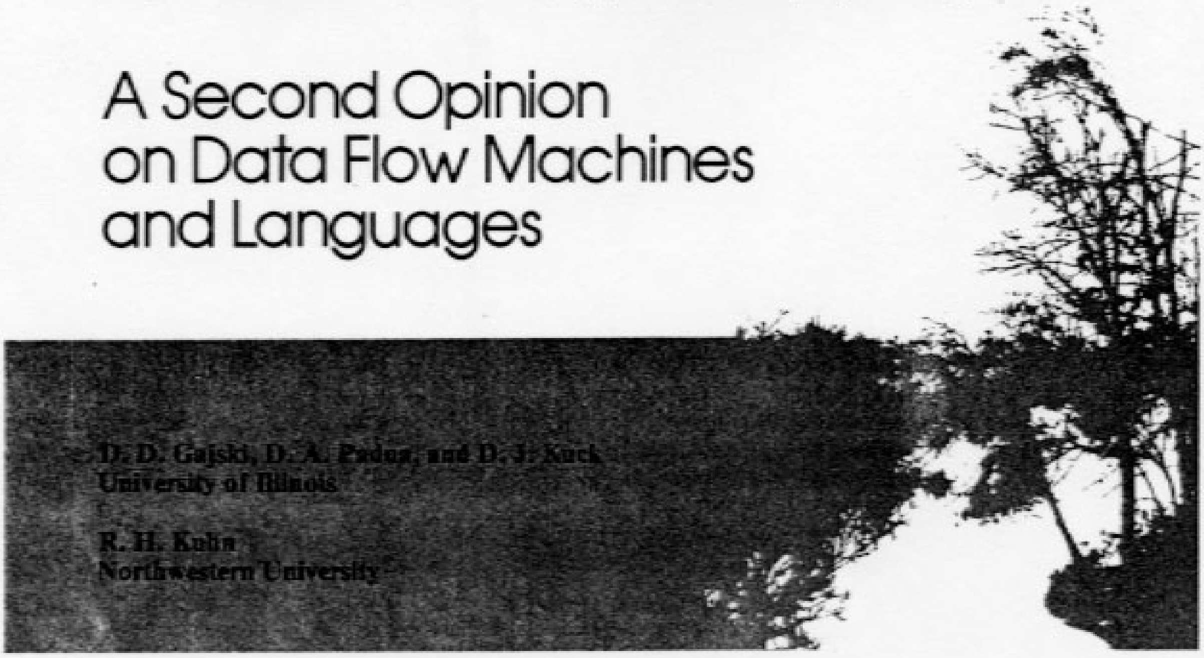
Due to their simplicity and strong appeal to intuition, data flow techniques attract a great deal of attention. Other alternatives, however, offer more hope for the future.

A Second Opinion on Data Flow Machines and Languages



D. D. Gajski, D. A. Padua, and D. J. Kuck
University of Illinois

R. H. Kuhn
Northwestern University

Simultaneity is a key to high-speed computation. Assuming hardware components of a given speed, it is the only remaining consideration in achieving raw speed. Simultaneity can be shackled by dependences, however, and years of hardware and software work have been devoted to understanding the types of dependences and how they can be obeyed or removed from a computation.

Dependence types. There are three types of dependence[1]: data, control, and resource. The first two arise in programs and the third in machines. Therefore, exact definitions depend on the type of language and machine under consideration, although many nearly universal dependences exist.

We will discuss three types of *data dependence*[1] (see Kuck et al.[2] for a fourth type): flow dependence, output dependence, and antidependence. *Flow dependence* exists from the computation to the use of a variable. *Output dependence* exists between two subsequent computations of the same variable. *Antidependence* exists from the use of a variable to its next computation. These three types ensure that the intended values are, in fact, used in a computation.

Control dependence types vary from language to language. For example, loop dependences exist from a loop header to each statement inside the loop, conditional dependences exist from an IF to its THEN and ELSE parts, and GOTO dependences exist from a GOTO to its destination.

Resource dependences arise when programs are compiled for and executed on a particular machine. For example, the existence of an adder and a multiplier that can be sequenced simultaneously by the control unit allows these two (but no more) arithmetic operations to be executed at once. A four-way interleaved memory allows simultaneous access to four words, but no more. A single program counter, a single arithmetic unit, and a single memory led to the so-called von Neumann machine, and these resource dependences were reflected in the definition of Fortran and other high-level programming languages.

Dependence observation. Given a problem to solve on some machine, it is useful to observe dependences at five points in the selection, preparation, and execution of an algorithm. These are in (1) algorithm choice, (2) programming, (3) compiling, (4) instruction processing (control unit), and (5) instruction execution (processor, memory, interconnection).

A given algorithm has certain built-in data dependences. For example, in certain iterative computations, an iterate must be computed before it can be used. However, other algorithms that solve the same problem might have less sequential dependence. For example, many highly concurrent algorithms to solve linear recurrences are known,[1] and the use of any of these relaxes the sequentiality between iterations.

Once an algorithm has been selected, the programming language and programming style used to express the algorithm can introduce additional dependences, as well as encode its inherent dependences. For example, a complex expression could be computed once, then stored, and subsequently used in several other expressions. This introduces flow dependences. Usually, programmers are not concerned with the number or type of dependences they introduce. Some languages or styles prevent or try to avoid certain types of dependences in programs. For example, the single-assignment approach advocated in data flow languages avoids output and antidependences.

Compilers can remove and/or introduce dependences. For example, a block of assignment statements in any language can easily be compiled into a form that obeys the single-assignment rule. In fact, all three types of data dependences can be removed automatically to produce a completely independent set of assignment statements[2]; statement substitution is used to remove flow dependences. On the other hand, two array variables must not have a dependence, but a compiler that examines only array names and not subscripts will introduce a spurious dependence. For example, if $1 \leq I \leq n$, there is no dependence between $A[2I]$ and $A[2I-1]$. This will be missed if the check is only for the array name A.

Consider the instruction processing carried out by a control unit. Assumptions about data, control, and resource dependence are always built into the hardware of a control unit. For example, the traditional von Neumann machine assumes that machine instructions are processed one at a time, with some simultaneity possible in multiple address instruction formats. Multifunction machines such as the CDC 6600 have look-ahead control units that examine two or more instructions and check dependences at runtime. If data and control dependences in the instruction stream allow and resources are available, the control unit can sequence several instructions at once.

More recent machines (e.g., the CDC Cyber 205 and the Burroughs BSP) have machine instructions that can express array operations such as vector add or recurrence operations such as inner product. If the compiler recognizes such an operation (because it is, for example, programmed sequentially or expressed in a vector extension to a sequential language), it can generate a single machine instruction that carries it out by using a fast, highly parallel algorithm. In parallel or pipeline machines, the control unit must access chunks of the operands, process them, and store chunks of the result until the operation is finished. In the case of recurrence operations, the control unit must carry out a sequence of steps. These might correspond to a complex dependence graph in which simultaneity was maximized when the machine was designed. For example, a vectorized inner product could be a vector multiply followed by a summation tree. Thus, an entire small loop from a sequential program has all of its dependences mapped onto the control-unit hardware for fast runtime sequencing.

The line between instruction processing and instruction execution is somewhat blurred across various types of computers. In machines with array instruction sets, the control unit knows, by virtue of the way in which its array instructions were implemented, exactly how to sequence its memory, processor, and interconnection network parts to carry out an operation. In other types of machines, the control unit decodes instructions, processes addresses, and decides that an instruction can be issued. The individual steps, however, are not executed until all of the dependences of the instructions are satisfied. This principle is used in the Scoreboard of the CDC 6600,[3] the Tomasulo algorithm of the IBM 360/91,[4] and in current data flow proposals. The point is that the processor, memory, and interconnection networks themselves can ensure satisfaction of all dependences. For example, if there is a great deal of randomness in a program arising, perhaps, from conditional statements or irregular subscripts, the intuitive notion is that little can be preplanned and that execution time-dependence handling is, in fact, necessary for fast computation.

Article overview. In this article we undertake two tasks. The first is to sketch the principles and practices of data flow computation and to point out a number of shortcomings of this approach to high-speed computation. The second is to sketch an alternative that leads to high-speed computation through higher-level use of dependence graphs.

The data flow approach and our alternative are roughly characterized in Figure 1. The data flow approach usually begins with a special programming language, which researchers hope can be easily compiled into a dependence graph. (Ordinary languages can also be used.) The problem then becomes one of efficiently mapping this onto a machine that has decentralized control hardware. Often, this machine is capable of exploiting substantially less parallelism than exists in the program because it lacks the hardware to cause the dependence graph constraints to be followed as quickly as possible at runtime. That is, queues of partially completed computations are relied upon to keep the machine busy, and the queue lengths absorb some of the parallelism of the program.

Our approach can begin with either a data flow language or an ordinary programming language. A program is first put into a standard form (normalized). Next, a dependence graph is generated by, for example, analysis of array subscripts. Then some arcs are removed by program transformation, and some nodes and arcs are abstracted* because they represent high-level constructs for

*Abstraction here refers to the process of reducing a subgraph to a higher-level node.

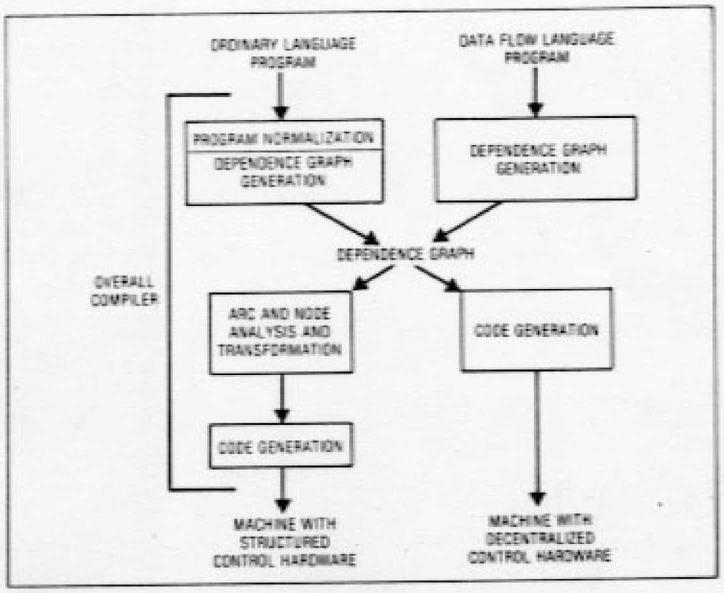

ORDINARY LANGUAGE PROGRAM
DATA FLOW LANGUAGE PROGRAM
PROGRAM NORMALIZATION
DEPENDENCE GRAPH GENERATION
DEPENDENCE GRAPH GENERATION
DEPENDENCE GRAPH
OVERALL COMPILER
ARC AND NODE ANALYSIS AND TRANSFORMATION
CODE GENERATION
CODE GENERATION
MACHINE WITH STRUCTURED CONTROL HARDWARE
MACHINE WITH DECENTRALIZED CONTROL HARDWARE

Figure 1. Comparison of methods.

which special algorithms (instructions) can be used. Finally, code generation produces high-level machine instructions that contain much of the dependence structure of the original program, but transformed for high-speed evaluation. The machine that executes the generated code can have a degree of simultaneity that is matched to the program and still execute it efficiently, because most dependence testing need not be done at runtime. However, this approach relies on compilation techniques more powerful than those usually assumed by data flow people.

Other sections of this article investigate the principles of data flow architecture and proposed data flow languages.

Data flow principles

In contrast to the sequential, one-instruction-at-a-time, memory cell semantics of the von Neumann model, the data flow model of computation is based on two principles:

(1) *Asynchrony*. All operations executed when and only when the required operands are available.
(2) *Functionality*. All operations are functions; that is, there are no side effects.

The first denotes an execution mechanism in which data values pass through data flow graphs as tokens and an operation is triggered whenever all input tokens are present at a node in the graph. The second principle implies that any two enabled operations can be executed in either order or concurrently.

Dynamic parallelism. Even when there is data dependence between operations of the same iteration of a loop, there is nothing to stop further iterations from proceeding, even though one iteration is not totally completed. This causes tokens to accumulate on certain arcs of the data flow graph. It is then no longer possible to declare a node executable by the presence of any two tokens on its input, as they might belong to totally different parts of the computation. There are five possible solutions to this problem:

(1) *The use of a re-entrant graph is prohibited.* That is, each stage of the iteration must be described by a separate graph. This solution obviously requires large amounts of program storage. It also requires dynamic code generation if the loop's iteration depth is only known at runtime. Both of these deficiencies can result in significant overhead in practical systems.

(2) *The use of a re-entrant graph is allowed, but an iteration is not allowed to start before the previous one has finished.* This approach does not allow for parallelism between iterations and requires extra instructions or hardware to test the completion of an iteration. It is used in the LAU system.[5]

(3) *The use of a data flow graph is limited by allowing only one token to reside on each arc of the graph at any time.* This is accomplished by allowing an operation to be executable only when all its input tokens are present and no tokens exist on its output arc. This approach, which implies sequential but pipelined use of the data flow graph, allows exploitation of more parallelism than do the previous two solutions. Pipelining is implemented through the use of acknowledge signals, which are returned to the nodes in the graph that generated those values by the nodes that consumed the values.[6,7] These acknowledge signals approximately double the number of arcs in the corresponding data flow graph, and therefore double the traffic through the data flow machine.

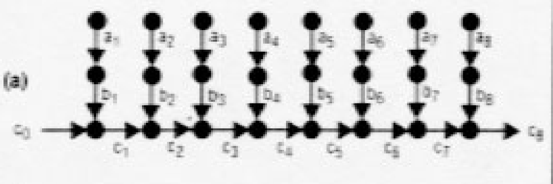
(4) *The tokens are assumed to carry their index and iteration level as a label.* This label is usually called *color*. A node is executable only if all input tokens have the same color. The labeling method permits the use of pure static code and enables maximum use of any parallelism that exists in the problem specification. This is clearly at the cost of the extra information that must be carried by each token and the extra nodes (instructions) for labeling and delabeling.[8,9] The penalty of this approach is obviously extra time for calculating labels, or extra hardware (silicon area) if calculation is concurrent.

(5) *The tokens are queued on arcs in order of their arrival.* This solution can deliver as much parallelism as the labeling approach, but requires large queues, which are very costly.

To compare the performance of data flow machines that use these five approaches, consider the program in Figure 2a. Assume that division takes three time units, multiplication two, and addition one. The hypothetical data flow machine has four processing units, each capable of executing any operation. We idealize the machine by assuming that memory and interconnection delays are zero.

Our example program dictates a certain order of execution, which is determined by the simplified data flow program in Figure 2b. Obviously, the critical path is $a_1, b_1, c_1, c_2, \ldots, c_8$, which results in a lower bound on execution of 13 time units.

(a)



(b)

```
input d,e,f
        $c_0 = 0$
for  i  from 1 to 8 do
    begin
        $a_i := d_i + e_i$
        $b_i := a_i * f_i$
        $c_i := b_i + c_{i-1}$
    end
output a,b,c
```

Figure 2. A nonsense example: (a) program; (b) simplified data flow graph.

Since there is one division, one multiplication, and one addition in each iteration of the loop, it will take $6 \times 8 = 48$ time units (Figure 3a) to execute the complete loop when using the one-iteration-at-a-time strategy described in approach (2), above. This is basically a sequential execution, and one processor would suffice. In practice, the computation is distributed over all four processors, but the utilization of processors remains at $12/48 = 0.25$.

The one-token-per-arc strategy (3) practically turns into pipelining of the block of assignment statements inside the loop, as shown in Figure 3b. Execution time is determined by the longest operation (division) in the loop. Thus, $3 \times 8 + 3 = 27$ time units are necessary, with utilization at $12/27 = 0.44$. Approaches (1), (4), and (5) are similar. They achieve the best performance and utilization, as shown in Figure 3c. They need only 14 time units, with utilization equal to $12/14 = 0.86$. However, a random-scheduling strategy (followed in many data flow architecture proposals) can result in less than optimal execution, as shown in Figure 3d, where 18 time units were needed to finish the computation. The detection of possible critical paths and scheduling along these paths is a problem that none of the proposed data flow machines have solved.

For comparison, a possible execution on a vector machine with a vectorizing compiler is shown in Figure 3e. A mediocre vectorizing compiler would detect that the first and second statements in the loop can be vectorized. The execution time is 18 and the utilization $12/18 = 0.66$. A good vectorizing compiler would detect the recurrence in the third statement, substitute a different algorithm, and lower the execution time to 14 with a utilization of one, as shown in Figure 3f. Note that recurrences arise frequently in ordinary programs.[10]

It is obvious from this simple example that the sequential machine offers the worst performance and the data flow machine with labeled tokens the best. Pipelined and vector machines are somewhere between those extremes, although the vector machine with a good optimizing compiler was competitive with the data flow machine in our example. Remember, however, that our models are gross oversimplifications of real machines. Since there are no hard facts on performance of data flow machines, it remains to be seen whether the overhead in token labeling, data storage, and instruction communication will lower their theoretical upper bound on performance.

Performance under a low degree of parallelism. The data flow graph can be considered the machine language of a data flow machine. Each node of the graph represents an instruction, and the arcs pointing from each node can be thought of as the addresses of instructions receiving the

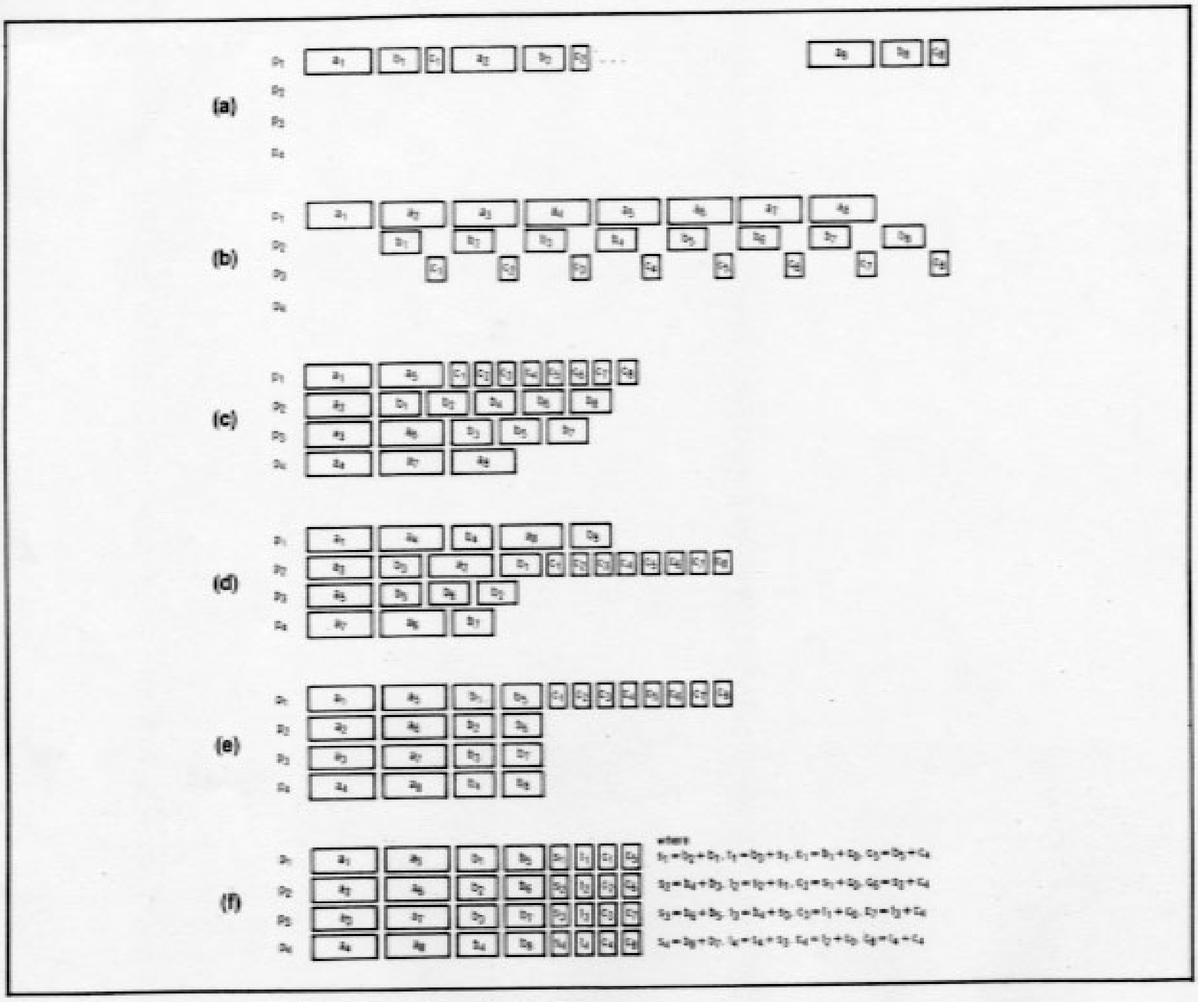


Figure 3. Comparison of data flow strategies (a) through (f).

result. Roughly speaking, a data flow machine consists of four components: an instruction memory that contains all instructions in the data flow graph, a set of processing units that perform the operations specified by each instruction, an arbitration network that carries instruction packets to appropriate processing units, and a distribution network that carries the result packets back to the instruction memory. Obviously, the instruction memory can be partitioned into several modules, to match the bandwidth of the processing units and communication networks. We assume also that one unit of time is needed to pass through each of the four components of our data flow model.

In contrast to the data flow model, the von Neumann model consists of a central processing unit and a memory. The central processing unit has a general-purpose register file and an arithmetic-logic unit. Each register-to-register operation takes one unit of time, as do fetch and store from memory.

To crudely compare the performance and the program size of these two models, we considered several programs that, like the one given by Arvind, Kathail, and Pingali,[11] integrate function *f* from *a* to *b* over *n* intervals of size *h* by the trapezoidal rule. We concluded that the data flow model apparently requires more instructions than the von Neumann model. This code inefficiency stems from two principles of the data flow model:

(1) *Distributed control.* Each datum (or path in the graph) is controlled individually. There are several separate SWITCH instructions (at least one for each variable assignment inside the body of a loop) that correspond to one BRANCH instruction in the von Neumann model. Similarly, several independent MERGE instructions substitute for one JUMP instruction.

(2) *No explicit storage.* Since only values are passed from one instruction to the other, values that do not change during the computation from one iteration to the other must circulate in one way or another through the system.

These redundancies lower the expected performance when the degree of parallelism (the number of operations executable in parallel) is equal to or greater than the ideal rate (the maximum possible number of operations executed concurrently).

In terms of raw speed on small programs, the von Neumann model requires less time. This performance advantage is the consequence of two things:

(1) *Instruction pipelining.* In von Neumann computers, the fetch, decode address generation, and execution phases of an instruction are allowed to overlap. Thus, each instruction averages only one time unit in a reasonably sequential code. On the other hand, the data flow model does not allow pipelining on the critical path. That is, each instruction must complete before the new one—which uses the result from the previous one—can start.

(2) *Local storage.* A data flow machine is basically a memory-to-memory machine, since there is no concept of storage. It usually helps to keep all the input parameters to a subroutine in high-speed, general-purpose registers, as in our examples. This lack of locality severely degrades the performance of the data flow machines on programs with a low degree of parallelism.

One might argue that although the data flow processor is slower in raw speed, it is faster overall because it contains many overlapped processing units operating in parallel. Still, the degree of parallelism must be taken into account. In a crude approximation appropriate to this case, the data flow machine can be thought of as a long pipeline. To keep the pipeline saturated, the degree of parallelism must be larger than the number of stages in the pipeline. Under low parallelism, the pipeline is not saturated for a long period of time and serious degradation of performance occurs. For comparison, Cray-1 computers have functional unit pipelines with five to eight stages in which register fetches are included. Data flow machines have pipelines many times longer. They include functional units, communication networks, and instruction memory. Therefore, we can expect them to perform poorly under low parallelism. If features for parallelism exploitation such as token labeling and array management are added, the performance under low parallelism becomes even worse.

Data flow machines require a parallelism of several hundred independent instructions to saturate the pipeline. Arvind et al.[11] have computed, for example, that for a data flow machine with 100-microsecond interprocessor communication time and 64 processing units, each of which performs a floating-point operation in 10 microseconds, the degree of parallelism to keep the machine saturated is 640. To date, the only programs with such a high degree of parallelism are computationally intensive numerical calculations that operate on large arrays of data. Unfortunately, data flow machines do not handle arrays of data very efficiently because of their emphasis on fine-grain, operation-level concurrency.

Structures storage. If tokens are allowed to carry vectors, arrays, and other structures in general, the result is a large transmission and storage overhead. This is particularly the case when operators modify only a small part (possibly only one element) of the whole structure. For this reason, Dennis[6] suggested that all structures be represented by a finite, acyclic, directed graph having one or more root nodes arranged so that each node can be reached over some directed path from some root node (that is, a forest of trees with possibly common nodes).

Arrays are stored as trees, with array elements at the leaves. For example, an array $A = [a_{i,j}]$, $1 \le i,j \le 3$ can be stored as a ternary tree. Obviously, trees of any order can be used for storing arrays.

According to the functionality principle of the data flow model, a data structure must be free of any side-effects. An easy way to accomplish this is to forbid any sharing or overlapping of structures. Since every structure would have its own private area of memory, there would be no side effects. However, this is prohibitively expensive since it requires each structure to be completely copied whenever its value is duplicated. The solution proposed by Dennis is to share the structures whenever possible and use the reference count technique. Each node of a structure has a reference count, which is the total number of pointers to that node from other nodes and tokens in the data flow program. For example, if a copy B of the array A is created, the pointer for B points to the same root node as the pointer for A (Figure 4a).

When an APPEND operator is used, it is necessary to copy all nodes with a reference count greater than one, as well as their successors, on the directed path from the root to the selected node. For example, if an array B' is obtained from B by setting $a_{23} = 0$, the structure in Figure 4b will be generated. Similarly, B'' can be obtained from B' by setting $a_{33} = 0$ (Figure 4c). The above two operations, setting $a_{23} = 0$ and $a_{33} = 0$, are completely independent of each other but cannot be executed concurrently; by the asynchrony principle of the data flow model, concurrent execution could result in two different structures, B' and B''' (Figure 4d), from which it is very difficult to obtain B''.

Here we see that a simple operation, such as setting a row or a column in a matrix to zero, requires sequential execution, which in turn significantly degrades performance for large structures. The performance degradation can come from two independent mechanisms used in this scheme. The first mechanism uses the reference count to share data. Therefore, there will be many unnecessary accesses to the memory in order to change the reference count without using data. This occurs for all operations that create or destroy pointers. For example, when a SWITCH operator destroys a token, the count must be decreased. The second mechanism uses tree structures to store arrays. If the order of the tree is small, large arrays will be stored as trees of considerable depth and therefore many memory references will be needed to access an array element. On the other hand, there is an unnecessary data transmission and wasted space due to excessively large memory blocks when the order is large.

To avoid the excessive storage demand and slow access time due to the functional semantics of the data structure operations, Arvind and Thomas[12] proposed I-structures, array-like data structures whose storage is allocated before expressions to produce them are invoked. Since an I-structure construction is not strictly ordered (to improve parallelism), it is possible that part of a program might attempt to read an element before that element's creation. Therefore, a presence bit is associated with every element, and an attempt to read an empty location causes deferral of the read operation. Unfortunately, when the element is finally created all deferred reads must be executed. Checking for those deferred reads on every write slows down the access of I-structures in comparison with the simple von Neumann model and a language based on it, in which the programmer has some control over storage.

Basically, two observations can be made:

(1) Instead of sending data in only one direction—from place of creation to place of consumption—a request (address in the von Neumann model) must be sent in one direction, with the I-structure value (data) returning in the other direction. This unnecessarily increases traffic through the system.

(2) Since memory is allocated before it is used, the problem of optimally distributing I-structures over many processors to minimize traffic through the networks has been introduced. This problem, well known as the *memory contention problem*, has plagued designers of vector machines and multiprocessors for years.

In summary, the proposed I-structures, although expected to solve data storage and access problems more efficiently, are a small step back toward the von Neumann model.

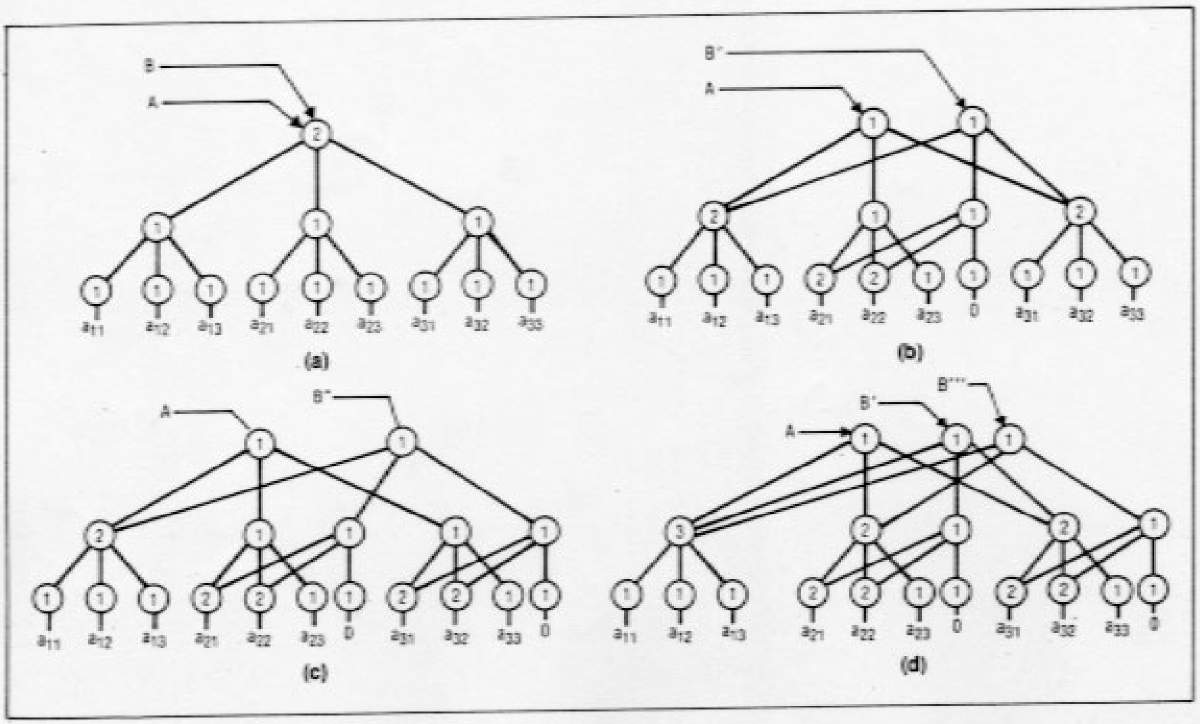


Figure 4. Storage scheme for a 3 × 3 array.

Data flow languages

The success of any computer, data flow or otherwise, depends on the quality of its programming languages. Data flow machines demand high-level languages, since graphs, their machine language, are not an appropriate programming medium; they are error-prone and hard to manipulate.

Three high-level language classes have been considered by data flow researchers. The first is the *imperative* class. For instance, the Texas Instruments group considered the use of a modified ASC Fortran compiler for their data flow machine.[13] Compiler techniques for the translation of imperative high-level languages into data flow graphic languages have also been studied at Iowa State University.[14] The second is the *functional* class. By functional languages, we mean those resembling pure Lisp, which is based on Church's lambda calculus, and Backus' FP, which is based on Curry's combinatory logic. This second class is now being studied in a data flow context at the University of Utah.[15]

The third class—our focus here—consists of the so-called *data flow languages*, which are designed with data flow machines in mind. The most notable examples are Id,[9,12] LAU,[5] and Val.[16] The syntax of these languages is essentially that of imperative languages. For example, all data flow languages include IF and LOOP statements. On the other hand, their semantics are basically that of functional languages.

Below, focus on the two characteristics that set data flow languages apart: the functional semantics of the language and the implicit expression of parallelism. Data flow languages have many other characteristics, which are not unique. For example, the freedom from side effects and the locality of effects have been mentioned as being of paramount importance,[17] and we agree. However, some imperative languages possess these characteristics.

Functional semantics. A consequence of this first characteristic is that in data flow languages variables stand for values and not for memory locations. Imperative languages like Fortran, PL/I, and Pascal allow programmers to be aware of and have some control over the primary memory allocation for both programs and data. Thus, in PL/I we can classify variables as static or dynamic, and memory can be explicitly requested and freed.

In data flow and functional languages, on the other hand, programmers deal only with values. These languages do not allow the explicit control of memory allocation, relying instead on mechanisms like garbage collection to keep memory utilization at a reasonable level.

Functional semantics offers parallel processing the advantage of a simplified translation process. Thus, data flow languages are free of side effects. This makes it possible to translate subroutines separately, without unnecessarily constraining parallelism. Again, freedom from side effects is not unique to data flow languages; imperative languages can also be side-effect free.

Another welcome consequence of functional semantics is the single assignment rule.[18] Thanks to this rule, parallelism is less constrained by anti dependences and output dependences than it is in conventional imperative languages. Consider, for example, the following Fortran program:

```
1    A=D+1
2    B=A+1
3    A=0.
```

It is easy to see that statement 3 cannot be executed until statement 2 fetches *A*; that is, statement 3 is antidependent on statement 2. In a data flow language, the use of *A* in statement 3 is not valid; a different variable must be used in place of *A* to allow execution of statements 1 and 2 to be concurrent with statement 3.

On the other hand, a compiler can very easily rid imperative language programs of antidependences and output dependences by using the simple transformation techniques of renaming and expansion. Renaming, as its name indicates, changes variable names to avoid antidependences and output dependences. This transformation would replace *A* in statement 3 with some other variable. To understand expansion, consider the following Fortran loop:

```
        DO      10  I=1,N
                    X=A(I)+1
   10               B(I)=X**2
```

The different iterations of this loop cannot be executed in parallel, since there is only one memory location corresponding to *X* and because *N* locations would be needed for all iterations to proceed in parallel. After expansion, the scalar variable *X* would be replaced by a vector of *N* elements, and the occurrences of *X* would be replaced by *X*(*I*). This would allow parallelism, at the expense of using more memory. We have found these techniques, as implemented in the Parafrase system,[2] to be successful almost all the time.

Some data flow researchers have been unaware of this fact. This has led them to believe that functional semantics makes a big difference in terms of efficient parallel object code, but this does not seem to be true.

Consider the following quotation from Arvind.[19]

A straightforward Fortran program would do this in the following way.

```
C   X IS AN ARRAY OF N+2 ELEMENTS
C   X(1) AND X(N+2) REMAIN CONSTANT
    N1 = N+1
    DO  20 K = 1, KMAX
    DO  10 I = 2, N1
    Y(I) = (X(I-1) + (X(I) + X(I+1))/3
10  CONTINUE
    DO  15 I = 2, N1
    X(I) = Y(I)
15  CONTINUE
20  CONTINUE                                  (1)
```

A compiler can easily generate good code for a multiple processor machine from the above program. Even if a programmer is clever, and avoids copying array Y into X by switching back and forth between X and Y, a vectorizing compiler will be able to deal with it effectively. However, if array X is large, and a programmer decides to avoid using another array Y altogether, the following program may result:

```
        N1  =  N+1
        DO    20 K  = 1, KMAX
        T1  =  X(1)
        T2  =  X(2)
        DO    10 I  = 2, N1
        X(I)= , (T1 + T2 + X(I + 1))/3.
        T1  =  T2
        T2  =  X(I + 1)
   10   CONTINUE
   20   CONTINUE                                        (2)
```

It would be extremely difficult for a compiler to detect a transformation in which all the elements of array X are relaxed simultaneously.

When the last Fortran program is transformed by Parafrase, the simple expansion technique leads to the following program. It can be effectively executed in parallel (loops 2, 3, and 4 are detected by Parafrase as vector operations).

```
        N1  =  N+1
        DO   1    I  = 1, KMAX
             T1(I)  =  X(1)
             T2(I)  =  X(2)
             DO  2   J  = 1, N1
  2                  T2(J + 1)  =  X(J + 2)
             DO  3   J  = 1, N1
  3                  T1(J + 1)  =  T2(J)
             DO  4   J  = 1, N1
                     X(J + 1)  =  (T1(J) + T2(J) + X(J + 2))/3
  4     CONTINUE
```

The functional semantics might have advantages besides those related to parallel processing. For example, data flow languages might help produce programs that are easier to verify and understand than those in imperative languages. But so far, no scientific evidence has been produced to either confirm or deny such advantages.

Our main objection to functional semantics is that it denies the programmer direct control of memory allocation. Thus, the success of data flow languages depends on how efficiently garbage collection can be implemented and on the specific compiler algorithms used to control memory allocation.

Implicit parallelism. The second characteristic is that parallelism is often implicit in data flow languages. Thus, a data flow language compiler must compute the flow dependences and use them to generate parallel machine code. Implicit parallelism is a worthy goal; it can save the programmer tedious, error-prone tasks. However, we would like to make some observations on compiler techniques and on the need for explicit parallelism.

Compiler techniques. The algorithms used by a data flow compiler determine how much implicit parallelism can be exploited. Therefore, implicit parallelism and compilers must be discussed together. The data flow literature discusses two compiler techniques: flow dependence computation and loop unraveling.[9] These techniques must be developed further if data flow compilers are to successfully exploit implicit parallelism.

Flow dependence is computed by using variable names only. It is very important, however, to look at subscripts, as well. Consider, for example, the Fortran program in Figure 5a. A compiler that ignores the subscripts will not detect the parallelism in this program. Furthermore, the application of loop unraveling when the target machine is a data flow multiprocessor requires some study. Now consider the Fortran program in Figure 5b. If the different iterations of the inner loop are distributed across the data flow processors, a speed-up on the order of N could be obtained. However, if the distribution is done on the basis of the outer loop, and M is much smaller than the number of processors, the speed-up will be substantially smaller.

There are other important techniques that are not discussed in the data flow literature. These include techniques for handling memory allocation and deallocation for code and data (see "functional semantics," above) and techniques that define the storage layout of arrays when the target machine is a multiprocessor.

Explicit vs. implicit parallelism. Implicit parallelism is not sufficient for a powerful programming language, for at least two reasons. The first is the spurious flow dependences mentioned above, and the second is the need to express in summary form the parallel evaluation of recurrences.

Spurious flow dependences are due to the limitations of the compiler. Some can be removed by improving the compiler algorithms; others might be impossible to remove. The discussion of Figure 5a, above, provides an example of how to remove limitations by improving the compiler algorithm. An example in which the limitations cannot be removed is shown in Figure 5c. The Fortran program in this figure is the same as the one in Figure 5a, except that 1 is replaced by $W(K)$. Since $W(K)$ is not known at compile time, it is not possible to determine how or even if this program can be executed in parallel.

Parallel execution is possible in cases like the program in Figure 5c, but only through explicit parallelism. The programmer might know that $W(K)$ is always less than some small value and therefore know that the wavefront algorithm[20] can be applied successfully. In the LAU language and in Val, the programmer could handle this by using the FORALL construct or the EXPAND constructs. However, it is not possible to handle this problem in Id, which has no form of explicit parallelism.

```
(a)     DO  11   I=1, N
             DO  11   J=1, N
                      A(I,J)=A(I-1,J)+A(I,J-1)

(b)     DO  12   I=1, M
             DO  12   J=1, N
                      A(I,J)=A(I-1,J)+1

(c)     DO  13   I=1, N
             DO  13   J=1, N
                      A(I,J)=A(I-W(K),J)+A(I,J-W(K))
```

Figure 5. Fortran programs: (a) requiring subscript analysis for parallelism detection; (b) with inner loop parallel and outer loop sequential; (c) with parallelism that cannot be detected by a compiler.

It should be clear from the previous example that data flow languages need better ways to express general forms of parallelism. It is not clear what those constructs must be, and it is not clear that these constructs can be nicely incorporated in a data flow language.

Only the designers of Val have recognized the need to express recurrences. However, they provide only reduction-type recurrences such as sums and minimums. Other types of recurrences arise often enough to require their inclusion in a parallel programming language.[10] Examples include general arithmetic recurrences and boolean recurrences originating from IF statements inside loops.

Comments. It has been claimed that data flow languages have some advantages over imperative languages for parallel processing and programming in general. Functional semantics, however, is not a real advantage, since well-known compiler techniques applied to a good imperative language allow equal exploitation of parallelism. Also, implicit parallelism requires translation techniques as complicated as those used to extract parallelism from imperative languages. In fact, most of the techniques used in Parafrase[2,21] to translate Fortran programs into parallel programs can be used without change in data flow compilers.

Certainly, data flow languages have nice features, such as freedom from side effects, which are very advantageous for the compiler writer and programmer. However, these do not justify the effort required for the introduction of a totally new class of programming languages. Clearly, imperative languages with these characteristics can be designed.

The immensity of introducing a new language class becomes clear when we consider all the work required before data flow languages stand a chance of becoming common tools. This work must start with syntax; data flow languages are verbose. This verbosity might be a consequence of the syntactic similarity between data flow and imperative languages. The language designers, striving to make the semantic difference clear, introduced unnecessary keywords like NEW in Id, and cumbersome expressions like Y[I:X(I)] in Val to denote an array Y with the Ith element replaced by $X(I)$.

Work is also necessary in the area of explicit parallelism. Data flow languages need constructs to specify parallelism in a general form and to specify general forms of recurrences. Finally, the functional semantics could be a source of difficulty, since it implies that memory allocation is not a concern of the programmer.

Conclusion

In all high-speed computer systems, it is important to achieve two goals:

(1) the discovery of as much potential simultaneity as possible in the computations to be performed; and

(2) the delivery at runtime of as much of the potential simultaneity as possible.

We have discussed various aspects of these points and argued that data flow researchers have done little to further our understanding of the first. It would appear that their contributions have been more concentrated on the second point, but there are a number of shortcomings in data flow ideas in this regard. It is possible to design much better machines than those available today—supersystems, in fact—but by following a bottom-up approach, the data flow people have made it difficult to reach their goal. Data flow notions are quite appealing at the scalar level, but array, recurrence, and other high-level operations become difficult to manage.

In pursuit of the first goal, data flow researchers have introduced the concept of value instead of location into high-level languages. In principle, this was a praiseworthy move. From the compiler point of view, however, there is little improvement over imperative languages. Explicit parallelism and nontrivial compiler techniques are still needed, mostly because of array variables. I-structures represent an attempt to free data flow languages from these two concerns.[12] They essentially allow the flow dependences between array element operations to be automatically satisfied at runtime. It is unlikely, however, that such a mechanism will efficiently solve many of the problems associated with flow dependence between arrays.

We question whether programming language design, as practiced by data flow researchers, is germane to the task of high-speed computer design. We are not prepared to make a pronouncement on programming languages for parallel processing; both applicative and imperative languages have advantages and drawbacks. We do, however, have some questions. First, are data flow languages marketable? To date, the high-speed computer market has been dominated by conservatism and software compatibility. Can data flow languages, as currently proposed, overcome this conservatism? Second, will data flow languages enhance programmer productivity? (The emphasis in imperative programming language design has also been toward increasing programmer productivity.) Although data flow researchers have made some claims to this effect, they remain, to our knowledge, unsubstantiated.

An alternative approach. A much better approach to successful high-speed machine design begins by acknowledging that the programming interface to a high-speed machine requires more latitude than is allowed by current data flow architectures. The following alternative incorporates that latitude. We define *compound functions* with the following properties:

- They represent computational tasks for which good speed-up can be achieved (in most cases) by using multiple processors.
- The compound operations that implement them allow simple control of a substantial amount of hardware in parallel.
- Fast compiler algorithms for deriving them from programs can be written in ordinary sequential programming languages.

Six such compound functions are discussed in Gajski et al.[22]: array operations, linear recurrences, FORALL loops, pipeline loops, blocks of assignment statements, and compound conditional expressions.

We can view a program as a dependence graph connecting compound function nodes. A function dispatch unit must schedule the execution of the compound function nodes. Since the times required by the nodes can be determined at runtime, the function dispatch unit might be considered a data flow machine. We call this a *dependence-driven computation* because several types of data and control dependence are used in determining the execution sequence.

As we return to the second goal of high-speed computer systems—the delivery of simultaneity at runtime—our criticisms of data flow processing should be put in perspective. High-speed computer architecture, in general, has many flaws and weaknesses: Pipeline processors often suffer from long start-up times, and parallel or multiprocessors can waste processor cycles because of mismatches between machine size and problem size. It is very difficult to design a multipurpose machine that is well-matched to a wide range of computations.

The scope of the problem is such that an appeal to engineering intuition should be made at this point. Such an appeal yields three observations:

(1) Dependences should be attacked on all fronts, subject to system design constraints.

(2) Designers should be guided by previously successful designs, when such designs are consistent with the overall constraints.

(3) Deterministic analysis and system operation should be favored over probabilistic analysis and system operation.

With regard to point 1, some data flow researchers have ignored explicit parallelism, and most have considered compiling techniques only superficially. With regard to point 2, data flow researchers often claim that an entirely new approach to high-speed computation is needed. The frequent occurrence of such constructs as array and recurrence operations, however, justifies the exploitation of well-known designs for conflict-free memory access and centrally synchronized global instructions. Adherence to point 3 can guarantee rather than maximize the likelihood of good system performance. In regard to all these aspects, data flow researchers tend not to exploit the global regularity in the problem because the focus is on a small granularity.

Summary of arguments. The following is a brief summary of the arguments against the data flow approach with respect to the principal architectural components in a high-speed computer system.

First, consider main memory array access, which is by far the biggest bandwidth load in many computations. Well-known methods can achieve conflict-free array access[23]; they have been demonstrated in the Burroughs BSP.[24] While data flow people claim to be trying to eliminate the von Neumann bottleneck[25] between CPU and memory, they have created several new bottlenecks of their own. Array access conflicts arise due to asynchrony, shared data cannot be accessed in parallel, unnecessary memory accesses can arise, and tree-like storage of arrays can lead to multiple accesses per array element. Furthermore, data flow programs tend to waste memory space for programs and arrays.

In the area of interconnection networks, data flow machines with nontrivial parallelism (only four-processor machines have been built) will have the same types of problems found in other architectures. No interesting new results in this area have come from data flow researchers; the problem remains an important one.

With respect to processing speeds, data flow architectures seem to inherently deliver less than maximum speedups. Control unit pipelining and instruction look-ahead cannot be exploited to the degree they are in other architectures. Furthermore, since data paths contain very long pipelines, data flow machines suffer from the same long pipeline-filling problems as other pipelined processors, and one must settle for less than maximum speedup. Thus, the performance is weak for programs with low parallelism.

Several practical aspects of data flow machines are worrisome. To date, no one has proposed a way to handle input/output operations, although they seem to be solvable, and debugging data flow programs could be difficult. We have already remarked on the questionable marketability of a data flow processor. As far as we know, there is no difference between the ability to implement a highly parallel data flow processor (with its global arbitration and distribution networks) using present VLSI technology, and the ability to implement a more conventional machine. Finally, there has been no discussion of the diagnosability and maintainability of data flow machines. These could be difficult areas for machines without program counters or deterministic behavior.

We have tried to level pointed criticism directly at data flow principles or at least at a majority of data flow systems. Our task was complicated by several factors. The design of a computing system from language to machine covers a lot of ground, and some researchers gloss over some aspects of the problem. Several groups have independently interpreted data flow principles with different design goals, and these goals are not always spelled out. Finally, a paper design is rarely as good as a practical implementation. Hence, we have had a difficult time discerning exactly what the data flow principles are.

Although we have attempted to point out weaknesses, we should add that data flow does have a good deal of potential. In small-scale parallel systems, data flow principles have been successfully demonstrated. When simultaneity is low, irregular, and runtime-dependent, data flow might be the architecture of choice. In very large-scale parallel systems, data flow principles still show some potential for high-level control. When several compound functions are to be executed in parallel, data flow offers some software engineering benefits, such as elimination of side effects.

It is in medium-scale parallel systems that data flow has little chance of success. Pipelined, parallel, and multiprocessor systems are all effective in this range. For data flow processing to become established here, its inherent inefficiencies must be overcome.

Most data flow researchers are engaged at too low a level of abstraction in dealing with dependence graphs and their relations to machines. They have placed much importance on language design issues that are not always inherently tied to their architecture. While they sometimes imply a radically new approach to high-speed computation, they are plagued by its standard problems. ■

Acknowledgment

We are in debt to Arvind, who provided help in many ways during the writing of this article. Furthermore, we want to acknowledge Burton J. Smith and the anonymous referees whose constructive comments greatly improved the article. Finally, we thank Vivian Alsip for the high-quality job of typing and retyping the manuscript.

This work was supported in part by the National Science Foundation under Grants US NSF MCS76-81686 and MCS80-01561 and the US Department of Energy under Grant US DOE DE-AC02-81ER10822.

References

1. D. J. Kuck, *The Structure of Computers and Computations*, Vol. I, John Wiley & Sons, New York, 1978.
2. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symp Principles Programming Languages*, Jan. 1981, pp. 207-218.
3. J. E. Thornton, *Design of a Computer, The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill., 1970.
4. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, Vol. 11, No. 1, Jan. 1967, pp. 25-33.
5. D. Conte, N. Hifdi, and J. C. Syre, "The Data Driven LAU Multiprocessor System: Results and Perspectives," *Proc. IFIP Congress*, 1980.
6. J. B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science*, Vol. 19, Springer-Verlag, 1974, pp. 362-376.
7. J. B. Dennis, "Data Flow Supercomputers," *Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.
8. I. Watson and J. Gurd, "A Practical Data Flow Computer," *Computer*, this issue.
9. Arvind, K. P. Gostelow, and W. E. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Dept. of Information and Computer Science Report TR 114a, University of California, Irvine, Dec. 1978.
10. D. J. Kuck, "Parallel Processing of Ordinary Programs," in *Advances in Computers*, Vol. 15, M. Rubinoff and M. C. Yovits, eds., Academic Press, New York, 1976, pp. 119-179.
11. Arvind, V. Kathail, and K. Pingali, *A Data Flow Architecture with Tagged Tokens*, Laboratory for Computer Science, Technical Memo 174, MIT, Cambridge, Mass., Sept. 1980.
12. Arvind and R. H. Thomas, *I-Structures: An Efficient Data Type for Functional Languages*, Laboratory for Computer Science, Technical Memo 178, MIT, Cambridge, Mass., Sept. 1980.
13. J. C. Jensen, "Basic Program Representation in the Texas Instruments Data Flow Test Bed Compiler," unpublished memo, Texas Instruments, Inc., Jan. 1980.
14. S. J. Allan and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language", *Proc. Int'l Conf. Parallel Processing*, Aug. 1979, pp. 26-34.
15. R. M. Keller, B. Jayaraman, D. Rose, and G. Lindstrom, *FGL Programmer's Guide*, Dept. of Computer Science AMPS Technical Memo 1, University of Utah, Salt Lake City, Utah, July 1980.
16. W. B. Ackerman and J. B. Dennis, *VAL—A Value-Oriented Algorithmic Language, Preliminary Reference Manual*, Laboratory for Computer Science Technical Report 218, MIT, Cambridge, Mass., June 1979.
17. W. B. Ackerman, "Data Flow Languages," *Computer*, this issue.
18. L. G. Tesler and H. J. Enea, "A Language Design for Concurrent Processes," *AFIPS Conf. Proc.*, Vol. 32, 1968 SJCC, pp. 403-408.
19. Arvind, "Decomposing a Program for Multiple Processor Systems," *Proc. Int'l Conf. Parallel Processing*, Aug. 1980, pp. 7-16.
20. R. H. Kuhn, *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees*, PhD thesis, Dept. of Computer Science Report 80-1009, University of Illinois, Urbana-Champaign, Ill., Feb. 1980.
21. D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers*, Vol. C-29, No. 9, Sept. 1980, pp. 763-776.
22. D. D. Gajski, D. J. Kuck, and D. A. Padua, "Dependence Driven Computation," *Proc. Compcon Spring*, Feb. 1981, pp. 168-172.
23. P. P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Trans. Computers*, Vol. C-20, No. 12, Dec. 1971, pp. 1566-1569.

◄ Use order form on p. 136C

24. D. Lawrie and C. Vora, "The Prime Memory System for Array Access," submitted for publication, 1980.

25. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, Vol. 21, No. 8, Aug. 1978, pp. 613-641.

Daniel D. Gajski is an associate professor in the Department of Computer Science at the University of Illinois, Urbana-Champaign. Before joining the university in 1978, he had 10 years of industrial experience in digital circuits, switching systems, supercomputer design, and VLSI structures. His research interests are in computer system design, algorithm design for supercomputers, hardware and silicon compilers, and design automation. He received the Dipl. Ing. and MS degrees in electrical engineering from the University of Zagreb, Yugoslavia, and the PhD in computer and information sciences from the University of Pennsylvania.

David A. Padua is Professor Agregado at the Universidad Simón Bolivar in Venezuela. From 1979 to 1981, he was a visiting assistant professor at the University of Illinois.

He received the degree of Licenciado en Computación from the Universidad Central de Venezuela in 1973 and a PhD in computer science from the University of Illinois in 1979.

David J. Kuck is a professor in the Department of Computer Science at the University of Illinois, Urbana-Champaign. He joined the department in 1965. Currently, his research interests are in the coherent design of hardware and software systems. This includes the development of the Parafrase system, a program transformation facility for array and multiprocessor machines.

Kuck has served as an editor for a number of professional journals; among his publications is *The Structure of Computers and Computations, Vol. 1.* He has also consulted with many computer manufacturers and users and is the founder and president of Kuck and Associates, Inc., an architecture and optimizing compiler company.

Kuck received the BSEE degree from the University of Michigan, Ann Arbor, in 1959, and the MS and PhD degrees from Northwestern University in 1960 and 1963.

Robert H. Kuhn is an assistant professor at Northwestern University in the Department of Electrical Engineering and Computer Science. His interests are in machine organization and VLSI design. He is currently on the editorial board of *Computer* and is a member of the ACM and IEEE.

Kuhn obtained his PhD in computer science at the University of Illinois in 1980 and received a master's degree in computer science from the University of Connecticut in 1976. At Illinois, he worked with D. J. Kuck on vectorizing compilers for pipelined machines and parallel computer organization.

Use order form on p. 136C ►