer degree in electronics and telecommunications from the Polytechnic Institute, Bucharest. He is a member of IEEE and ACM.

Jay Hoeflinger is a member of the compiler group at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interests include optimizing compilers for supercomputers, program transformation tools, and transformations for program restructuring. He received his MS and BS in computer science in 1977 and 1974 from the University of Illinois at Urbana-Champaign.

Peng Tu is a PhD candidate in computer science at the University of Illinois at Urbana-Champaign. His research interests include program transformation for concurrency and locality, symbolic evaluation, and high-level optimization. He received his MS and BS in computer science in 1984 and 1980 from Shanghai Jiao Tong University, Shanghai.

Readers can contact the authors at the Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana, IL 61801; Internet: hoefling@csrd.uiuc.edu