Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs

Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois

Abstract.

This paper discusses the techniques used to hand-parallelize, for the Alliant FX/80, four Fortran programs from the Perfect-Benchmark suite. The paper also includes the execution times of the programs before and after the transformations. The four programs considered here were not effectively parallelized by the automatic translators available to the authors. However, most of the techniques used for hand parallelization, and perhaps all of them, have wide applicability and can be incorporated into existing translators.

1. Introduction

It is by now widely accepted that in many real-life applications, supercomputers have been unable to deliver a reasonable fraction of their peak performance. An illustration of this is provided by the Perfect Benchmark programs [3], many of which effectively use less than 1% of the computational resources available in the most powerful supercomputers. While it is apparent that the reason for this dismal behavior is sometimes the result of the machine organization and the algorithms used, we believe that much better performance could be achieved by using more powerful compilers than those available today. In this paper we present some evidence to back this claim.

During the last couple of years, our group at CSRD has developed a modified version of the KAP parallelizer with optimizations for the Cedar machine [4]. We ran several simple kernels through our parallelizer and the results were satisfactory. However, when we

This work was supported by the U.S. Department of Energy under grant no. DOE DE-FG02-85ER25001

ran the Perfect Benchmark programs, we obtained limited success or just complete failures in several of the codes for reasons that have nothing to do with the Cedar architecture or with Cedar-specific transformations, but are instead related to the general capability of the existing technology to detect and exploit parallelism. This raised questions on the effectiveness of the restructuring compiler approach in general and on the compiler framework we and many others have been using for many years. To determine the cause of the difficulties and to try to design strategies to overcome them, we decided to do hand-analyses of some of the Perfect Benchmark programs and of their restructured versions produced by the two restructuring compilers available to us: the Alliant Fortran compiler and a 1988 version of KAP.

We proceeded on a loop-by-loop basis and in each loop we tried to determine why the parallelizer had failed and then hand-manipulated the loop to exploit implicit parallelism. This manipulation was done using new transformations, most of which we believe can be easily incorporated in the parallelizers of today. Many, but not all, of the transformations we used are relatively simple extensions of well known techniques. We concentrated our work on loop parallelism because most of the existing compiler methodology is aimed at loop parallelization.

Because hand parallelization is cumbersome, we decided in this first experiment not to use Cedar [6] as our target machine but rather to use the Alliant FX/80 which is the main building block of Cedar and therefore has a simpler organization than the complete system. The Alliant FX/80 is an eight-processor shared-memory multiprocessor which includes pipelined units to process vector instructions. A speedup greater than eight can sometimes be obtained in this machine because of its two levels of parallelism.

In this paper we report the results of the manual analysis and transformation of four of the Perfect Benchmarks programs. The names and some of the characteristics of these are shown in Table 1. Table 2 shows the rather low speedups obtained from the parallelization option of the Alliant Fortran compiler. Similar speedups were obtained with a 1988 version of the KAP parallelizer.

We obtained a speedup about five times larger after manually transforming the programs, as shown in Table 3. These speedups can probably be improved by using better techniques or by a more careful analysis. However, we believe that the most important contribution of this paper is not the specific speedup values obtained but presenting examples

for which meaningful speedup improvements can be achieved by using techniques that could be added to existing parallelizers.

In the rest of the paper we describe the techniques we used to transform those loops that remain practically unchanged after processing by the automatic parallelizers. Table 4 indicates which transformation we applied to the most important loops of the programs in Table 1 to obtain the speedups in Table 3.

2. Compiler Techniques

In the discussion below, we assume that the reader is familiar with basic parallelizing compiler concepts including the notion of dependence, dependence analysis techniques and some basic transformations. Introductory material on parallelizing compilers can be found in [8], [9], and [11]. Also, all the parallel programs presented in the examples below use extensions similar to those described in the preliminary draft of the X3H5 ANSI committee developing parallel constructs for high level programming languages.

2.1. Run-Time Dependence Analysis

Many of the dependence analysis techniques used in today's compilers need the values of the coefficients in the subscript expressions. Most often, the precise set of values of these coefficients is needed, even though in some cases a range of values is sufficient. When the coefficient values are not available at compile-time, parallelizing compilers usually assume dependences exist wherever they are possible.

This assumption was particularly unfortunate for the OCEAN program from the Perfect suite. The major loop in the program (and several of the less important loops) uses variables as coefficients in array subscript expressions, producing potential dependences. Still worse, the array being subscripted is a singly-dimensioned array, yet the subscripting expressions are linear combinations of the three loop indices. However, the actual values used for the coefficients at runtime make the loops perfectly parallelizable.

We attempted to expose this parallelism by doing the dependence test at runtime, when all coefficients are known. The test takes the form of a conditional statement which chooses between versions of the loop. If a dependence exists in the loop, the serial version of the loop is run. Otherwise, the parallel version of the loop is run.

The idea is to test whether the singly-dimensioned array is being used as a linearization of a multi-dimensional array. The terms of the subscripting expression are checked in order, from least significant to most significant, to see if they meet the criteria for being such a linearization. If the array *is* being used in this linearized fashion, then all subscript values are unique within the nest, and the loops may be parallelized.

The parallelizing technique used was to

1) normalize all loops in the loop nest.
2) obtain the subscripting expression for the array used in the nest.
3) construct a run-time check testing the expression for being a proper linearization of a multi-dimensional array.

Suppose the following expression is the subscripting expression:

$$I_1K_1 + I_2K_2 + \ldots + I_nK_n$$

in which I_i is the index for (normalized) loop *i* and K_i is the coefficient of that index in the subscripting expression. Assume that I_1K_1 is the least significant term in the expression, I_2K_2 is the next most significant term, and so on, with I_nK_n being the most significant term. Or, stated another way,

$$|K_1| < |K_2| < \ldots < |K_n|$$

The values for each index run from 0 up to some maximum:

$$I_i = [\ 0\ ..\ I_{i_{max}}\ ]$$

with a stride of 1.

The tests which we need to make are simply:

$$\text{For all } j < n,\ \sum_{i=1}^{j}(|I_{i_{max}} K_i|) < |K_{j+1}|$$

Or, stated another way, the sum of the maximum values which each term can take on, up to a given term, must be less than the coefficient of the next-most significant term.

To illustrate the technique, we present a simple two-dimensional loop with its translation. The translation includes a test for the linearization condition.

```
do i=1,N
  do j=1,M
     a(K*i+L*j)=b(i)
  end do
end do
```

The following shows the translation of the loop:

```
loop_is = serial
if ( abs(K) .lt. abs(L) ) then
    if ( abs(K*(N-1)) .lt. abs(L) ) loop_is = parallel
else
    if ( abs(L*(M-1)) .lt. abs(K) ) loop_is = parallel
endif

if (loop_is .eq. parallel) then
  parallel do i=1,N
    parallel do j=1,M
       a(K*i+L*j)=b(i)
    end parallel do
  end parallel do
else
  do i=1,N
    do j=1,M
      a(K*i + L*j) = b(i)
    end do
  end do
endif
```

This test is not an exact test in that it will not always find parallelism where it exists for all possible combinations of K, L, M, and N. It *is* exact when a one-dimensional array is a proper linearization of a higher-dimensioned array. The advantage of using the linearization test rather than an exact test is that the linearization test is trivially automatable, while exact tests are not. For example, although the construction of a test involving the solution to Diophantine equations is automatable in two dimensions, we have not found a way to automate it for higher dimensions. We assume, although we have no proof, that in the majority of cases where an expression involving several loop indices is used to subscript a single dimension, the array is a linearization of a higher-dimensioned array. If that is the case, then the linearization test is a suitable substitute for exact tests.

The time consumed by a dependence analysis test is clearly much more important at run-time than at compile-time. A run-time test whose running time is a large fraction of the execution of the loop itself may prove too costly to justify its use, even if it uncovers a parallel loop nest. Unfortunately, today we lack techniques to decide which potential dependences to test at runtime.

2.2. Array Privatization

Many of the recently-developed parallel Fortran extensions such as Cedar Fortran[5], the IBM parallel Fortran, and the Parallel Computing Forum (PCF) Fortran include parallel loop constructs with private declarations. A copy of each private item is allocated for each processor cooperating in the execution of the loop. For example, the loop

```
      do i=1,n
S1:       a=b(i)+c(i)
S2:       d(i)=a**2
      end do
```

can be transformed into parallel form by declaring a private

```
      parallel do i=1,n
          private a
S1:           a=b(i)+c(i)
S2:           d(i)=a**2
      end parallel do
```

Privatization makes parallelization possible by replicating a, and in this way avoiding the two cycles, caused by memory-related dependences (i.e. anti- and output dependences), in the dependence graph of the original loop. An alternative to scalar privatization is scalar expansion which achieves the same effect as privatization by replacing all occurrences of a scalar variable with an array element whose subscript is the loop index. However, expansion may consume an unnecessary amount of memory, since all that is needed when parallelizing a loop is a copy of each expandable scalar for each processor cooperating in the parallel execution of the loop.

In the previous example we tacitly assumed that the value of a when the loop completes is not used anywhere else in the program. If a is used after the loop, a last-value assignment has to be performed. A way to do this for the previous loop is as follows:

```
      parallel do i=1,n
            private a'
            if (i.lt.n) then
S1':              a'=b(i)+c(i)
S2':              d(i)=a'**2
            else
S1:               a=b(i)+c(i)
S2:               d(i)=a**2
            end if
      end parallel do
```

The private variable a' is used in all the iterations of the parallel loop except for the last one where a is used instead. In this way, a will have the expected value just after the loop completes.

Even though the previous discussion dealt with scalars, all that was said applies also to arrays. However, the parallelizers we used were only capable of doing scalar privatization, not array privatization. For this reason they failed to parallelize several loops in the programs we analyzed as can be seen in Table 4.

Despite its apparent importance, very little has been published on array privatization even though scalar expansion is frequently mentioned and sometimes carefully described. The criterion for privatizing a scalar is that all accesses to the scalar be either a store or be preceded (in the control-flow sense) by a store inside the loop body. The equivalent criterion for array privatization is that each element of the array be assigned before it is used in the loop body. Detecting whether an array can be privatized is more difficult than in the scalar case because in the first case it is necessary to determine the array sub-range accessed by each statement in the loop body.

Clearly, there will be some cases where the compiler will be unable to determine whether privatization is or is not possible, and last value assignment could present difficulties if not all loop iterations assign to the same set of array elements. However, all cases we found when transforming the programs in Table 1 (array privatization was useful in transforming the four programs discussed here), were relatively straightforward, and there is no doubt that a parallelizing compiler can perform the same array privatizations we did by hand.

Interprocedural analysis is an important complementing technique to detect arrays that can be privatized. Many of the loops we parallelized contain subroutine calls. As a result,

definitions and uses of the candidate arrays had to be searched across procedure bounds.

2.3. Generalized Induction Variables

The sequence of values that an induction variable receives throughout the execution of a loop form an arithmetic progression [1]. Most often, a new value is assigned to an induction variable, say V, in statements of the form

```
V = V + K
```

where K is a value that remains constant throughout the execution of the loop. To eliminate the loop-carried dependences generated by this statement, parallelizing compilers replace the occurrences of the induction variable with a linear function of K and the loop index (or indices). For example, variable j in statement S2 of the loop

```
      j=0
      do i=1,n
S1:       j=j+m
S2:       b(i)=j+d(i)
      end do
```

can be replaced by m*i. Formally, this means that the cycle in the dependence graph caused by the use of j in S2 will no longer exist, and the assignment to b(i) can execute in parallel. Statement S1 can then be eliminated if j is never used again after the loop completes. Otherwise it should be replaced by a new assignment j=n*m to be placed immediately after the transformed loop.

Another important class of variables, which we will call here *generalized induction variables*, are those that assume a sequence of values which do not necessarily constitute an arithmetic progression but for which a closed-form expression can be computed at compile-time. In our work we found only two types of GIVs. The first type is updated using multiplication instead of the addition typical of the regular induction variable. The sequence of values generated in this way forms a geometric progression and clearly the same machinery used for regular induction variables can be used to process this type of GIV. As can be seen in Table 4, one loop in the program OCEAN could be parallelized thanks to the recognition of this type of GIV.

The second type of GIV consists of those variables whose sequence of values do not form an arithmetic progression even though they are updated using constant increments. For example, in the following loop

```
        j = 0
        do i=1,n
              ...
S1:           j=j+1
              ...
              do k=1,i
                    ...
S2:                   j=j+1
                    ...
              end do
        ...
        end do
```

the occurrences of j after S1 and before the inner loop could be replaced by

$$1+\sum_{i=1}^{I-1} 1+\sum_{i=1}^{I-1}\sum_{k=1}^{i} 1 = I+I*(I-1)/2,$$

where I is the value of the index i at that point. The occurrences between S2 and the end of the inner loop could be replaced by

$$I+I*(I-1)/2+K$$

where I and K are the current values of the loop indices.

Generalized induction variables of this second type could be incremented by a constant as in the previous example or by another induction variable.

As shown in Table 4, we found some generalized induction variables of this second type in the program TRFD. All were decoupled. Because the existing parallelizers can only handle regular induction variables we had to find the closed-form expression by hand. The methodology used was to compute the contribution of each assignment statement by symbolically evaluating a (possibly nested) summation. The expressions were then formed as the sum of the contributions of all the assignment statement in the loop body. A symbolic algebra package, Maple, was used to help our hand calculations, and clearly, symbolic algebra algorithms will be needed to implement powerful GIV transformation algorithms.

One important limitation of today's symbolic algebra packages arises when an inner loop containing an increment to a GIV is not executed in all the iterations of the outer loop

because in some cases its upper limit is less than its lower limit. In this case, the contribution of the increment operation has the form of a nested summation where one of the inner summations have an upper bound less than the lower bound. The value of this inner sum has to be defined as zero in order to obtain the right answer to our problem. However, the symbolic algebra packages we tried make other assumptions, and therefore they produced incorrect results in these cases. However, zero-trip loops have to be considered when developing a generally-applicable GIV algorithm.

An important observation is that when the GIVs are used as subscripts it is necessary to detect the monotonicity of the sequence of values for the purpose of dependence analysis. This has to be done before the GIV is replaced by a closed form expression because these could be non-linear on the loop indices, and current dependence analysis techniques do not work on non-linear subscript expressions. In the programs discussed in this paper, detecting the monotonicity of the sequence of values was necessary to perform an accurate dependence analysis. This was accomplished by analyzing the original assignment statements. In our case, monotonicity could be guaranteed because the GIVs were always incremented by values of the same sign.

2.4. Doacross transformations

A doacross is a parallel loop, automatically generated from a sequential version, where synchronization takes place in the form of a cascade. In other words, a wait in iteration i can only be triggered by an event in an iteration j which precedes i in the sequential version of the loop. For example, one way in which the loop

```
      do i=1,n
S1:       a(i) = b(i) + a(i-1)
S2:       c(i) = b(i) + e(i-1)
S3:       e(i) = c(i) + 1
S4:       d(i) = e(i) ** 2
      end do
```

can be executed in parallel is by executing the π-blocks with cross-iteration dependences ({S1} and {S2, S3}) sequentially across the iteration space. It is usually said that each such π-block is enclosed in an *ordered critical section*. In this type of translation, parallelism is achieved by overlapping the execution of the different π-blocks as illustrated in the following translation of the previous loop.

```
      post (s1(0))
      post (s2(0))
      parallel do (ordered) i=1,n
            wait (s1(i-1))
S1:           a(i) = b(i) + a(i-1)
            post (s1(i))
            wait (s2(i-1))
S2:           c(i) = b(i) + e(i-1)
S3:           e(i) = c(i) + 1
            post (s2(i))
S4:           d(i) = e(i) ** 2
      end parallel do
```

In doacross loops, when only assignment statements are considered, the translator only has to consider issues related to synchronization and statement reordering. However, the presence of conditional statements introduce other difficulties. In this section, we discuss two transformations that we found useful when dealing with conditional statements in the loop body.

The first transformation operates on do loops which could potentially terminate by executing conditional goto statements targeted outside the loop body. Several strategies can be used to parallelize these loops. The technique we used in TRACK consists in letting all the iterations of the loop run in parallel and storing all the results only in temporary locations. A boolean vector is used to keep a record of which iterations execute the exit goto statement.

After the parallel loop we generated a second parallel loop which stores the values computed in the first loop in their definitive locations. For example:

```
      do i=1,n
            a(i) = b(i) ** 2
            if a(i) < 5 then go to 5

      end do
      ...
5     ...
      ...
```

can be transformed into

```
parallel do i=1,n
      a'(i) = b(i) ** 2
      if a'(i) < 5 then bool_vector(i) = .true.

end parallel do
parallel do i=1,first_true(bool_vector(1:n))
      a(i) = a'(i)
end parallel do
```

where first_true when applied to a boolean vector v computes the location of the first true value in v.

The second doacross transformation to be discussed in this section operates on loops of the form

```
do i=1,n
      if a(b(i)) < k then to to L
      ...
      a(b(i)) = a(b(i)) - 1
      ...
L:    ...
      ...
end do
```

where there is an if statement whose boolean expression is persistent, i.e. if it is true in iteration k, then it is also true in all iterations j > k. In a parallel execution of the loop, such an if statement will have to wait for all previous iterations only if at the time of evaluating it, its boolean expression is found to be false.

For example, the previous loop can be transformed into the parallel loop

```
parallel do (ordered) i=1,n
      if a(b(i)) < k then
            post (ev(i))
            go to L
      end if
      wait(ev(i-1))
      if a(b(i)) < k then go to L
      ...
      a(b(i)) = a(b(i)) - 1
      post (ev(i))
      ...
L:     ...
      ...
end parallel do
```

where a wait is only necessary when the boolean expression a(i) < k is false. This transformation was applied to a loop of the program TRACK and it accelerated the program by a factor of 1.8. A related transformation strategy is described in [7].

2.5. Unordered Critical Sections and Parallel Reductions

The techniques described in this section allow the transformation of do loops into parallel form by exploiting the fact that we can reorder the execution of certain operations even thougth they are involved in data dependences.

The first case is illustrated by the increment operation to a(k) in the following loop.

```
do i=1,n
      do j=1,m
      S1:    k=funct(i,j)
            ...
      S2:    a(k) = a(k) + <expression>
      end do
end do
```

We will assume that S2 is the only statement accessing an element of array a in the loop. This assumption is not necessary to apply the technique, but it helps in the discussion. Depending on the function funct(i,j) it may be possible to transform the outer do loop into a doacross. For example, if funct(i,j) is just j then different iterations of the outer loop could

proceed in parallel and synchronization will only take place between assignments to the same array element as shown next.

```
parallel do j=1,m
  post (e(0,j))
end parallel do
parallel do (ordered) i=1,n
      do j=1,m
            ...
            wait (e(i-1,j))
            a(j) = a(j) + <expression>
            post(e(i,j))
      end do
end parallel do
```

However, if we are willing to allow the order of the additions to change, we can apply a more general transformation that will work for any form of the function funct and will not require enforcing a particular order to the different assignments to a(j) for a given j. This transformation consists of surrounding the assignment statement with a critical section which guarantees that, at any time, only one processor updates each element a(k):

This transformation is called in this paper *unordered reduction*, and its main advantage over the doacross transformation is that it does not require total serialization of the assignment statement even when the subscripts are non-linear or include unknown terms. Unordered reduction only requires that the modification to each array element proceed serially. In the traditional doacross transformation, if the subscripts are not analyzable at compile-time, the statement will have to be executed serially across all loop iterations. In some cases, the transformation may also produce a faster parallel loop than the doacross transformation because it allows total flexibility in the scheduling of the iterations and their critical sections.

In some cases we used an alternative way of implementing the unordered reduction. Instead of synchronizing the update of variable a(k), we accumulated the values on each processor separately and summed up the partial results after the loop. The resulting loop is completely parallel, at the cost of an additional sum operation. This is illustrated in the following example.

```
dimension temp(nr_processors,maxk)
temp(1:nr_processors,1:maxk) = 0
parallel do i=1,n
    do j=1,n
        k=funct(i,j)
        ...
        temp(my_proc,k) = temp(my_proc,k) + <expr>
    end do
end parallel do
parallel do k=1,maxk
      a(k) = a(k) + SUM(temp(1:nr_processors,k))
end parallel do
```

Unordered reductions were used in both MDG and TRFD, and it is likely that it will find use in a relatively large collection of numerical programs. Simple reduction operations are usually recognized and parallelized by available compilers, such as a single loop statement summing into a scalar element. A more complex pattern is that of multiple loop statements summming into the same scalar. Another example is the accumulation into array elements, where sometimes the array index is unknown.

The second type of transformation which we call *unordered assignment* is illustrated by the assignment to x(k) in the loop

```
do i=1,n1
      do j=1,n2
            k=funct(i,j)
            ...
            x(k) = <expression>
      end do
end do
```

This is very similar to the previous case, except that instead of accumulating on an array element successive iterations may rewrite the array element. As in the previous case we assume that there are no other references to the array x in the loop. As can be inferred from the discussion that follows it is not difficult to extend this transformation if x appears in other statements but only on the left-hand side. The technique, however, will not work if x also appears on the right-hand side (see [10] for a discussion of this case).

As in the previous case, a doacross transformation could be used. However, the resulting loop may not be efficient if funct cannot be analyzed at compile-time. However, because x(k) is not read in the loop body an assignment to x will not affect the correctness

as long as each element of x has, at the end of the loop, the value assigned in the last iteration accessing the element. A parallel version of the loop is shown next:

```
order = 0
parallel do i=1,n1
      private k
      do j=1,n2
            k = funct(i,j)
            ...
            critical section c(k)
                  if order(k) <= i then
                        x(k) = <expression>
                        order(k)=i
                  end if
            end critical section
      end do
end parallel do
```

In this loop, the assignment to x(k) is surrounded by a critical section controlled by a lock variable, c indexed by k. An integer array, order, is also introduced in such a way that order(k) at any point during the execution of the loop will contain the value of the last iteration where x(k) was assigned (or zero if x(k) has not been accessed). Inside the critical section, the value of order(k) is compared against the current iteration. If it is lower, it means that either x(k) has not been assigned yet or was assigned in an iteration that precedes the current one in the sequential version of the loop. In this case, x(k) should be assigned a new value. On the contrary, if order(k) is greater than the current iteration, then x(k) should not be assigned in this iteration because an iteration with a higher value has already been assigned to it. In the sequential program, a later iteration would have rewritten the variable.

2.6. Assignment Compaction

In this technique, used in TRACK, we assume the subscript values cannot be computed independently in each iteration, but are conditionally incremented in each iteration under the control of an if statement. A typical situation is shown in the next loop.

```
do i=1,n1
      do j=1,n2
            ...
            k=m+1
            ...
            a(k) = <expression-1>
            ...
            if <expression-2> then m=k
      end do
end do
```

To allow the loop to execute in parallel, <expression-1> could be stored in a temporary two-dimensional array, say ta, and a two dimensional boolean array, say ma can be used to keep track of the iteration in which k is assigned to m.

```
parallel do i=1,n1
      do j=1,n2
            ...
            ta(i,j)=<expression-1>
            ...
            if <expression-2> then ma(i,j)=.true.
                        else ma(i,j)=.false.
      end do
end parallel do
```

After the loop completes, ma can be used to gather ta into array a. This can be expressed in Fortran 90 using the pack intrinsic function:

```
a(m+1:):=pack(ta(1:n1,1:n2), mask=ma)
```

3. Conclusions

The study reported in this paper deals with a small sample of programs. Some of the transformations we present here were useful in only one of the programs. Nevertheless, the resulting speedup of those programs demonstrates that there is much room for improvement in the existing parallelizing compiler methodology. Our strategy to demonstrate this was to hand-analyze the output of existing parallelizers, study the reasons for their failures, and then develop strategies to overcome some of these limitations. We believe that this is a good approach to advance compiler technology.

Even though this is an obvious strategy, a careful reading of the compiler literature reveals surprisingly few experimental compiler studies. Much of the literature focuses on improving the efficiency of existing compiler algorithms or on the development of techniques with very few experimental results to demonstrate their effectiveness. Much more experimental work is needed. We believe that only through the analysis of a large collection of programs will it be possible to develop successful techniques applicable across a wide spectrum of programs.

References

[1] Alfred Aho and Jeffrey Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. 2.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[2] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA. 1988.

[3] George Cybenko, Lyle Kipp, Lynn Pointer and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks^{TM}. *Proceedings of ICS, Amsterdam, Netherlands*, March 1990.

[4] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li and David Padua. Restructuring Fortran Programs for Cedar. *Proc. of the Int. Conf. on Parallel Processing*, pp. I 57-66, August 1991.

[5] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger and Duncan H. Lawrie. Cedar Fortran and Other Vector and Parallel Fortran Dialects. *Jour. of Supercomputing*, Vol. 4, No. 1, pp. 37-62, March 1990.

[6] David Kuck, Edward Davidson, Duncan Lawrie and Ahmed Sameh. Parallel Supercomputing Today and the Cedar Approach. In: *Experimental Parallel Computing Architectures*, J. J. Dongarra, ed. Elsevier Science Publishers B.V. (North-Holland), New York, NY, pp. 1-20, 1987.

[7] Zhiyuan Li. Compiler Algorithms for Event Variable Synchronization. *Proceedings of ICS 91*, pp. 85-95, June 1991.

[8] David Padua and Michael Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, Vol. 29, No. 12, pp. 1184-1201. December 1986.

[9] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press. Boston, MA. 1989.

[10] Chuan-Qi Zhu and Pen-Chung Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Trans. on Software Eng.*, Vol. SE-13, No. 6, pp. 726-739, June 1987.

[11] Hans Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press. New York, NY. 1991.

Table 1. The names of four of the codes in the Perfect Benchmarks, their application areas, the dominant algorithms in the codes, and the number of lines of Fortran source code.

Code	Application Area	Dominant Algorithm(s)	Lines of Fortran source
MDG	Chemical & Physical Model	ODE Solvers	1238
OCEAN	Fluid Dynamics	FFTs	4343
TRACK	Signal Processing	Convolution	3784
TRFD	Chemical and Physical Model	Integral Transforms	435

Table 2. Speedups produced by the Alliant FX/80 Fortran compiler.

Code	Speedup
MDG	1.1
OCEAN	1.42
TRACK	0.90
TRFD	2.36

Table 3. Speedups on the Alliant FX/80 after our hand transformations.

Code	Speedup
MDG	5.5
OCEAN	8.3
TRACK	5.1
TRFD	13.2

Prog	Label	Routine	Loop Spdup	% of Seq	Array Privatization	GIV (additive)	GIV (multiplicative)	Unordered Reduction	Unordered Assignment	DOACROSS Conditional Stmts	DOACROSS Exit	Runtime Dependence Test	Multi-version Loops	Assignment Compaction	Interprocedural Analysis
TRACK	300	nlfilt	5.2	40%	x					x					
	300	fptrack	6.0	9%							x			x	
	400	extend	7.0	34%					x		x			x	
MDG	1000	interf	6.0	90%	x			x							
	2000	poten	5.2	8%	x										
TRFD	100	olda	16.4	69%	x	x									
	300	olda	12.3	29%	x	x									
	140	intgrl	5.5	1%					x						
OCEAN	109	ftrvmt	8.1	40%			x					x	x		
	20	csr	6.6	4%								x	x		
	30	acac	3.8	3%								x	x		
	40	acac	6.1	2%								x	x		
	30	scsc	9.6	2%								x	x		
	20	rcs	6.6	1%								x	x		
	116	ftrvmt	5.0	4%								x	x		
	270	ocean	8.0	3%	x										
	480	ocean	6.1	4%	x										
	500	ocean	6.5	3%	x										
	340	ocean	5.5	4%											x
	360	ocean	6.3	2%											x
	400	ocean	6.0	2%											x
	420	ocean	5.8	3%											x
	440	ocean	7.4	3%											x
	460	ocean	5.1	3%											x

Table 4. Summary of transformations used.