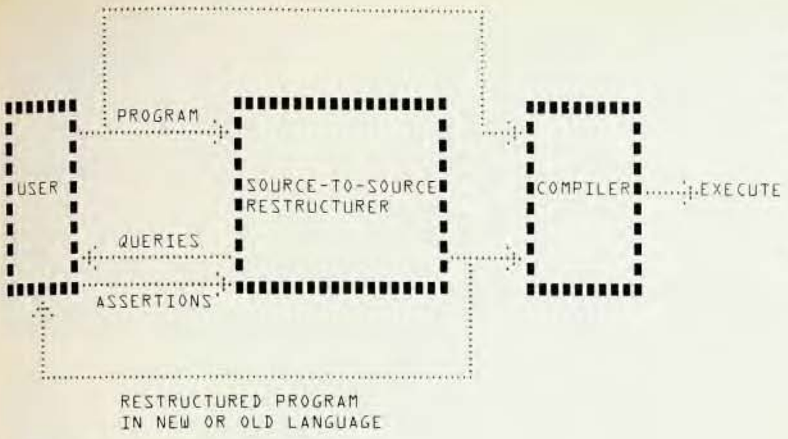

PROGRAM
USER
SOURCE-TO-SOURCE RESTRUCTURER
COMPILER
EXECUTE
QUERIES
ASSERTIONS
RESTRUCTURED PROGRAM IN NEW OR OLD LANGUAGE

Program development with automatic restructuring for concurrent processing. In the "source-to-source" scheme illustrated here, an automatic restructurer takes a user's source program, asks questions, receives answers and other statements, and then produces and prints a new source program, which the user can compile and execute. The arrow returning to the user indicates that this program development process is iterative. Figure 1

Supercomputers simply need supercompilers that automatically restructure programs for concurrent processing.

David J. Kuck and Michael Wolfe

To take full advantage of the ever more complex architectures of the latest supercomputers, users are having to write increasingly intricate programs. Thus, one of the most pressing questions associated with the powerful new concurrent processors is how to make them practical for ordinary users. In our opinion, answering this question means figuring out how to use algorithms, languages and compilers to give users four crucial capabilities and tools, namely

▶ The ability to use old programs, in old languages, as well as new programs, in old or new languages.

▶ New languages that allow one to express, in a well-structured form, algorithms that are amenable to parallel processing.

▶ Software that is able to exploit effectively all available architectural features, for use in developing and compiling programs in old and new languages.

▶ Packages and library routines that provide standard algorithms that are very amenable to parallel processing.

In this article we discuss the rationale for these objectives and our strategy for achieving them. We will argue that the best approach is the construction of powerful compilers that can automatically restructure source programs to exploit a machine's ability to do many computations concurrently.

Allowing for evolution. The purpose of the first of the four objectives above is to allow users to approach new machines without having to rewrite their programs in a new style or a new language. This makes for an easy transition from an old machine to a new machine, that is, it provides for architectural evolution. The ability to use old languages is probably a necessary condition for the general acceptance of a new machine.

The second and third objectives above, taken together, allow users to learn a new language or new features in an old language, especially if the program-development system can translate the old language to the new. This allows for language evolution as the user moves from familiar programs to new high-performance programs that are easier to understand. New language features are not a sufficient condition for the success or acceptance of a new machine.

New languages should permit the user to make assertions about the program that allow faster execution. In fact, the program-development software should query the user for such assertions.

Packages and library routines, mentioned in the fourth objective above, have always been important to computer users. However, as a part of program-restructuring techniques, they may lead to new and powerful program-development systems.

What language to use?

Debate over which programming language to use has been going on for many years and will probably continue indefinitely. In some ways, the discussion of this subject is similar to that of natural languages. People are biased by what they understand and grew up with, but are willing to learn a new language if their livelihood or interests depend on it.

Programming languages are, of course, much simpler and easier to learn than natural languages. Furthermore, even though a certain program has existed in an organization for many years, its parts may be rewritten with a frequency that has the effect of a complete rewrite of the code every few years. Thus, switching to a new language might not seem to be so difficult.

Nevertheless, whereas natural languages may follow commerce and ideas across national borders, computer users need compelling arguments to change languages. Probably the only

Continued on page 69

David Kuck is professor of computer science at the University of Illinois at Urbana–Champaign. Michael Wolfe is technical manager at Kuck & Associates, Inc., in Champaign, Illinois.

two arguments that could effect a change of language or computer system are faster computation and easier programming. Speed is easy to measure, but ease of use is somewhat subjective, and both require the user to invest substantial time to make a comparison. Thus, change has been slow to come.

Since its introduction in the late 1950s, FORTRAN has been the established programming language of science and engineering. Just as natural languages evolve with time, so has FORTRAN. New definitions were released in 1966 and 1977, and another, FORTRAN 8x, is planned for the 1980s. These have tended to keep the language current and to retain users. We have never argued that FORTRAN is a good language for supercomputers, but we are forced to deal with it because of its dominance.

Computation speedup can come to users of a given algorithm or program in two ways, through a new computer system or through a new optimizing compiler. Traditionally, increases in circuit speeds alone produced a steady flow of faster and faster computers. As the rate of increase in the speed of the raw hardware has diminished, new computer architectures have kept up the increases in computer speed. However, as the organization of computers has become more and more complex, the effective speed obtained by most users is a smaller and smaller fraction of the peak speed of a machine.

If we define efficiency as the ratio of effective to peak speed, we observe that such machines as the Cray-1 or Cyber-205 give most users only about 10% efficiency through their FORTRAN compilers. Of course, efficiencies of 50% or higher are possible on some computations if a user is willing to work hard enough at rewriting a program. Manufacturers have provided FORTRAN extensions that can lead to higher speeds if properly used, but it is sometimes difficult to know how to use the extensions effectively.

Thus, we see that compilers that can automatically restructure source programs to exploit a machine's features more effectively have the potential of improving performance. We will argue later that this is one of the key areas in which a high payoff is possible in the near future. The alternative of manually restructuring programs to achieve high performance, while often effective, violates all definitions of "ease of use."

The ease of use of a programming language has many aspects and is certainly much harder to measure than computation speed. Computer scientists and users have introduced many new languages and many new programming styles, all of which have as advertised features "ease of use" in one sense or another. Ease of use applies to such activities as writing a program, debugging a program and moving a program to another machine. The existence of a large user community contributes greatly to ease of use, as does the support of the software by a large number of machine manufacturers.

One reason why this subject has led to so much emotional discussion over such a long period of time is simply that it has so many countervailing aspects. For example, few people would argue that FORTRAN is one of the easier languages in which to write a new program, but on the other hand, because of its simplicity and established position, FORTRAN's portability is relatively good.

Computer-science journals are filled with hundreds of language definitions and extensions that are demonstrably "better" than FORTRAN in some way. Most have never made it beyond a small cult of users for one reason or another. One of the most famous is APL, which has very powerful expressive powers, still has a very vigorous following, and has enjoyed a number of implementations. Yet, lacking anything near "major" use after nearly twenty years, it will probably always remain in that position.

There is no simple explanation for this. All of the factors in the definition of "ease of use" are involved. Probably the most important are endorsement by a number of major manufacturers (which is difficult to obtain) and portability, which seems to explain the success of the language PASCAL and the operating system UNIX far better than does their superiority of conception.

Program-development systems

Assuming a user has a good algorithm to solve a problem on some machine, the remaining problem is to obtain high performance as easily as possible. A good programming environment is essential for this and should include a good programming language, a powerful editor, compiler and debugger, as well as a rich library of application packages. Using these tools, a programmer can quickly and painlessly transform an algorithm into a good program. We feel that the choice of language is much less important than the rest of the environment.

The program-development system must also be able to restructure a program, that is, transform it into a form more suitable for the target machine. Restructuring can be done in the compiler, or on a "source-to-source" basis, producing an updated source program for the user to maintain.

The TAMPR system at Argonne National Laboratory has successfully tailored individual source programs to different machines or to several applications. It can, for example, change program arithmetic from complex to real, or change the array storage mechanism from packed storage to full array storage. The Gibbs Project at Cornell University is a more ambitious program-development system, targeted specifically at providing physicists with a rich programming environment.

We see program development as an iterative process, as figure 1 indicates. The user enters a program into the program-development system, which analyzes it and possibly restructures it for the target machine. The system can query the user to get additional information that can lead to an optimization of the program. Finally, the program can be compiled and tested.

Because program restructuring can deliver enormous payoffs, and because the necessary transformations are essentially independent of programming languages, we will stress restructuring here. We will not argue for or against any particular language, leaving that to the user's taste. We will, however, argue very strongly for powerful program restructurers, and we will illustrate our discussion with FORTRAN-like examples. We have demonstrated that restructuring FORTRAN programs to exploit today's supercomputers can be done effectively. Languages that show parallelism explicitly may simplify the analysis of programs for parallel processors, but will probably not change the difficulty of optimizing a program to exploit a particular architecture.

Power of program restructuring

The simple program of figure 2a will help us illustrate some techniques for restructuring. We will first see how to analyze a program to expose its potential for parallel processing in a way independent of the target machine. One can use the results of this analysis to transform the program further to exploit a particular architecture.

Observe that the scalar on the left-hand side of the first assignment statement must be expanded to an array to prevent an "assignment bottleneck"; this is done in a reversible way. Figure

2**b** shows the result of this transformation, and also the program's "dependence graph." Nodes in the dependence graph correspond to statements; arcs extend from the generation of a variable to its use; brackets show loops. (Also see figure 3.)

We can discover the program's amenability to parallel processing from the dependence graph, and we can use this information to restructure the program to get the best performance on the target machine. The restructurer is built to characterize the target machine and to decide how to transform the program based on answers to the following questions:

▶ What is the architecture of the machine? Two examples are vector instruction machines and multiprocessor machines.

▶ Are the operands held in memory or registers? The CDC Cyber-205 holds the vector operands in memory, while the Cray-1 uses vector registers.

▶ Do irregular array accesses incur any penalty? On the Cyber-205, array references that are not "stride-1," that is, array references that do not access consecutive array elements, must be gathered with a separate vector instruction. On the Cray-1, array references with a fixed stride do not slow the machine down unless the stride is a multiple of 8; however, indexed array references, A[IP[I]] for example, cannot be done in a vector way.

▶ What is the overhead? On the Cyber-205, a vector instruction has a large startup time. Hence, one wants to use vectors with a large number of components to amortize the startup time. On the Cray-1, vector startup times are smaller, so small vectors are not as detrimental to performance.

Now we will restructure our example program in several ways. For a memory-to-memory target machine like the Cyber-205, we might distribute and interchange the loops as in figure 2**c**. Vectorizing the DO I loops over the statements for T and Z gives the largest vectors, reducing the vector startup overhead. Notice also that we reordered the statements so as to reduce the serial loop overhead. The dependence cycle in the statement for D is satisfied by executing the DO I loop serially, and the DO J loop can be vectorized. Now we can generate code, as shown in figure 2**d**. The notation here means that in the first statement, vectors B[*,J] and C[*,J] of length 300 (all elements denoted by *) are added and assigned to vector T[*,J], and so on. If the arrays B, C, Z and K are all dimensioned [300,80], then the first loop may be "collapsed" into one long vector operation, eliminating the problem of multiple startup times.

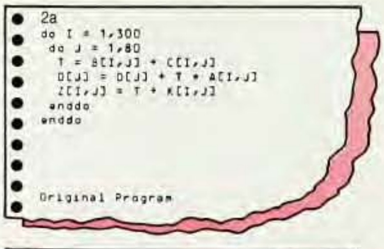
If the arrays are stored as columns, this restructuring scheme might not execute very well on a real Cyber-205, because the array accesses in the second loop would require a separate "gather" instruction. Figure 2**e** shows a better choice. Here the temporary array T has been partially shrunk, because both dimensions are not necessary. The second statement must now be executed as a vector sum reduction, and the Cyber-205 does have such an instruction. Code generation would now produce what we see in figure 2**f**.

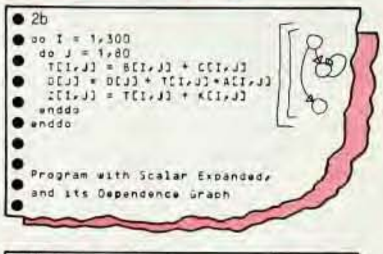
For a vector-register machine such as the Cray-1, the restructuring decisions would be quite different. The Cray loads 64 elements of each array into registers and operates on the registers. Since the load instruction slows down for array strides that are multiples of 8, accessing in the "I" direction in an array could cause problems, because it would lead to strides of length 80 if the arrays were stored as rows.

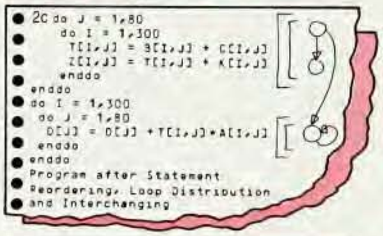
Finally, suppose we have a multiprocessor architecture. Now we want the parallel loop of figure 2**b** on the outside, with scalar computation inside. Scheduling becomes a problem and we want to match the outer loop to the number of processors available as well as possible. In a machine with 80 processors we would want to interchange the loops. For a machine with 60 processors we would block the outer loop as shown in figure 2**g**. The instruction doacr (do-across) denotes a loop that assigns its iterations to processors indexed by proc. Thus processor 1 executes iterations 1, 61, 121, . . . , processor 2 executes iterations 2, 62, 122, . . . , and so on. In some cases alternate processor assignments are preferable, and we must generally be concerned about data movement and synchronization time in multiprocessors.

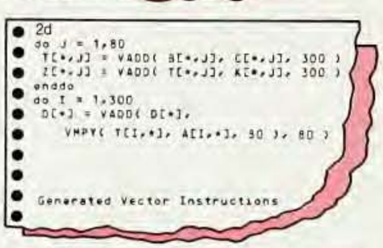
It should be clear that even simple programs lead to a number of complexities in restructuring for compilation. Few humans can do this as well as a powerful program restructurer.

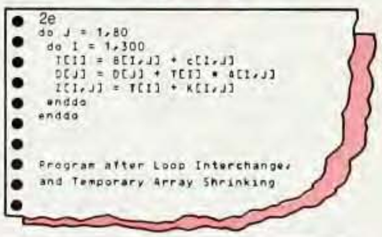
Various studies[1] have assessed the effectiveness of automatic restructuring. It is generally believed that the Cray-1 and Cyber-205 can achieve factors of two or four over scalar performance for programs that yield to the

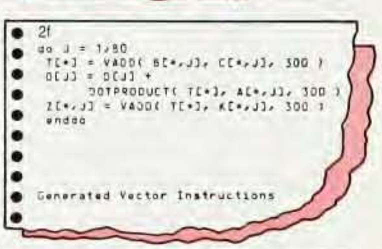

2a
```
do I = 1,300
 do J = 1,80
  T = B[I,J] + C[I,J]
  D[J] = D[J] + T * A[I,J]
  Z[I,J] = T + K[I,J]
 enddo
enddo
```
Original Program

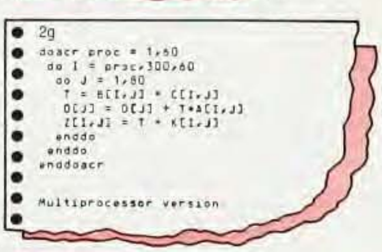

2b
```
do I = 1,300
 do J = 1,80
  T[I,J] = B[I,J] + C[I,J]
  D[J] = D[J] + T[I,J]*A[I,J]
  Z[I,J] = T[I,J] + K[I,J]
 enddo
enddo
```
Program with Scalar Expanded, and its Dependence Graph


2c
```
do J = 1,80
    do I = 1,300
     T[I,J] = B[I,J] + C[I,J]
     Z[I,J] = T[I,J] + K[I,J]
    enddo
enddo
do I = 1,300
 do J = 1,80
  D[J] = D[J] +T[I,J]*A[I,J]
 enddo
enddo
```
Program after Statement Reordering, Loop Distribution and Interchanging


2d
```
do J = 1,80
 T[*,J] = VADD( B[*,J], C[*,J], 300 )
 Z[*,J] = VADD( T[*,J], K[*,J], 300 )
enddo
do I = 1,300
 D[*] = VADD( D[*],
     VMPY( T[I,*], A[I,*], 80 ), 80 )
```
Generated Vector Instructions


2e
```
do J = 1,80
 do I = 1,300
  T[I] = B[I,J] + c[I,J]
  D[J] = D[J] + T[I] * A[I,J]
  Z[I,J] = T[I] + K[I,J]
 enddo
enddo
```
Program after Loop Interchange, and Temporary Array Shrinking


2f
```
do J = 1,80
 T[*] = VADD( B[*,J], C[*,J], 300 )
 D[J] = D[J] +
       DOTPRODUCT( T[*], A[*,J], 300 )
 Z[*,J] = VADD( T[*], K[*,J], 300 )
 enddo
```
Generated Vector Instructions

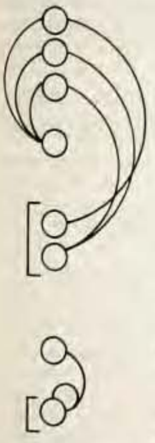

2g
```
doacr proc = 1,60
 do I = proc,300,60
  do J = 1,80
   T = B[I,J] + C[I,J]
   D[J] = D[J] + T*A[I,J]
   Z[I,J] = T + K[I,J]
  enddo
 enddo
enddoacr
```
Multiprocessor version

automatic vectorizers of these machines. Such factors are substantial—a new computer system providing such speedups would be regarded as a major step forward.

Neither the Cray nor the CDC vectorizers are nearly as powerful as the best vectorizers that could be written for those machines. Clifford N. Arnold demonstrated[2] this in a study of the "Livermore loops," 18 very simple FORTRAN loops developed at Livermore to compare the speeds of supercomputers. Four of the loops ran more than five times faster on the Cyber-205 after use of a vectorizer known as the "KAP," as compared to the standard CDC compiler. The KAP, our own commercial product, is an outgrowth of the "Parafrase" system at the University of Illinois. Recent Parafrase experiments with automatic restructuring and manual algorithm change on larger programs are summarized in reference 3.

More recently, a test showed[4] that the Fujitsu VP200 compiler is able to vectorize more loops than the Cray CFT 1.11 compiler. The test consisted of runs on both machines of three benchmarks that contained 33, 73 and 88 loops, respectively. The Cray compiler vectorized 11, 42 and 58 loops in the three benchmarks, respectively, while the Fujitsu compiler vectorized 19, 51 and 64 loops.

Neither the Cray nor the CDC compilers can interchange loops, vectorize loops containing IF statements, or vectorize parts of loops while leaving the rest serial, for example. Both the KAP and the Fujitsu compiler can do such transformations.



```
    SUBROUTINE CALC
    S1 = X + Y
    S2 = X - Y
    S3 = X * Y

    IF (S1 + S2 .GT. S3)
       GOTO 100
    DO 50 I = 1,N
     A(I) = S1 + B(I)
     B(I) = S2 + S3
 50 CONTINUE
    GOTO 999
100 S = 0
    DO 150 I = 1,N
     S = S + A(I)
150 CONTINUE
999 RETURN
```

Dependence graph, showing how lines in a computer program (above right) depend on each other because of the flow of data in the program. The flow of control in a program—indicated by statements such as GO TO, IF and DO—is not relevant in making up a dependence graph. A dependence graph is an intermediate step in decomposing a problem for concurrent processing. Figure 3

Assertions

A restructurer often needs more information than is present in a FORTRAN-like source program. For instance, the loop in figure **4a** can be vectorized if M is greater than or equal to zero, but not if M is negative. The program gives no information about the sign of M. A powerful restructurer will be able to accept such information from the programmer in the form of assertions, as in figure **4b**. Assertions may be tested at run time; invalid assertions would cause an ASSERT-ERROR. Some assertions, like the one in figure **4c**, would require a nontrivial validation test, which might be done while the program is being debugged. Some information similar to that given in assertions can be derived from the source program itself. For example, if the program in figure **4a** were preceded by the statement IF M>0 THEN, the restructurer would gain as much information as it does from the assertion in figure **4b**.

The restructurer can use assertions of the sort discussed above to improve the quality of the dependence graph and expose more opportunities for parallel processing. There is another broad class of assertions that the restructurer can use to make optimization choices, that is, decisions between two or more equally valid ways to generate code for a given program. Such choices are made on the basis of run-time probabilities. In this class of assertions are statements about loop length or branching frequencies. In figure **4d**, for example, if the programmer knows that N will be much larger than M, then the restructurer can use this information to decide to interchange the loops and vectorize the DO I loop to get a longer vector, instead of simply vectorizing the DO J loop. Assertions used to make optimization choices need not be validated, because an incorrect assertion will not make the compiler generate incorrect code. However, the program may execute more slowly, so validation is desirable.

When, for whatever reason, the restructurer wants more information about the program, it can put queries to the user. The biggest problem is phrasing the query in terms of the original program.

When assertion information is not available, there are still several ways to restructure the program. In the absence of information on dependence, the most obvious action is to make a conservative choice—assume dependence unless nondependence can be proved. Where the decisions depend on the data, a useful directive to the restructurer is to produce two versions of the program and choose at run-time which version to execute. For example, a programmer who does not know the sign of M in figure **4a**, but thinks that M will be greater than or equal to zero often enough to single out that case, might simply preceed the program with the directive COMPILE WITH M >= 0 AND M<0. Then, two versions of the loop will be compiled, one vectorizable and the other an arithmetic recurrence. Only one source loop would need to be maintained, but the best performance could still be expected.

Automatic algorithm change

Currently we can automatically change small algorithms in programs to make them better suited to certain machine architectures. For example, first-order linear recurrences can easily be compiled using appropriate fast recurrence solvers. This algorithm change inserts a new algorithm that may be far beyond the understanding of the user in terms of parallel thinking. It may also have numerical properties that are different from the original program. But it can achieve

4a

```
do I = 1,N
 A[I] = A[I+M] + B[I]
 enddo
```

4b

```
assert M > 0
do I = 1, N
 A[I] = A[I+M] + B[I]
 enddo
```

4c

```
assert IP is a permutation array
do I = 1,N
 A[IP[I]] = B[I] + C[I]
 enddo
```

4d

```
do I = 1,N
 do J = 1,M
  A[J,I] = B[J,I] + C[J,I]
 enddo
enddo
```

substantial speedups in sections of the program where it is used. As another example, a program restructurer can "recognize" the computation being done; for instance, it might identify particular recurrences as inner products.

Pattern matching can lead to other kinds of algorithm change. Generally, nonlinear recurrences are very difficult to speed up. However, one can handle certain numerical nonlinear recurrences by appropriate changes of variables, and one can tabulate other nonlinear recurrences for use whenever an appropriate pattern occurs. Similar techniques work for seminumerical nonlinear recurrences. For example, finding the index and magnitude of the maximum element of an array can be done rapidly on many architectures, and pattern matching can recognize such computations in source programs.

The notion of "recognizing" what a program is doing could be a useful one. If we can recognize an inner product, then we can certainly recognize a matrix multiplication or a partitioned matrix multiplication. Of course, there are limits to this; it would not be difficult to write a matrix multiplication program that would defy recognition by any program (and most humans!). However, most users write fairly straightforward programs. We can put these programs into a canonical form, from which recognizing what the algorithm is doing may not be difficult.

Suppose we were able to do this for much of linear algebra. There might be half a dozen distinct types of linear system solvers that are frequently used, as well as a dozen obscure ones. Assume a user writes a linear system solver on a serial machine for matrices of some very special type and wants to run it on a supercomputer. The system we propose would be able to recognize (perhaps after asking some questions) that the program is a linear system solver with certain variations. If the user's algorithm were based on Gaussian elimination with partial pivoting, and it performed poorly on the supercomputer of choice because of the column-access, the system would select Gaussian elimination with pairwise pivoting, insert the code variation, and inform the user of the result to check whether the numerical accuracy is acceptable. If a user wrote a Gaussian elimination program to work on banded matrices, the system would recognize the algorithm and ask whether the problems to be solved were well-conditioned. If the user responded "no, ill-conditioned," the system would replace the program with a Given's solver, properly indexed for banded systems.

If such an approach were possible in one area, it would probably be possible in other areas. We feel that because we already change small algorithms automatically, we should be able to do the same with certain larger algorithms. It also seems clear that if users are able to interact with the system, the range of successful application will be broadened significantly.

The powerful, interactive program restructurers that we have argued for can be very useful in moving programs to new machines or getting more performance from existing machines. It is clear that a powerful restructurer can make tradeoffs better than most users can, simply because modern supercomputers are so complex. A side-effect of using a powerful interactive restructurer should be that users will learn, from the kinds of decisions the system makes, more about how to use the machine effectively.

* * *

Our work is supported in part by National Science Foundation grant NSF MCS81-00512, by US Department of Energy contract DOE DE-AC02-ER10822 and by Kuck & Associates, Inc.

References

1. R. Kuhn, D. Padua, eds., *Tutorial on Parallel Processing*, IEEE Computer Society Press, Silver Spring, Maryland (August 1981).
2. C. N. Arnold, ICPP Proc. (1982), page 235.
3. D. J. Kuck, "Supercomputer Prospectives," Invited Paper for the 4th Jerusalem Conference on Information Technology, IEEE Computer Society Press, Silver Spring, Maryland (May 1984).
4. R. Mendez, SIAM News, March 1984.
5. D. J. Kuck, *The Structure of Computers and Computations*, vol. 1, Wiley, New York (1978).
6. D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, Computer **15**, 58 (1982). □

Response to McGraw

We have no argument with the development of new languages, such as functional languages. However, a language whose key advantages are "in what they [programmers] cannot do" will find resistance in the realm of high-speed computing. For example, McGraw states that the second program loop on page 72 is not programmable in a concurrent way in any applicative language. If this were the key loop in a program, the programmer would want to be able to optimize its performance. Programmers simply will not use languages that do not allow them to get the performance they require.

Our second point is philosophical. McGraw states that "Effective use of multiprocessors depends heavily on a user's ability to program algorithms that contain large amounts of concurrency." We agree that an inappropriate algorithm will perform poorly on a multiprocessor. However, we feel that automatic restructuring is an important technique for developing new programs for any supercomputer. In addition, we feel that the language should be suited to the application, rather that to some machine or model of execution.

We are not saying that FORTRAN is the best language for any application—we have never said that it is. The restructuring we propose is independent of language. Certainly, dealing with problems such as COMMON blocks makes FORTRAN one of the hardest languages to restructure. McGraw blasts FORTRAN because of the "aliasing" problems of COMMON blocks, parameters and pointers in the FORTRAN extensions used at Livermore. In fact, any program that depends on "aliasing" is an illegal FORTRAN program, according to section 15.9.3.6 of the FORTRAN 77 standard. Any language that allows separate compilation will have similar problems when programmers use their knowledge of the implementation—in this case, their knowledge that parameters are passed by reference—to do illegal things. With regard to pointers, we have spoken to users of the FORTRAN extension at Livermore. We find that they use pointers as a way to perform dynamic array allocation, but that they use them in well-structured ways that would allow program restructurers to deal with the majority of cases and get good performance.

Our previous criticism[6] to which McGraw is reacting was aimed at the traditional data-flow computer design philosophies, about which we are still pessimistic, regardless of the languages that might be involved. We believe that language and architecture designs must each be able to stand on their own. Unfortunately, it seems that many people embrace the data-flow philosophy because they are ignorant of the power of automatic program restructuring and become convinced that restricted languages or programming styles are necessary to achieve high-speed computation. It should be realized that it is possible to compile FORTRAN for a data-flow machine and that it is possible to compile VAL or ID for a multiprocessor such as the University of Illinois Cedar system.

—DJK&MW