Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks^{TM} Programs [*]

William Blume and Rudolf Eigenmann
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

We have studied the effectiveness of parallelizing compilers and the underlying transformation techniques. This paper reports the speedups of the Perfect Benchmarks^{TM} codes that result from automatic parallelization. We have further measured the performance gains caused by individual restructuring techniques. Specific reasons for the successes and failures of the transformations are discussed, and potential improvements that result in measurably better program performance are analyzed. Our most important findings are that available restructurers often cause insignificant performance gains in real programs and that only few restructuring techniques contribute to this gain. However, we can also show that there is potential for advancing compiler technology so that many of the most important loops in these programs can be parallelized.

Keywords: Automatic parallelization, restructuring techniques, effectiveness analysis, compiler evaluation, Perfect Benchmarks

1 Introduction

1.1 Motivation and Goals

Over the years a large number of techniques have been developed to transform a sequential program so that it runs efficiently on a parallel architecture [27]. Many of these techniques have been incorporated into the compilers of these machines. These compilers are known as parallelizing compilers.

Despite the wealth of research on new restructuring techniques, little work has been done on evaluating their effectiveness. Many early studies measured the success rate of automatic vectorizers on a suite of test loops [4, 6, 7, 10, 23, 26]. The need for more comprehensive studies has been pointed out, and more recent work has measured performance results of automatic parallelizers on a representative set of real programs [8, 13]. However, very few papers have reported evaluation measures of individual restructuring techniques [9, 13]. This paper extends the measurments presented in [13]. We measure both overall performance improvements from automatic parallelization and the contribution of individual restructuring techniques. The measurements are taken on an Alliant FX/80 machine using the Perfect Benchmarks programs

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

and the two compilers, *Kap* and *Vast*. Variants of these compilers were the primary subject of investigations in the evaluation studies cited above.

The result of a comprehensive study of techniques used by restructurers will benefit both academia and industry. It can determine the importance of individual techniques for improving program speed, which can help an engineer to weigh the costs of implementing a technique in a parallelizing compiler against the performance loss of excluding it. For researchers the results would guide research toward more important techniques and away from the less effective ones. The study will also uncover problems with existing techniques and point toward the need for new ones.

We measure the effectiveness of existing parallelizing compilers on real programs as well as the restructuring techniques they use. The overall effectiveness of the restructurers are measured by the speedup in execution time from that of the serial code. The effectiveness of individual techniques is then determined. The performance figures for both the compilers and the restructuring techniques they use are analyzed to determine the underlying causes of such figures. Finally, potential improvements are discussed.

1.2 Caveats

As in all benchmarking reports, our measurements will be biased toward the machine and the compilers used, and inaccuracies from run-to-run variations. The program suite that we use as a representation of the "real world" is the Perfect Benchmarks^{TM} suite [29], and we are subject to all its caveats. These are: missing I/O information, compromised data sets, and missing throughput measures. Furthermore, one can always question the representativeness of these codes.

There is no ideal way to avoid these problems. We introduce important characteristics of the machine used for the measurements, and we add some performance numbers from other compilers so one can determine the bias of our results. Furthermore, we only interpret effects that are above a certain threshold of significance.

We believe that the results of real measurements are important enough to outweigh the mentioned shortcomings. For us, what has been learned so far is important. It has given us a valuable view of the effectiveness of compiler technology and suggestions for its future research and development.

2 Scope of Measurements

2.1 The Perfect Benchmarks^{TM} Suite

The Perfect Benchmarks^{TM} (**PERF**ormance **E**valuation for **C**ost-effective **T**ransformations) is a suite of 13 Fortran 77 programs that total about 60,000 lines of source code. They represent applications in a number of areas of engineering and scientific computing. In many cases they represent codes that are currently used by computational research and development groups.

We do not include code excerpts in this report. However, for the interested reader we will refer to the codes, their subroutines, and loop numbers in our discussion. We use the notation (foo/10) to refer to the Fortran `DO` loop 10 in subroutine foo. The Perfect Benchmarks are publicly available.[1]

Three codes in the Perfect Benchmarks needed modifications so that they could be compiled successfully. Directives were inserted into SPEC77 to prevent certain loops from being vectorized. Because these loops account for less than 1% of the code's execution time, this modification does not alter its performance characteristics. Directives that prevent transformation of loops into concurrent form were inserted into certain subroutines of SPICE. The affected loops account for only 2% of the code's execution time. Finally, MG3D was modified so that a large temporary file was kept in memory. We believe that these changes alter our findings insignificantly.

2.2 Machine used

We have taken our measurements on an Alliant/FX80 system, which consists of eight vector-processors that share both the main memory and its cache. The peak performance is 94 MFlops, our memory size is 96 MBytes, and the cache size is 512 kBytes. The vector pipelines are four stages long. Thus the hard upper limit in speedups is 32 (4 for vectorization and 8 for concurrent execution).

Figure 1 shows the speedups we observed in an algorithm[2] that can be fully vectorized and concurrentized. We can see that vectorization yields a speedup of about 3.5. Vector-concurrent execution runs up to 12.5 times faster than serial, which is less than half of the limit mentioned above. Concurrent execution without vectorization yields a best speedup of about 3.5 as well. We have seen some variations in these numbers depending on how well a parallel algorithm exploits the cache. Other factors, such as vector register allocation and memory access patterns, may also affect program speeds. The figures will help us later in interpreting the performance gains we get in large programs.

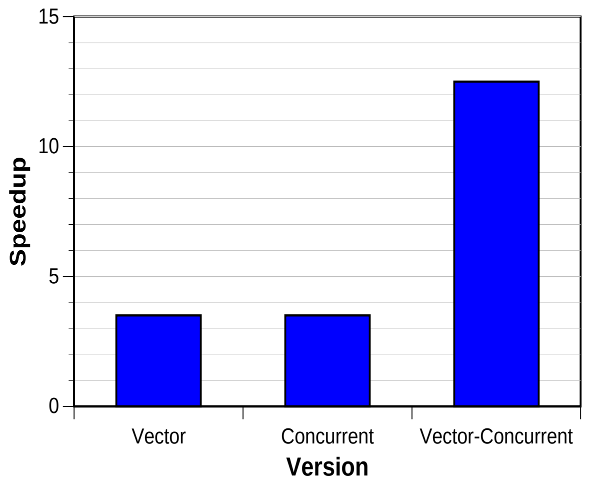

Figure 1: Typical speedups of an ideally parallelizable algorithm

[1]For information mail to: Perfect Benchmarks, CSRD, 305 Talbot Laboratory, 104 S. Wright Street, Urbana IL 61801

[2]The Conjugate Gradient Algorithm

2.3 Compilers used

The parallelizing compiler we used for our measurements is a modified version of *Kap*, the source-to-source restructurer developed at Kuck & Associates [21]. This compiler was modified as part of the *Cedar* project conducted at CSRD [12]. As a byproduct, an Alliant/FX80 version of the restructurer has been developed. It contains a number of improvements over the original Kap version. For example, we have added *strip mining* capabilities to allow single loops to be vectorized and concurrentized. We also added switches to disable individual restructuring techniques. It should be noted that our Kap version may be different from the one released by Kuck & Associates in 1992.

For comparison we will add results gained by the Vast restructurer, the optimizer built into the Fortran compilation system of the Alliant/FX80 machine. In most programs the two restructurers yielded comparable results, although there are differences in the optimization of individual loops. We will discuss some of the interesting differences.

In this paper *Kap* refers to the restructuring compiler under development at CSRD. *Vast* is the standard optimizing Fortran compiler that is available on the Alliant/FX80 machine.

2.4 Parallelization techniques evaluated

2.4.1 Overall performance of vector and concurrent transformations

We have measured the performance gains that resulted from transforming a serial program into vector, concurrent, and vector-concurrent forms. By *vector* we mean the best transformation the restructurer can do for execution on one processor. By *concurrent* we mean the optimization for multiple processors without exploiting vector functional units. *Vector-concurrent* means using both architectural features, which is the best the restructurer can do.

What answers do we expect from this? It will show us a rough picture of the usefulness of parallelization. The comparison with Vast measurements helps one see the representativeness of our results. It will also be interesting to compare the vector-concurrent profile with the vector profile. Vectorization is a relatively old discipline; concurrent transformations were developed more recently. We will see whether parallelizing compilers can exploit multiple processors as successfully as vector hardware.

The analysis of our measurements is as important as displaying them. We will do this by looking more closely into some of the important code sections and determining why vector and concurrent transformations have made a difference and why they sometimes failed.

2.4.2 Individual restructuring techniques

We will measure the contribution of each restructuring technique by disabling the technique and timing the programs. We will look at the following techniques: *induction variable substitution, scalar expansion, forward substitution, synchronized doacross loops, recurrence recognition, reduction replacement, loop interchanging*, and *strip mining*. We expect to learn what transformations cause or contribute to significant performance gains. Again, we will then investigate the reasons for successes or failures by examining whether and how the transformations were applied in important program sections.

We refer to [27] for an introduction to restructuring techniques. For the purposes of this paper we will define the terms by means of brief examples. All named techniques analyze and transform loops so that they can be executed in parallel.

Induction variable substitution recognizes variables that are incremented in each loop iteration by a constant term. The references to these variables are then substituted by a function of the loop variable:

```
j = j0
DO i=1,n                DOALL i=1,n
  a(j) = ...       -->    a(j0+(i-1)*2) = ...
  j = j + 2             END DOALL
END DO
```

Scalar expansion recognizes variables that are used temporarily within a loop iteration and replaces them by an array so that each iteration has its individual copy of the variable.

```
DO i=1,n                   DOALL i=1,n
  t   =  ...       -->       t1(i) = ...
  ... = t + ...              ...   = t1(i) + ...
END DO                     END DOALL
```

Forward substitution makes known to subsequent statements what values were assigned to a given variable. This can be used to simplify expressions or resolve equations, particularly in data-dependence tests. In the following example "no data dependence" can be recognized, using the fact that m is greater than n.

```
m = n+1                    m = n+1
DO i=1,n                   DOALL i=1,n
  a(i) = a(i+m)    -->       a(i) = a(i+n+1)
END DO                     END DOALL
```

Loop interchanging moves an inner loop of a nest to an outer position (and vice versa) if possible. The reference pattern to arrays can be changed in a beneficial way, such as the innermost loop operating on a contiguous memory segment, or a concurrent loop can be placed in a preferable outer position.

```
DO i=1,n                  DOALL j=2,m
  DOALL j=2,m     -->       DO i=1,n
    a(i,j)=b(i,j)             a(i,j) = b(i,j)
  END DOALL                 END DO
END DO                    END DOALL
```

Synchronized doacross loops are loops that contain data dependences that are run in parallel where the dependent statements are properly synchronized.

```
                           DOACROSS i=1,n
DO i=1,n                     a(i) = b(i)
  a(i)=b(i)        -->       <release current iter>
  c(i)=a(i-1)                <await previous iter>
END DO                       c(i) = a(i-1)
                           END DOACROSS
```

Recurrence recognition There are parallel algorithms that compute the results of loops of the type

```
DO i=1,n
  a(i) = a(i-1)*b + c    -->   CALL rec_solve(a,b,c,n)
END DO
```

These loop patterns are recognized and replaced by a call to the corresponding solver library routine.

Strip-mining In some cases, only one loop can be parallelized. Strip-mining converts a loop into a doubly nested loop, so that it can be executed in vector and concurrent mode.

```
DO i=1,n                    DOALL i=1,n,32
  a(i)=c         -->          j = MIN(i+31,n)
END DO                        a(i:j) = c
                            END DOALL
```

Reduction replacement Loops that accumulate sums into a variable can be vectorized with the sum operator. Many machines provide special instructions for such accumulation operations:

```
DO i=1,n
  s = s + a(i)      -->    s = s + SUM(a(1:n))
END DO
```

Similar operators exist for the calculations of dotproducts, minimums, and maximums.

2.5 Analysis of potential improvements

In order to uncover possible tune-ups for the techniques, we will analyze the situations where the transformations failed and search for more advanced optimizations. We will report about the manually applied transformations that caused measurably better program performance and discuss implementation aspects in a parallelizing compiler.

2.6 Important loop nests

In much of this paper, we will refer to the effects of compilation techniques on important loop nests in a program. This is a practical evaluation measure because currently available compilers operate almost exclusively on loop nests. By “important” we mean the time-consuming parts of the code, which are responsible for good or bad program performance. This is straightforward in most cases: we consider the loop nests that spend at least 90% of the program's execution time. In two cases, SPICE and QCD, significant execution time is spent in loops that the compilers do not recognize. There we call the 10 to 20 most time-consuming loops important. In SPICE many loops are constructed from `IF` and `GOTO` statements. In QCD important code sections look like straight-line code to the compiler because it does not see the enclosing `DO` loop in the calling procedure. We will discuss this further when talking about interprocedural analysis.

2.7 What is not measured?

No studies are complete. A number of important transformation techniques available in the inspected compilers have not been examined in our measurements. Examples are *statement reordering* and *loop distribution*. In addition, there are complementing techniques not commonly called transformations, but which are important parts of restructuring compilers. Examples are last-value assignments and loop-normalizations. We did not evaluate data dependence tests; such measurements are being done in complementary projects [16, 24, 28]. Another important basis of restructuring compilers that we have not covered is the set of analysis techniques, such as the *life-time analysis*. We will discuss some potential improvements through *interprocedural analysis* techniques in Section 5.

There is an additional range of newly proposed restructuring techniques that are not yet available in compilers. Notable techniques are those for *high-level spreading* [15, 19], loop pipelining [1], optimizing loop synchronization [20, 22, 25], tiling [3, 18, 30, 31], recurrence parallelization [2, 5], and runtime data-dependence tests [32, 33]. The quantitative evaluation of this technology will take considerable effort in future projects.

3 Speedups from vectorization and concurrent execution

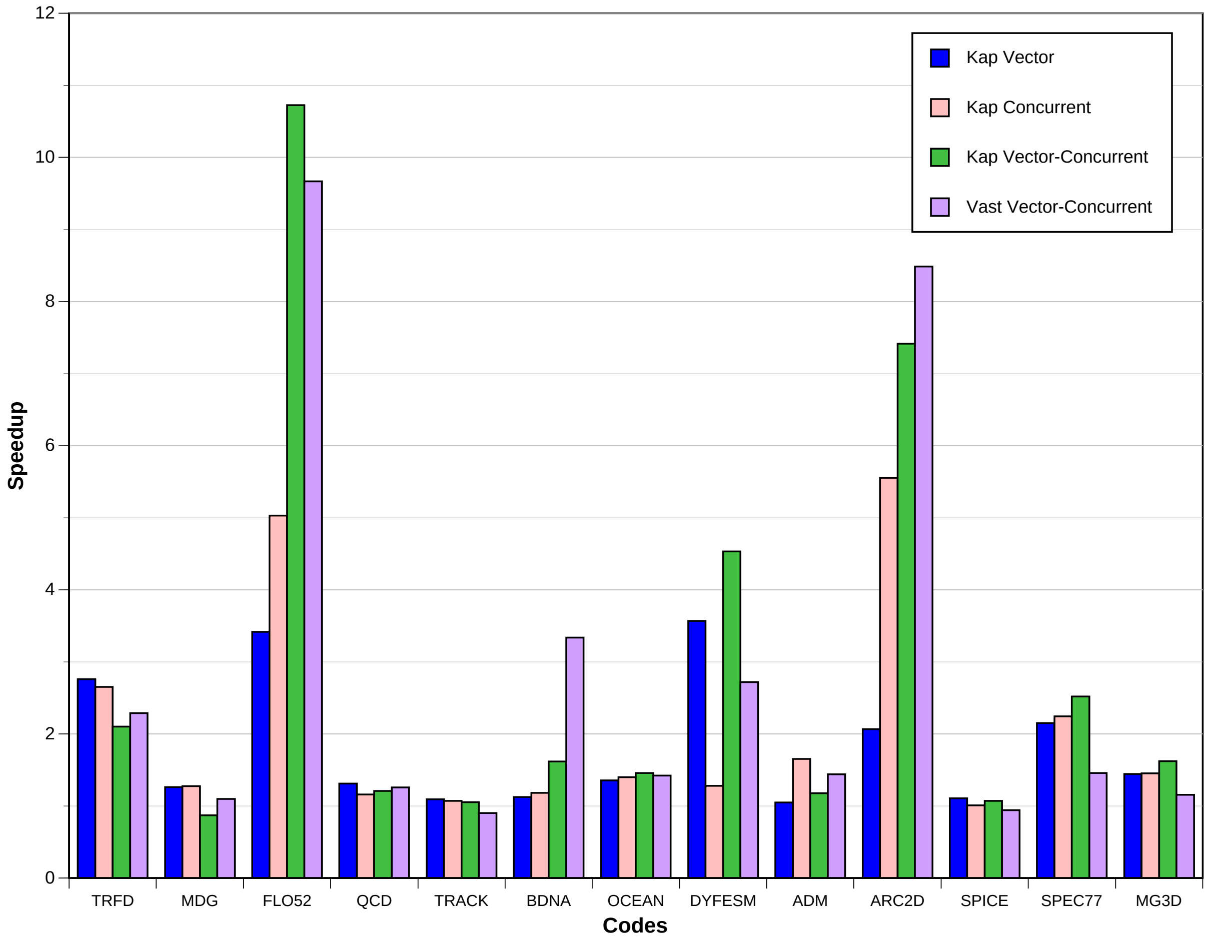

Figure 2: Overall speedups

Our measurements of the speedups from vectorization and concurrent execution are displayed in Figure 2. The figure shows the speedup from serial execution time of the vector, concurrent, and vector-concurrent versions generated by the Kap compiler as well as the vector-concurrent version generated by the Vast compiler. Speedups were calculated from the overall execution time of the different versions. All measurements were initially taken in a single run in multi-user mode using cpu times. For those codes that exhibited instability in their performance, the codes were rerun in single user mode using wall-clock times. In these cases, the single-user time results were used instead. For all codes except for BDNA, we were able to achieve relatively accurate speedups. For BDNA, the speedups were calculated from the minimum execution time of three runs in single-user mode.

Two application codes, namely FLO52 and ARC2D, display very good speedups. FLO52, whose speedup is approximately 11, is close to the best speedups we've seen for application programs, as discussed in Section 2.2. Five other programs exhibit a significant speedup in the range of 2 to 4.5. In six of the thirteen codes the speedup is minimal.

Of the six programs that did poorly, the speedup from vectorization was small. So we can say that when parallelization fails, vectorization fails as well. For the other seven programs, six of the programs exhibit speedups greater than two. Of these six, two programs have speedups close to the ideal number of four. In several cases, the vector or concurrent versions are faster than the vector-concurrent version. This effect is most pronounced in TRFD, MDG, and ADM. This effect is due to the strip-mining of loops with small iterations. The effects of strip-mining will be discussed in Section 4.2.2.

In the remainder of this section we will investigate why the optimizations failed for half of the codes. The discrepancies between the speedups of Vast and Kap will also be examined.

3.1 Why half of the programs don't exhibit speedups

The most noticeable trait of the measurements of overall speedups is that about half of the measured Perfect Benchmarks suite show no significant increase in speed for any of the transformations. This poor performance of current restructuring compilers cannot be attributed to a single cause or flaw; rather, there are a variety of explanations.

3.1.1 Few acceptable candidates for parallelization

Modern-day parallelizing compilers can parallelize only certain kinds of program constructs. Both Kap and Vast can only parallelize `DO` loops without abnormal exits such as `RETURN` statements. They also reject any loops that contain call or I/O statements.

Loops built from `IF/GOTO` Statements One major restriction on parallelizing compilers is that only `DO` loops can be vectorized or transformed into concurrent code. Thus, loops built from `IF` and `GOTO` statements are ignored. SPICE, for example, made extensive use of `IF/GOTO` loops. Only 27% of its execution time was spent inside a `DO` loop. Thus, at most, 27% of SPICE was parallelizable by Kap or Vast.

The statement that current parallelizing compilers cannot parallelize loops built from `IF` and `GOTO` statements is not necessarily true. In fact, Kap is able to transform some `IF/GOTO` loops into `DO` loops and then parallelize them. However, for the restructurer to transform a `IF/GOTO` loop into a `DO` loop, the number of iterations to be made by the loop must be known before the loop is entered. Unfortunately this is not the case for most `IF/GOTO` loops. For example, nearly all of the `IF/GOTO` loops in SPICE execute an unpredictable number of iterations; therefore, this transformation was used only once out of all of the important loop nests.

Loops with abnormal exits Another restriction was that loops must not contain any abnormal exits. That is, any `RETURN` statement or jump from inside the loop to outside would cause the loop to be rejected as a candidate for parallelism. For example, several important loop nests ((getdat/300), (extend/400), and (fptrack/300)) in TRACK contain `RETURN` or jump statements. These statements were used to abort a subroutine if it encountered a major error, such as a data structure overflow. For other codes, abnormal exits were used in loops that perform searches on arrays.

Code	No. of Nests	Call Statements		I/O Statements	
TRFD	4	0		0	
MDG	3	2	(66.7%)	0	
FLO52	19	0		1	(5.2%)
QCD	10	1	(10.0%)	0	
TRACK	23	7	(30.4%)	1	(4.3%)
BDNA	4	0		1	(25.0%)
OCEAN	11	1	(9.1%)	0	
DYFESM	12	2	(16.7%)	0	
ADM	31	11	(35.5%)	0	
ARC2D	28	2	(7.1%)	0	
SPICE	14	2	(14.3%)	1	(7.1%)
Total	159	28	(17.3%)	4	(3.8%)

Table 1: Percentage of important loop nests containing call or I/O statements

Call and I/O statements Call and I/O statements also prevent loops from being parallelized. Table 1 displays the percentage of important loop nests that contain a call or I/O statement. The table shows that MDG, TRACK, and ADM all have call statements in a large amount of their important loop nests. Additionally, the table shows that it is common to have a call statement in at least one important loop nest. It also shows that I/O statements are less common.

Table 1 does not yet quantify the execution time spent inside loops with call statements. We have seen that this time is significant for the programs MDG, QCD, TRACK, and ADM. For example, at least 40% of QCD's execution time is spent inside such loops. The four codes resulted in poor performance primarily because of call statements. The compiler attempts to parallelize only loops inside the called routine or those adjacent to the call statement in the outer loop. We have observed that these loops often have a small number of iterations (1–16) and small loop bodies (1–4 statements). Small numbers of iterations limit parallelism, while overheads in concurrent execution greatly offset speedups for small loop bodies. For example, almost all of the loops in TRACK were 4–16 iterations, and many had one-statement bodies. For these reasons, even manual parallelization of loops without call statements could not improve ADM's speedup beyond two.

By parallelizing loops with call statements, we can achieve good overall speedups in some codes. In MDG, QCD, TRACK, and ADM, at least 90% of the execution time is spent in loop nests with call statements or their called subroutines. For ADM, manually parallelizing these loop nests achieved a speedup of 6.6 from serial execution time. Impressive speedups of the other three codes were also achieved either by parallelizing loops with call statements or using inline expansion[14, 17]. Other techniques that were used to achieve these improvements are described in section 5.

The effect of I/O statements on program performance is less pronounced. In BDNA, the important loop nest containing an I/O statement accounted for only 7.8% of the vector-concurrent execution time. For FLO52, TRACK, and SPICE, the important loop nests containing I/O statements accounted for 1.4%, 4.2%, and 0.5% of their vector-concurrent execution time respectively. Thus, from Table 1, we can say that call statements play an important role in parallelizing programs while I/O statements play a lesser role.

3.1.2 Important loops were not parallelized

As described in 3.1.1, Kap and Vast will not attempt to parallelize loops that do not meet certain criteria, such as being `DO` loops or not having call or I/O statements. However, even among the loops that the compilers accepted for parallelization, there are significant numbers of loops that were not parallelized. We found three reasons why Kap did not parallelize such loops.

Unresolvable data dependences The most common reason why a loop nest is not parallelized is that it contains data dependences that cannot be broken. In some cases, data dependences prevent entire loop nests from being parallelized. In the code OCEAN, 8 of the 11 most important loop nests are completely unparallelizable because of data dependences. About 65% of the code's execution time is spent in these nests, which is the dominant reason for OCEAN's poor performance. In other programs, data dependences prevent only some of the loops in loop nests from being parallelized. Usually, it is the outer loops that cannot be parallelized. A good example of this is TRFD, where only the innermost loops of all four of its important loop nests were parallelized.

Lets examine some of the sources of data dependences. OCEAN is unparallelizable mainly because of complicated subscripts. Subscripts of the form: $ia + jb + c$, where i and j are loop index variables, a and b are subroutine parameters, and c is an unknown constant, appear in nearly every important loop nest. For the other codes, a common culprit of data dependences are accumulations; that is, statements of the form: $sum = sum + \cdots$. In some cases, the variable sum is a scalar; in other cases, it is an array element. Accumulations appeared in nearly every code. It was an accumulation into an array element that prevented all but the innermost loops of the important loop nests of TRFD from being parallelized. By transforming the accumulations into a parallelizable form, the middle loops of these nests can be parallelized. Performing this transformation manually, the vector-concurrent speedup of TRFD improved from 2.1 to 8.5. Another common source of data dependences are the use of arrays as temporary variables inside loop nests. For example, MDG uses arrays as temporaries inside two of its important loop nests (interf/1000 and poteng/2000), which account for 69% of the code's execution time. Section 5 will discuss techniques for parallelizing some of these forms of data dependences.

Too conservative with synchronizations One way of parallelizing loops with data dependences is to use synchronizations. Yet synchronizations were used in only five important loop nests in all the programs measured. Synchronizations are used so rarely not because they cannot

Code	No. of Nests	12	25	50	70	100
TRFD	4	0	0	0	0	0
MDG	3	0	0	1 (33.3%)	1 (33.3%)	1 (33.0%)
FLO52	19	0	1 (5.3%)	1 (5.3%)	1 (5.3%)	1 (5.3%)
TRACK	23	0	0	4 (17.4%)	4 (17.4%)	4 (17.4%)
BDNA	4	1 (25.0%)	1 (25.0%)	1 (25.0%)	2 (50.0%)	—

Table 2: Percentage of important loop nests using synchronizations for varying concurrency threshold

be applied but because the compiler is conservative in their use. Because synchronizations add some overhead to the loop's execution time, the parallelization of a loop with a large sequential section may be slower than the serial version of the loop. To prevent this, Kap examines all candidates for synchronizations and rejects all loops in which the serial portion seems too large. A Kap variable, known as the concurrency threshold, determines the cutoff point between serial and concurrent. Roughly, the concurrency threshold holds the maximum percentage of the loop that can be serial. By default, the concurrency threshold was set to the low value of twelve. Table 2 displays the number of important loop nests that used synchronization for varying concurrency thresholds for a subset of the Perfect Benchmarks. As the table shows, increasing the concurrency threshold increases the percentage of loops nests using synchronizations. In the two codes, FLO52 and BDNA, the loop nests with synchronizations run significantly faster than the serial version of the nests. In FLO52 setting the concurrency threshold to 25 or above allowed a loop nest (euler/70) to be parallelized. The loop nest exhibited a speedup of 2.5, improving the speed of the vector-concurrent program by 10%. In BDNA, the higher concurrency threshold allows an important loop nest (actfor/500), which accounts for 62% of the vector-concurrent execution time of the code, to be parallelized. It should be noted, however, that a concurrency threshold that is too high can be as detrimental as a threshold that is too low. For example,TRACK ran slightly more slowly with a concurrency threshold greater than or equal to 50. ARC2D also suffers from a performance loss when the concurrency threshold is increased.

Compiler data structure overflow Limitations on the size of a restructurer's internal data structures may also prevent parallelization of loops. A loop that is too big or that has too many data dependences would cause the restructurer's data structures to overflow. When such overflows occur, the restructurer usually gives up on the loop.

An example of a important loop that is not parallelized because of data structure overflow occurs in BDNA. This loop (actfor/237) is more than 300 lines long. When Kap examines this loop for dependences, it detects about 7000 of them, which exceeds its capacity. By increasing the limits of Kap's data structures, Kap was able to vectorize this loop.

3.1.3 Poorly chosen candidates for parallelism

Another reason why the codes show little speedup is that the restructurer parallelized the wrong loops; that is, it parallelized loops that gave small speedups. However, if it parallelized some other loop, the speedup would have been much greater. This effect occurred when Kap chose a loop with too few iterations or when Kap ignored the outer loops, due to a restricted search space for the best combination of parallelized loops.

Parallelized loop with too few iterations In several cases Kap chose to parallelize a loop with too few iterations. In DYFESM, Kap chose to transform the outermost loop of a triply nested loop (matmul/400) into a concurrent loop. Unfortunately, the outer loop has only one iteration. Thus the benefits of executing this loop nest concurrently were completely negated. There were several other cases where Kap chose to transform a loop into concurrent form where that loop had too few iterations to keep all eight processors busy. The same effect can occur with vectorization. For example, Kap vectorized a one-iteration loop in a triply-nested loop (hyd/30) in ADM. Since vectorization incurs some overhead, the vectorized version ran twice as slowly as the serial version.

Considering the effects of parallelizing loops with few iterations, it seems wise to include loop bounds checking in a restructurer. In fact, Kap does check the bounds of loops that it tries to vectorize. Unfortunately, variables are often used in loop bounds. For both examples given in the previous paragraph, the upper bound was a variable. Advanced techniques are needed to perform bounds checking on loops with unknown bounds. We will discuss these techniques in section 5.

Outermost loops not parallelized Another problem with Kap is that it doesn't always parallelize the outermost loop in a nest, even though that loop is parallelizable. Parallelizing the outermost loop is usually desirable because it cuts down on the costs of startup as well as the cost of the barrier synchronization at the end of the loop. Because of this, Kap tries its very best to parallelize the outermost loop, using loop interchanging if necessary. However, to prevent Kap from taking too much time in parallelizing programs, limits are placed in the search space for an optimal combination of parallelized loops. If Kap reaches its limits in its search, it gives up and instead parallelizes the inner loops. In the code DYFESM, the third from outermost loop in an important nest (mnlbyx/50) was chosen for concurrent mode even though the outer two loops of that loop nest are parallelizable. In FLO52, it is common for important loop nests to consist of one outer loop surrounding 3 or 4 singly or doubly nested loops. In several cases, (that is, in loop nests (dflux/30/60), (dfluxc/20), and (addx/20)) the outermost loop was not parallelized even though it is parallelizable. By turning off loop interchanging, the outer loops of all these cases were parallelized. This occurs because Kap does not need to try all the ways to perform of loop interchanging, cutting down on the size of the search space.

3.2 Why do Kap and Vast perform differently in some programs?

For the most part the performances of Vast and Kap are comparable. However, there are two codes (BDNA and DYFESM) with which one restructurer generates code that is about twice as fast as the other. We will determine the reasons for these differences.

In BDNA, Vast's vector-concurrent version is twice as fast as Kap's because Vast is more successful at parallelizing two of the four important loop nests of the code. In BDNA's most important loop nest (actfor/500), Kap is only able to vectorize the inner loop of the nest; this is because the nest contains reduction statements of the form $sum = sum + a(i)$. Kap can vectorize loops with such reduction statements but is unable to transform them into concurrent loops. Vast, on the other hand, is able to translate loops with reductions into concurrent or vector-concurrent form. Since this loop nest takes up about 60% of the vector-concurrent execution time, the transformation of this loop into vector-concurrent form had a profound effect on the speed of BDNA. Vast also did better than Kap in another important loop nest (actfor/237). Kap was not able to parallelize this loop because of an overflow of internal data structures, as described in Section 3.1.2. Vast did not run into this difficulty.

The Kap-optimized version of DYFESM runs significantly faster than the one optimized by Vast. The fact that Kap does not generate code that performs reductions concurrently is of significant advantage in this code. The iteration numbers in DYFESM are generally very low, so that the overhead of executing the two loops (chosol/53/80) vector-concurrently is high. In another nest (mnlbyx/50), Vast strip-mines the innermost loop while Kap chooses an outer loop in the nest for concurrent execution. The major difference comes from (matmul/400), the most time-consuming loop nest, where Vast generates a synchronized loop which has more overhead than benefit. Kap chooses an outer loop in the nest for concurrent execution. Although this loop has only one iteration, as we described in Section 3.1.3, the generated code outperforms the Vast variant.

4 Contributions of individual restructuring techniques

Our measurements of the individual restructuring techniques are displayed in Figure 3. The measurements for the code MG3D could not be obtained because of compilation difficulties. The two missing results in the figure (no scalar expansion for ADM and recurrences for SPEC77) were also left out because of compilation difficulties. The speedups for the vector-concurrent versions are the same as in Figure 2. The additional bars represent the vector-concurrent code with a particular transformation turned off. The only exception is recurrence replacement, in which the bar displays the execution time with the transformation on. For all other versions, recurrence replacement is off. The transformations examined are *recurrence replacement, synchronized doacross loops, induction variable substitution, scalar expansion, forward substitution, reduction statements, loop interchanging,* and *strip-mining.*

Surprisingly enough, the restructuring techniques have little effect on the speed of the codes. There are a few exceptions, however. The most noticeable exception is *scalar expansion*, which has a great effect on the speeds of the programs FLO52, BDNA, DYFESM, and ARC2D. Another interesting transformation is *strip-mining.* In TRFD, MDG, ADM, and SPEC77, strip-mining causes a loss in program speed while in FLO52 it shows an improvement. *Reduction replacement* also has a significant effect on the parallelization of BDNA, DYFESM, and SPEC77. *Induction variable substitution* and *synchronized loops* give a small performance gain for FLO52 while substituting recurrence loops has a negative performance impact on the same code. This section will examine why the techniques did so poorly, as well as why the exceptions did well.

A general characteristic of Figure 3 is that restructuring techniques have a greater effect in codes that exhibit a good speedup. This is not surprising, because when there is no speedup there are no techniques to credit. This characteristic makes one ask, "How well could the techniques do if the other shortcomings of the compilation process could be overcome?" Our experience indicates that available techniques *will* become more important in combination with new transformations.

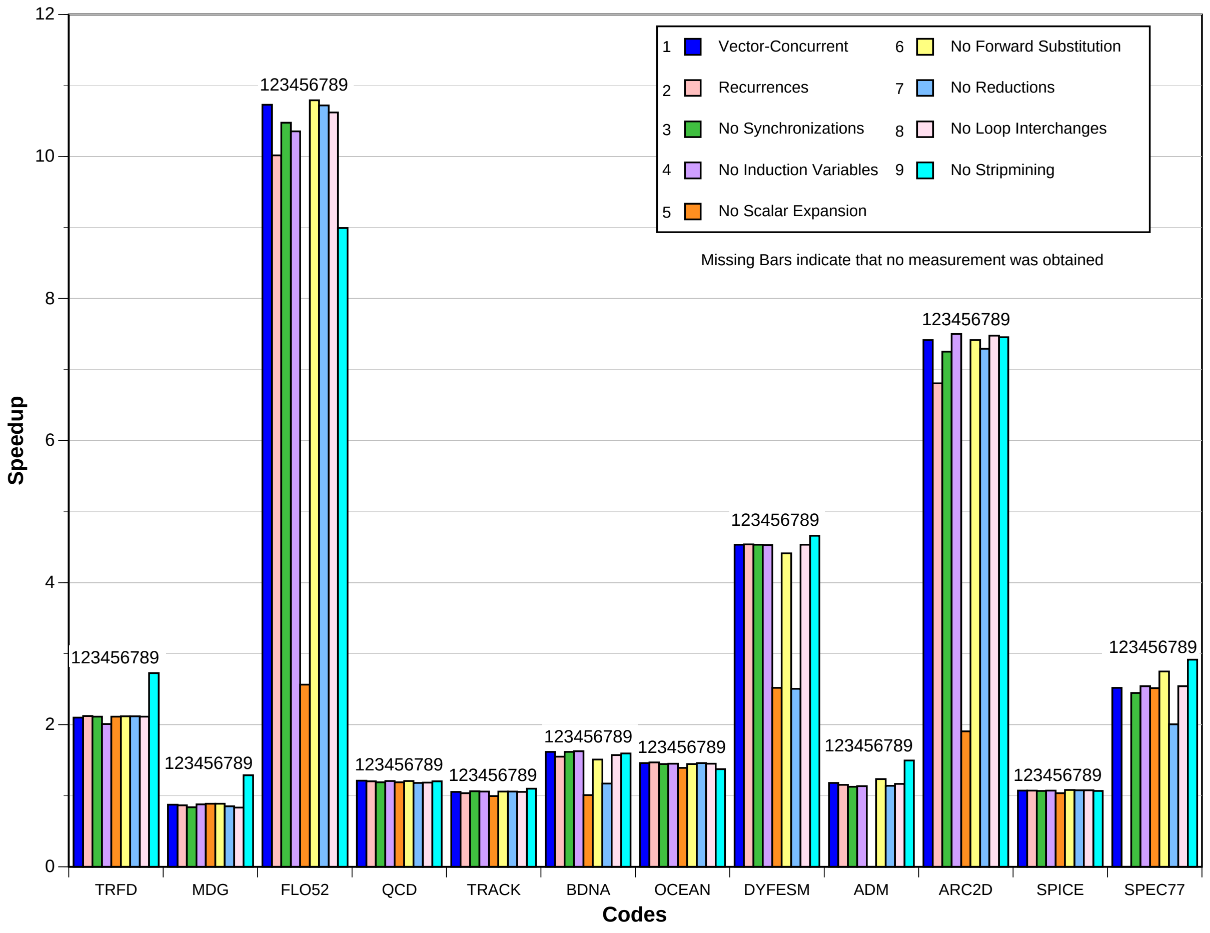

Figure 3: Speedups after disabling individual transformations

4.1 The transformations that make a difference: scalar expansion and reduction replacement

Of all the restructuring techniques examined, scalar expansion had the greatest effect on performance of the codes. Intuitively, this is not too surprising, for it is natural for a programmer to use temporary variables inside a loop. If left alone, these temporaries cause data dependences between iterations, preventing parallelization. Scalar expansion of these temporaries break the data dependences, allowing the loop to be transformed into a vector or concurrent form. Thus, without scalar expansion, many loops would not be parallelized.

This assertion is supported by the measured results. In FLO52, 30–40% of its vector-concurrent execution time is spent in loop nests that used scalar expansion. With scalar expansion, these loop nests exhibited a speedup of about 11.5 from serial execution time. If scalar expansion is turned off, all of these loop nests except one are completely serial. Because of this, turning off scalar expansion causes an overall slowdown of 4, where these loop nests now take 80–88% of execution time. For BDNA, this effect is even worse. Every important loop nest that was parallelized used scalar expansion. When scalar expansion is turned off, the program shows no speedup from serial execution time. The effects of scalar expansion on DYFESM and ARC2D is similar.

Reduction replacement is another transformation that significantly improves the performance of several codes. Five of twelve loop nests in DYFESM, which account for at least 25% of its execution time, use reduction replacement. Turning off reduction replacement essentially serializes three of these five loop nests. This causes the execution time of DYFESM to nearly double, where these five loop nests now account for at least 53% of the execution time. Turning off reduction replacment in BDNA causes its most important loop nest (actfor/500), which takes 62% of its execution time, to become completely serial. The code SPEC77 is affected similarly by reduction replacement.

4.2 Analysis of the generally low effectiveness

In the remainder of this section we discuss several reasons why some individual restructuring techniques do not pay off in any of the programs.

4.2.1 The techniques are not used in important loops

The simplest answer to why restructuring techniques are not effective is that they are not being used. Table 3 gives the percentage of loop nests that use a given technique for a given program. A blank cell in the table indicates that the technique was not used in important loop nests. A technique is said to be used in an important loop nest if the application of the technique changes how the loops inside the nest are parallelized. The last two techniques, strip-mining and loop interchanging, are separated from the other techniques because these techniques improve upon the machine's use of parallelism rather than enabling loops to be parallelized. Note that this table indicates only how many important loop nests use a certain technique. It does not indicate the importance of the loop nests that use the technique.

Examining Table 3, we can see that the transformations: *recurrence replacement, synchronizations, induction variable recognition,* and *forward substitution* were all used infrequently. We have found that a restructuring technique may not be used for three reasons. First, the

Code	No. of Nests	Rec	Sync	Indvar	Scalar	ForSubst	Red	Loopint	Strip
TRFD	4			50.0					100.0
MDG	3						66.7		100.0
FLO52	19	10.5		5.3	52.6		5.3	31.6	42.1
QCD	10						10.0	20.0	60.0
TRACK	23				8.7	8.7	4.3		39.1
BDNA	4		25.0		75.0		25.0		25.0
OCEAN	11				9.1				27.3
DYFESM	12				25.0		41.7	8.3	41.7
ADM	31	3.2		3.2	58.1	3.2	19.4	6.5	54.8
ARC2D	28	14.3			67.9		7.1	28.6	28.6
SPICE	14								21.4
SPEC77	30	3.3	13.3	33.3	20.0	20.0	13.3		36.7
Total	189	2.6	3.2	7.7	26.4	2.7	16.1	7.9	48.1

Key:

Rec	Recurrence replacement
Sync	Synchronizations
IndVar	Induction variable recognition
Scalar	Scalar expansion
ForSubst	Forward substitution
Red	Reduction replacement
LoopInt	Loop interchanging
Strip	Strip-Mining

Table 3: Percentage of important loop nests that use the given restructuring technique

technique cannot be applied in important loops. Second, the technique can be applied but the compiler is unable to do so. Third, the technique can be applied but the compiler chooses not to by some heuristics.

Code	No. of Nests	Using induction vars. No.	%
TRFD	4	4	(100.0)
MDG	3	2	(66.7)
FLO52	19	1	(5.3)
QCD	10	0	
TRACK	23	1	(4.3)
BDNA	4	0	
OCEAN	11	6	(54.5)
DYFESM	12	0	
ADM	31	4	(12.9)
ARC2D	28	0	
SPICE	14	2	(14.2)
SPEC77	30	14	(46.7)
Total	189	34	(25.4)

Table 4: Number and percentage of important loop nests that use induction variables

Few opportunities for use of technique The most common reason that restructuring techniques are not used is that they are not applicable. The three techniques that have few opportunities for use in important loop nests are induction variable recognition, recurrence replacement, and; to a lesser extent, loop interchanging. Table 4 displays the number and percentage of important loop nests which use induction variables at some point. Note that about two-thirds of the programs use induction variables rarely or not at all. While this shows that this technique is not applied frequently in available compilers, there is an indication that it may become more important in the future: induction variable recognition was critical in the manual parallelization of TRFD and MDG[14].

We pick *loop interchanging* to illustrate the applicability of a technique. Table 5 displays the percentage of loop nests that are parallelizable and at least doubly nested. This shows how many loops can potentially be interchanged; 47% of the loop nests can use the technique. However, the technique is applied much less frequently. We have observed that many loop nests already meet Kap's goal for loop interchanging: to achieve a stride-one access for vector statements, and to move a concurrent loop outward for reducing overhead. In Section 3.1.3 we have also mentioned that loop interchanging may not occur because Kap hits the limits of its search space for an optimal combination of parallelized loops and gives up.

Compiler cannot apply technique In some cases the application of one technique may prevent the application of another technique. We have found this to be mostly due to compiler

Code	No. Nests	possible interchange No.	%
TRFD	4	4	(100.0)
MDG	3	2	(66.7)
FLO52	19	15	(78.9)
QCD	10	2	(20.0)
TRACK	23	0	
BDNA	4	2	(50.0)
OCEAN	11	1	(9.1)
DYFESM	12	7	(58.3)
ADM	31	5	(16.1)
ARC2D	28	28	(100.0)
SPICE	14	0	
SPEC77	30	20	(66.7)
Total	189	86	(47.2)

Table 5: Number and percentage of important loop nests in which loop interchanging can be used

internal conflicts. One good example is the interference between loop interchanging and strip-mining in four important loop nests ((ypenta/1), (ypent2/1), (xpenta/3), and (xpent2/3)) in ARC2D. Kap performs loop interchanging and strip-mining in separate passes, with loop interchanging coming first. In these four loop nests, Kap moved the parallelizable inner loop outwards, which prevented the loop from being strip-mined in the later pass. By turning off loop interchanging, these inner loops were strip-mined.

In other cases, the application of a technique may complicate the code of a loop to the point that it could not be parallelized. An example of this occurs in a loop nest (assemr/40) in DYFESM. The innermost loops of this nest contain a statement of the form: $a(c+i) = a(c+i) + \cdots$ where c is a variable set before the start of these loops to a complicated expression involving an array subscript. With the application of forward substitution and scalar expansion, the subscripts of a become too complicated for the compiler to handle, causing it to give up on the loop. If these techniques are turned off, Kap can see that c is constant in respect to the loops and is able to parallelize them.

Compiler can use but rejects a technique The third reason why a technique is not used is that the compiler chooses not to apply the technique, even though it is applicable; that is, a set of heuristics inside the compiler decides whether the technique is applied or not. Parallelization with synchronization statements is a technique that is guided by heuristics. The conservative use of synchronizations was described in Section 3.1.2. Bad choices in the decision of which loop to parallelize can also be blamed on poor heuristics. Section 3.1.3 described the effects of these bad choices in detail.

4.2.2 The technique causes slow-downs that offset or dominate any speedups caused by the technique

The application of a restructuring technique does not always improve the performance in the program. In some cases, the technique may cause a slowdown, hiding or even dominating any positive effects it may have. We will examine three restructuring techniques that are prone to this behavior; strip-mining, loop interchanging, and recurrence replacement. Synchronizations can also cause slowdowns but will not be examined because the technique is used so infrequently.

Code	Vec-conc Exec Time (sec)	Sec. Slower	Sec. Faster
TRFD	291.6	66.9	0.0
MDG	598.1	54.8	0.0
TRACK	106.5	2.1	0.1
BDNA	274.0	2.9	0.0
OCEAN	1562.9	6.1	74.9
DYFESM	132.8	2.4	2.1
ADM	639.5	105.3	0.0
ARC2D	250.6	1.0	5.9
SPICE	113.9	0.0	0.5
SPEC77	679.3	63.2	0.0

Table 6: Effects of strip-mining on important loop nests

Strip-Mining Table 6, which displays the effect of strip-mining on ten of the twelve codes examined, was derived from loop-by-loop profiles. More specifically, loop nests that exhibit a speedup and loop nests that exhibit a slowdown are separated to make any trade-offs explicit. *Vec-conc Exec Time* shows the time taken by the vector-concurrent version with strip-mining of the codes. *Sec Slower* displays in seconds the total of all slowdowns caused by strip-mining. Similarly, *Sec Faster* displays the total of all speedups from strip-mining.

Examining Table 6, one can see that the effects of strip-mining vary widely from code to code. For TRFD, MDG, ADM, and SPEC77, strip-mining nearly always has a detrimental effect. However, for OCEAN and ARC2D strip-mining is beneficial. Another code that benefits from strip-mining is FLO52.

The reason why strip-mining caused slowdowns is that the strip-mined loops of the codes had too few iterations to support the transformation. Kap splits a loop into chunks of a fixed size, 32 by default, giving a chunk to each processor. The major drawback of this form of strip-mining is that for low iteration counts not all processors are used. In TRFD, all of the strip-mined loops have 40 iterations or less. Because all of TRFD's parallelized important loops are these strip-mined loops, the program never uses more than two of the eight processors in its execution. In MDG, all the parallelized loops in the important loop nests were strip-mined. These loops ranged from 5 to 14 iterations. Every strip-mined loop in TRACK had from 4 to 16 iterations. In ADM, the six most important parallelizable loop nests that were strip-mined had 1 to 16 iterations. For all three of these programs, only one processor was used.

Examining Figure 2, one can see that for the four programs that are hit hardest by strip-mining, (TRFD, MDG, BDNA, and ADM) the Vast version performs better than the vector-concurrent version but less well than the vector or concurrent versions. This is because Vast splits the vector equally among all eight processors. Unfortunately, for small iteration counts, this spreading across eight processors generates very small vector lengths. On the FX/80, the break-even point for vector lengths is 4; that is, the vector must be at least 4 elements long for it to be as fast or faster than serial code. Thus, for small iteration counts, the slowdowns caused by small vector lengths overwhelm the speedups due to concurrent execution, which is why Vast's vector-concurrent version cannot outperform the vector or the concurrent times for Kap.

Code	Vec-Conc Exec Time (sec)	Sec. Slower	Sec. Faster
FLO52	87.3	4.8	12.8
DYFESM	132.8	0.0	2.6
ADM	639.5	0.0	6.8
ARC2D	250.6	0.5	14.3

Table 7: Effects of loop interchanging on important loop nests

Loop interchanging Another restructuring technique that may cause slowdowns is loop interchanging. The two codes that use loop interchanging most often are FLO52 and ARC2D. Table 7 shows the effects of these two codes as well as two others. Note that loop interchanging improves the program's speed for the most part. To a great extent speedups or slowdowns from loop interchanging are due to side effects of the technique. For 33% of the interchanged loops in FLO52 and 62% of the interchanged loops in ARC2D, loop interchanging prevented strip-mining, as was described in Section 4.2.1. Depending on the number of iterations in the loop, this side effect sometimes improved execution time and sometimes worsened it. In the other 67% of FLO52's interchanged loops, turning off loop interchanging allowed the outermost loop of the nest to be transformed into concurrent form. This effect was described in Section 3.1.3. Again the impact of this side effect on execution time was mixed, owing to iteration counts.

Code	Vec-Conc Exec Time (sec)	Sec. Slower	Sec. Faster
FLO52	87.3	4.5	0.0
ADM	639.5	9.9	0.0
ARC2D	250.6	16.9	0.0

Table 8: Effects of recurrence replacement on important loop nests

Recurrence replacement Recurrence replacement may also incur slowdowns. Table 8 displays the effect of recurrence replacement on three of the four codes that use this technique in important loop nests. In all three cases, recurrence replacement causes a slowdown. There are two reasons for this negative impact. First, the length of the recurrence is too small. In FLO52 the recurrences are 10-100 elements long, while in ADM they are about 15 elements long. The second reason is that the recurrences must be conformed to match the format of the parameters of the subroutine calls to the recurrence solvers. This reshaping is done by an array-to-array copy operation. Similarly, the results of the solver must be moved into the destination array. For small recurrences, these copy operations can offset any speedups.

4.2.3 The transformation does not give the restructurer any new opportunities for parallelism

Code	No. of Nests	ForSubst Used		Loops Parallelized	
		No.	%	No.	%
TRFD	4	4	(100.0)	0	
MDG	3	0		0	
FLO52	19	0		0	
QCD	10	0		0	
TRACK	23	2	(8.7)	2	(8.7)
BDNA	4	2	(50.0)	0	
OCEAN	11	0		0	
DYFESM	12	3	(25.0)	0	
ADM	31	10	(32.3)	1	(3.2)
ARC2D	28	9	(32.1)	0	
SPICE	14	2	(14.3)	0	
SPEC77	30	8	(26.7)	6	(20.0)
Total	189	32	(24.1)	3	(2.7)

Table 9: Number and percentage of use and effectiveness of forward substitution in important loop nests

In some cases the application of a restructuring technique may not affect the ability to parallelize the loop. Most techniques exhibit this effect to some extent, but it occurs regularly for forward substitution. Table 3 displays only those loop nests in which forward substitution assisted in the parallelization. Forward substitution was used much more frequently than this table suggests, but it had little effect on loop nests. Table 9 displays the percentage of loop nests that used forward substitution. This table shows that a frequently used technique is not necessarily an effective technique for assisting parallelization.

5 Where is potential for improvements?

In the previous sections, we have examined the effectiveness of parallelizing compilers, the restructuring techniques they use, and the underlying causes of good and poor results. Considering the limited effectiveness of the parallelizing compilers examined in this paper, one would ask, “Is this the best a parallelizing compiler can do?” Although it is impossible for one to predict accurately how much better future parallelizing compilers will do, we do believe that they can be improved. There is potential of improvement for the implementation issues we have uncovered, such as compiler data structure overflow, interference between techniques, and outer loops not being parallelized due to time-outs in the compiler's search for the optimal parallelization. Solutions to these problems seem feasible, possibly at the cost of increased compilation time. Further evidence for potential improvement was delivered by the compiler group at CSRD which has been working on manually parallelizing the Perfect Benchmarks programs, applying mostly automatable techniques [11, 14]. The attained speedups ranged from 4 to 17, using the same measures as in Figure 2.

5.1 A case study: Parallelization of the code ADM

We use the ADM program to illustrate these findings.

ADM simulates air pollution concentration and deposition patterns for lake shore environments [29]. It is 6104 lines long and consists of 97 subroutines. The execution time is spread evenly throughout the program; 90% of the execution time is spent in 23 subroutines. Almost all of these subroutines contain 1–3 loop nests, all of which are important. Because of this, 90% of the program's execution time is spread across 31 loop nests. Thus, a large number of loops need to be parallelized to get any meaningful speedups from the program. However, 11 of these 31 important loop nests have calls in them. Almost all of these calls are made to subroutines containing important loop nests.

Kap was unable to parallelize ADM effectively for several reasons. First, Kap is unable to parallelize loops containing call statements, which excluded a good deal of the program from parallelization. Second, the loops Kap could parallelize had small iteration counts and were often singly nested. For the six most important loop nests, iteration counts ranged from 1 to 16 and four of them were singly nested. For other parallelizable loop nests, the iteration counts were 13–15 or 64. Most of these loops were singly nested. Since Kap strip-mines singly nested loops, the vector-concurrent version of Kap performed worse than either vector or concurrent versions. For these two reasons, the best speedup that Kap was able to get was 1.65. Even by manually parallelizing the loops without call statements, we were not able to get speedups much greater than two.

Our search for new compilation technology has led to a much more aggressive parallelization of ADM, with a resulting speedup of 6.6. All of the top-level, important loop nests in this code were transformed into concurrent loops. We have found these transformations to be representative for those described in [14]. The following paragraphs illustrate the most important analysis and restructuring techniques applied to ADM.

5.2 Advancing restructuring techniques

The performance gain in our experiments with ADM were the result of two techniques: *array privatization* and *parallel reductions*.

Array privatization can be considered an extension of the *scalar expansion* technique. It recognizes arrays that are used temporarily in a loop and replaces them with loop-local arrays or adds an additional dimension, respectively. The following example shows a loop nest using a temporary array in three forms: serial, concurrent using array expansion, and concurrent using array privatization.

```
DIMENSION t(m)            DIMENSION t(n,m)              DOALL i=1,n
DO i=1,n                  DOALL i=1,n                     DIMENSION t1(m)
  DO j=1,m                  DO j=1,m                      DO j=1,m
    t(j) =  ...               t1(i,j) = ...                 t1(j) = ...
  END DO                    END DO                        END DO
  ...                       ...                           ...
  DO j=1,m                  DO j=1,m                      DO j=1,m
    ... = t(j) + ...          ... = t1(i,j) + ...           ... = t1(j) + ...
  END DO                    END DO                        END DO
END DO                    END DOALL                     END DOALL
```

In ADM we have seen many occurrences of array data structures that are used temporarily within loop iterations. Left alone, these patterns represent data dependences that cannot be broken by available compilers. Privatizing these arrays gives each iteration its own copy of the data structure, which resolves the storage conflicts.

Another common source of data dependences in loops are statements of form $s = s + expr$, where a variable s accumulates expressions computed locally in each iteration. Kap can vectorize loops containing these statements by the use of *reduction replacement*. However, it is unable to execute them concurrently, or able to deal with loops containing multiple statements that accumulate into a variable, and with accumulations into array elements. There are several ways to implement concurrent reductions. For example, one can accumulate the expressions partially on each processor and sum up the partial results after the loop.

While these two techniques illustrate two important areas that deserve more future attention, there is a range of techniques whose automation can cause significant program performance improvements, as described in [11, 14]. Given the number of newly proposed techniques, an interesting question is, which of these techniques would cause notable performance gains. We are not in the position to give a final answer to this, as we pointed out in Section 2.7. However, in our experience so far, the techniques referenced in Section 2.7 contributed only little to the resulting performance. Further empirical performance studies are necessary to give more quantitative answers.

5.3 Advancing Analysis Methods

In order to find privatizable arrays in our experiments, we must analyze definitions and uses of arrays. An array (range) is privatizable to a given loop if it is defined before it is used in the loop body. We have found a number of issues that need be addressed to do a successful def/use analysis:

First, the notion of array ranges is important. It is not uncommon for ordinary Fortran programs to declare a large array in the main program and pass sections of the array to its subroutines. In ADM we have found arrays that consisted of both read-only and read-write sections in some loops. We have parallelized these loops by splitting off and privatizing the read-write section.

Second, array ranges can be described by variables whose values are not derivable from the local loop environment. We must use *Propagation techniques* of values, relations between variables, and sometimes even conditions under which relations hold. In some cases this requires advanced symbolic analysis capabilities.

Third, all these methods must be applied interprocedurally. For a successful parallelization usually outer loops of a program must be parallelized. Such loops are likely to contain subroutines, some of which define and others that make use of the array data. Variables that define array ranges are often defined in an initialization routine. The compiler used in our experiments is capable of doing *subroutine inline expansion* to address these needs. Unfortunately in many of our experiments this compiler feature hit some practical limits, so that it is not possible to present a consistent picture of its usefulness.

The analysis of variable values is of further importance for determining loop bounds. We have mentioned that strip-mining loops with few iterations caused performance degradations in some Perfect codes. In ADM eight of the important, top-level nests have two perfectly nested loops at their outermost positions. Either of these loops can be parallelized. However, for the input data included in the Perfect Benchmarks, one of these loops is always one iteration long. Similarly, there exist vectorizable loops with only one iteration. If the compiler chooses the wrong loop to parallelize, poor speedups or even slowdowns will be seen for the loop nest.

Some of the variable values can be derived interprocedurally. In other cases they are input-data dependent. A possible solution to this problem is to generate multiple version loops; that is, multiple copies of the same loop or loop nest are made, where each copy is transformed differently. An `IF` statement selects the proper version, based on the iteration count. The main problem with multiple version loops is a possibly huge growth in program size because there may be many optimization variants of a given loop nest. Alternatively, profile or input data can be used by the compiler. However, the power of this technique is also its weakness: the program performance may be very sensitive to input-data.

The transformation of reduction operations into parallel form needs advanced analysis techniques that detect multi-statement reductions. Current compilers are able to derive single-statement reductions from the dependence graph and statement pattern. The analysis of multiple reduction operations onto the same variable within a loop is an extension to this.

6 Conclusions

We have studied the effectiveness of parallelizing compilers and the restructuring techniques they use by measuring the speedups of transformed programs on the Alliant/FX80 computer, using the Perfect Benchmarks. Individual restructuring techniques were evaluated by turning them off and measuring the performance loss. A modified version of the Kap restructurer and the Vast restructurer were used as representatives of available parallelizing compilers.

Generally, we have concluded that available automatic restructurers have a limited effect on the performance of the given set of benchmark programs, because of: loops constructed from `IF` and `GOTO` statements; loops containing call or I/O statements; loops with abnormal entries or exits; unresolvable data dependences; conservative use of synchronizations; compiler data structure overflow; and poor choices in which loop of a loop nest to parallelize.

Most individual restructuring techniques failed to improve the performance of the benchmark suite. *Scalar expansion* and *reduction replacement* showed significant performance gains in a large portion of the benchmark suite. *Strip-mining* improved performance in some programs and decreased it in others and *Recurrence replacement* always worsened performance when applied. The rest of the techniques had insignificant effects on the benchmarks. The limited effectiveness of these techniques was caused by several factors: there were few loops in which the techniques could be applied; there was interference between techniques; compiler heuristics decided against their use; the techniques caused slowdowns that offset or dominated any speedups; and the techniques' application did not give new opportunity to exploit parallelism. The fact that some techniques have a negative performance impact is a particular concern. It shows that two common assumptions are not necessarily correct: First, a newly proposed technique may work well in a demonstration environment, but it fails for compiling ordinary programs. Second, we cannot justify new techniques arguing that they may speed up at least rare code patterns; we have to consider the potential performance degradation on the rest of the program.

Although we have shown that current automatic restructurers are limited in their effectiveness, we have also found that there exists room for significant improvement. We have derived this from manually restructuring the Perfect code ADM using techniques that we believe to be automatable. The findings for ADM are similar to those for other Perfect codes, reported elsewhere [11, 14]. Among the areas we identified are: certain implementation issues, advances of existing techniques, new analysis and transformation techniques, improved drivers for techniques and compiler passes, interprocedural optimization, and run-time driven optimization. The major performance gain is attributable to analysis and restructuring techniques that enable the recognition of parallelism. This is important because it benefits equally all parallel machines. While there may be different transformations to implement parallelism on different architectures, the successful recognition of parallelism is a precondition for all compilers. In our experiments techniques that were newly proposed in the literature were not used to achieve the reported speedups, and the areas of improvements we identified do not seem to be adequately addressed in recent literature. However, the main goal of this paper was to quantify the effectiveness of technology available in compilers. A comprehensive evaluation of more recently proposed technology will take considerable effort in future projects.

Most of all, we hope that this paper has shown the importance of collecting empirical data in the design and research of parallelizing compilers. There is both a need and room for further studies that will uncover little or sometimes negative effects in new technologies and will highlight areas that are most promising.

References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[2] Zahira Ammarguellat and Luddy Harrison. Automatic Recognition of Induction & Recurrence Relations by Abstract Interpretation. *Proceedings of Sigplan 1990, Yorktown Heights*, 25(6):283–295, June 1990.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Thrid ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 39–50, April, 1991.

[4] Clifford N. Arnold. Performance Evaluation of three Automatic Vectorizer Packages. In *International Conference on Parallel Processing*, pages 235–242, 1982.

[5] E. Ayguadé, J. Labarta, J. Torres, and P. Borensztejn. GTS: Parrallelization and vectorization of tight recurrences. In *Proceedings Supercomputing '89, Reno, Nevada, November 13-17*, pages 531–539. ACM Press, 1989.

[6] Robert N. Braswell and Malcolm S. Keech. An Evaluation of Vector Fortran 200 Generated by Cyber 205 and ETA-10 Pre-Compilation Tools. In *Proc. Supercomputing '88*, pages 106–113, 1988.

[7] David Callahan, Jack Dongarra, and David Levine. Vectorizing Compilers: A Test Suite and Results. In *Proc. Supercomputing '88*, pages 98–105, 1988.

[8] Doreen Y. Cheng and Douglas M. Pase. An Evaluation of Automatic and Interactive Parallel Programming Tools. In *Proc. Supercomputing '91*, pages 412–422, 1991.

[9] Ron Cytron, David J. Kuck, and Alex V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. *Special Issue of Computer Physics Communications devoted to the Proceedings of the Conference on Vector and Parallel Processors in Computational Science II*, 37:39–48, 1985. Invited paper.

[10] Ulrich Detert. Programmiertechniken für die Vektorisierung. In *Proc. Supercomputer '87, Mannheim, Germany*, June 1987.

[11] R. Eigenmann, J. Hoeflinger, G. Jaxon, Zhiyuan Li, and D. Padua. Restructuring Fortran Programs for Cedar. *to appear in Concurrency: Practice and Experience*, 1992.

[12] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar fortran and its restructuring compiler. In A. Nicolau D. Gelernter, T. Gross and D. Padua, editors, *Languages and Compilers for Parallel Computing II*. MIT Press, 1990.

[13] Rudolf Eigenmann and William Blume. An Effectiveness Study of Parallelizing Compiler Techniques. *Proceedings of ICPP'91, St. Charles, IL*, II:17–25, August 12-16, 1991.

[14] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, August 1991.

[15] M. Girkar and C. Polychronopoulos. Optimization of data/control conditions in task graphs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[16] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 26-28, 1991*, 1991. Available as *SIGPLAN Notices*, vol. 26, no. 6, pp. 15-29, June 1991.

[17] Jay Hoeflinger. QCD Optimization Report. Technical Report 1115, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, 1991.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, Jan. 1988.

[19] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A compilation scheme for macro-dataflow computation on hierarchical multiprocessor systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[20] V.P. Krothapalli and P. Sadayappan. Removal of redundant dependences in DOACROSS loops with constant dependences. In *SIGPLAN NOTICES: Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, April 21-24*, pages 51–60. ACM Press, 1991.

[21] Kuck & Associates, Inc., Champaign, Illinois. *KAP User's Guide*, 1988.

[22] Zhiyuan Li and Walid Abu-Sufah. On Reducing Data Synchronization in Multiprocessed Loops. *IEEE Trans. on Computers*, C-36(1):105–109, Jan., 1987.

[23] G.R. Luecke, J. Coyle, W. Haque, J. Hoekstra, H. Jespersen, and R. Schmidt. A comparative study of KAP and VAST: two automatic preprocessors with Fortran 8x Output. *Supercomputer 28*, V(6):15–25, 1988.

[24] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *SIGPLAN NOTICES: Proceedings of the ACM SIGPLAN 91 Conferen ce on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 1–14. ACM Press, 1991.

[25] Samuel Midkiff and David Padua. Compiler Algorithms for Synchronization. *IEEE Trans. on Computers*, C-36(12):1485–1495, December 1987.

[26] H. Nobayashi and C. Eoyang. A Comparison Study of Automatically Vectorizing Fortran Compilers. *Proc. Supercomputing '89*, 1989.

[27] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[28] Paul Petersen and David Padua. Machine-independent evaluation of parallelizing compilers. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1992. CSRD Report No. 1173.

[29] M. Berry; D. Chen; P. Koss; D. Kuck; L. Pointer, S. Lo; Y. Pang; R. Roloff; A. Sameh; E. Clementi, S. Chin; D. Schneider; G. Fox; P. Messina; D. Walker, C. Hsiung; J. Schwarzmeier; K. Lue; S. Orszag; F. Seidl, O. Johnson; G. Swanson; R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Jour. of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[30] M.E. Wolf and M. Lam. A data locality optimizing algorithm. In *SIGPLAN NOTICES: Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 30–44. ACM Press, 1991.

[31] M.J. Wolfe. More iteration space tiling. In *Proceedings Supercomputing '89, Reno, Nevada, November 13-17*, pages 655–664. ACM Press, 1989.

[32] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proceedings of the 1991 Int'l. Conf. on Parallel Processing, St. Charles, Illinois, August 12-16*, pages 26–30. CRC Press, Inc., 1990. Vol. II - Software.

[33] Chuan-Qi Zhu and Pen-Chung Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Trans. on Software Eng.*, SE-13(6):726–739, June 1987.