LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Third Opinion on Data Flow Machines and Languages

Computation Structures Group Memo 241
10 October 1984

Richard Mark Soley

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661 and in part through various grants from the International Machines Corporation. The author is supported by a fellowship from the National Science Foundation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Introduction

In the February, 1982 issue of the IEEE Computer Society magazine *Computer,* authors D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn discussed at length what they believe are the failings of the dataflow model of parallel computation.[1*] The authors discussed the general bankruptcy they find in the dataflow *language* technology in existence. They concluded that "Most data flow researchers are engaged at too low a level of abstraction..." and that "While they sometimes imply a radically new approach to high-speed computation, they are plagued by its standard problems." Their criticisms led them to believe that current dataflow technology does not stand up to other parallel computation models.

The intent of this article is to dispute this claim. A number of misconceptions, missing data, and errors clouded the reasoning of the aforementioned paper, guiding it to what we consider its incorrect conclusions. We will examine the specific criticisms of that paper, particularly in the light of the dataflow model that we are most familiar with, that of Arvind.[3] The tagged token architecture of that work and the dataflow language *Id*[4] will be used to highlight some of the arguments. We hope to present the promise of the dataflow work currently under way.

The Dataflow Solution to High-Speed Computing

Since the inception of the von Neumann flow-of-control design computer, single-processor design has pervaded all computational structures. Even multiprocessor designs have suffered the ill effects of the *von Neumann bottleneck,* the problem of limited access between processor (or *processors*) and memory.[2, 4, 5] Some modern supercomputers attempt to overcome this problem by means of extensive pipelining within a single processor;[6] though this avoids any problem of memory contention among multiple processors, it introduces either the need to re-code applications in a specially tuned pipelined/vectorized language, or to use high-powered program compilation techniques.[7]

---

* In the dataflow model, asynchrony and functionality form the key to a highly parallel computational model in which programs can be run in a parallel fashion without programmer specification of parallelism.[2]

With the advent of Very Large Scale Integrated circuitry, however, the multiple processor approach becomes even more attractive. The ability to replicate hundreds, or even thousands, of relatively complicated processors on a single chip[8] gives us new impetus for solving the problems of multiprocessors. The dataflow model, in particular the tagged token work of Arvind[3] presented here, solves these problems by using a concept of computation free of the current control-flow model; dataflow processors are controlled only by the flow of *data* in a program. We will present the specifics of the dataflow model as explanations and answers to the notes put forward by Gajski et. al.

Approach Overview

Gajski et. al. began their article with a rough comparative characterization of the two approaches in question, these being (1) the dataflow model and (2) the imperative language program analyzer *Parafrase*[7], a powerful global data flow analysis and transformation system. An annotated redrawing of their figure appears here. In their drawing, Gajski et. al. hoped to show that the compilation of ordinary language programs and the compilation of dataflow programs were quite similar. In fact, the differences inherent in such translations are amazingly great. The previous paper mentioned that "...researchers hope [dataflow language programs] can be easily compiled into a dependence graph." This dependence graph, more commonly referred to as a *dataflow graph,* is the basic language of a dataflow machine, defining a program's structure in terms of the data dependencies between operations of the program. Compilers for exactly the problem of generating dataflow graphs have been written, and they are quite simple; a dataflow language such as Id implies its dependence graph much as a FORTRAN program implies its memory requirements. In addition, once this dependence graph has been arrived at, code generation for a dataflow architecture, while not simple, is just as easy as code generation for any other machine, as parallelism is managed at *run* time, not compile time.

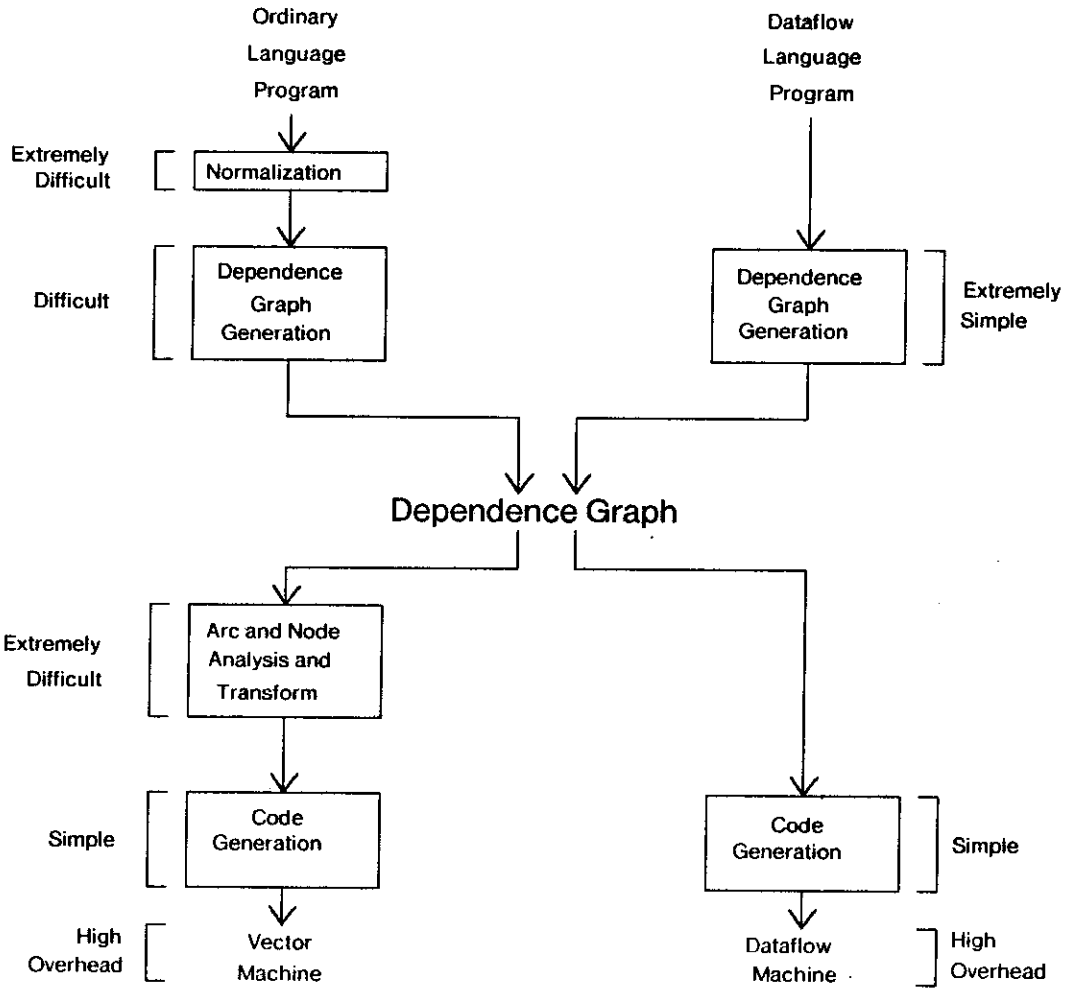


Figure 1

Comparison of Translation Methods

Gajski et. al. noted that "Ordinary languages can also be used [in the dataflow scheme]." However, as in their own approach, dataflow graph generation from an ordinary language program, such as one in FORTRAN, is extremely difficult due to the implied memory model. Global dataflow analysis or interprocedural dataflow analysis is prohibitively expensive on large programs (e.g., FORTRAN programs of greater than 1000 lines). We find it hard to believe that anyone could consider the program normalization and dependence graph generation stages of the previous paper (see figure 1) as a simple problem.[7] The approach of Gajski et. al. also included an arc and node analysis and transformation stage; this is a rather complex procedure which is totally unnecessary in the dataflow model.[9]

It is interesting to note that many of the generic arguments opposing the dataflow architectural point of view specify methods of program analysis within the von Neumann model for increasing the speed of "ordinary" language programs (for instance, global dataflow analysis). These arguments generally note that such principles and algorithms are not used by current dataflow researchers. It is often the case that such methods are in fact quite easily used within the dataflow scheme to provide even more speed gain. For instance, array index analysis, while not in great usage by dataflow machine architects, will certainly be useful in future implementations. That it is not in *current* use does not imply an inherent fault in the principles of dataflow architecture and programming. Gajski et. al. seem to be missing this point when they label some data flow researchers as "unaware"[1] of the powerful methods of programs such as *Parafrase;* in fact, dataflow researchers such as ourselves welcome and are considering the use of source manipulation methods like those presented by Kuck[7] as a method of extracting an extra dimension of parallelism from dataflow programs.

Dataflow Principles

Gajski et. al. then discussed the basic principles of dataflow operation, noting the problems of re-entering looped portions of dataflow graphs. We include a short overview of the dataflow model of computation here.

In an operational view of dataflow graphs, data flows along "wires" connecting operations, which are *enabled* for execution when all of the necessary input data are available at input data lines. After being enabled the operation is executed, resulting in an output value emitted from the output data line (or one of several output data lines)

connected to the operation. An example of a dataflow graph can be seen in Figure 2, which computes the result of adding three and four.

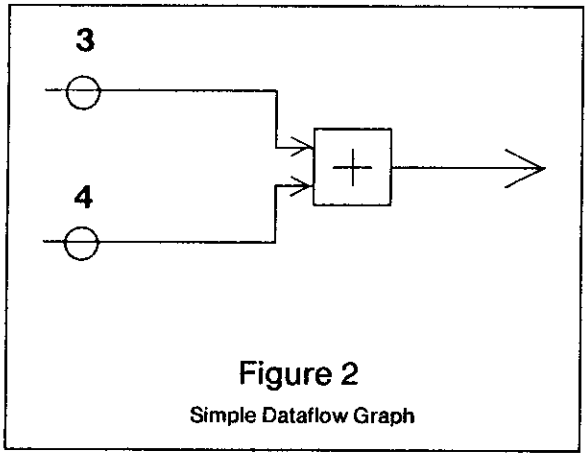


Figure 2
Simple Dataflow Graph

This model has one problem. Simply stated, the (earlier) operations which output the values three and four might have computed more output values before the "plus" operation had had time to compute. The question then becomes one of which values are "related to each other" for a given computation. This problem is further aggravated by the presence of loops in dataflow languages, which are represented by recursive graph structures such as that in Figure 3, which adds up the integers between one and ten inclusive.

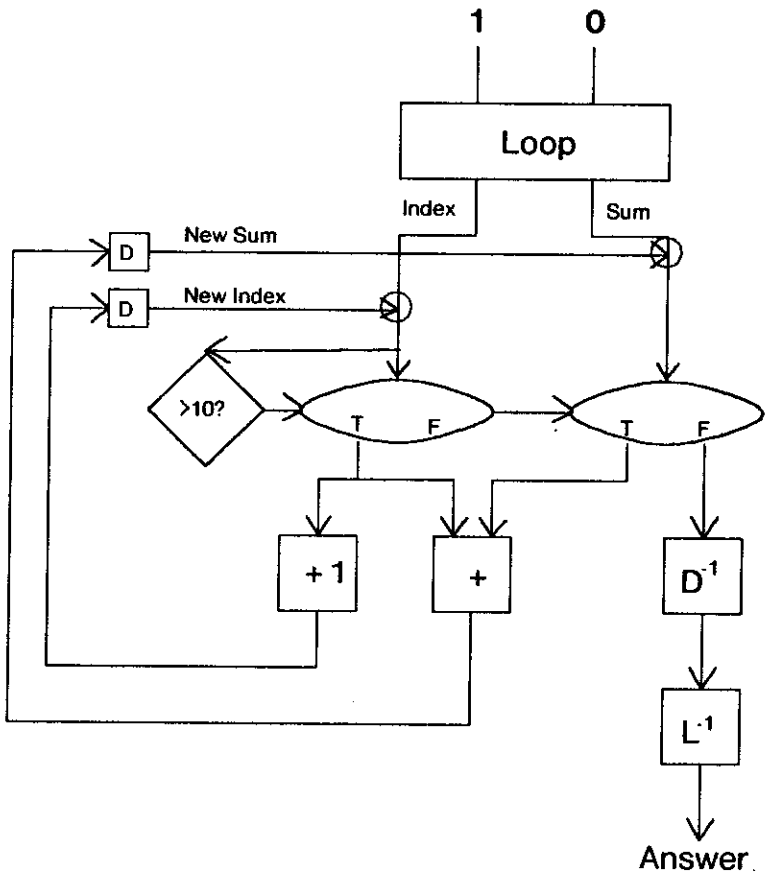


Figure 3

Dataflow Loop

Gajski et. al. went through the five best candidates for the resolution of this problem, pointing out the new problems introduced by each approach. This analysis led them to conclude that no acceptable solution to the re-entrant graph problem exists. We must agree with them on four of their five evaluations; however, we find the tagged-token approach a viable solution. The five possibilities are reiterated and commented upon here:

(1) *The use of a re-entrant graph is prohibited.* This only solves the looping problem, and in any case reduces a dataflow machine to an almost strict von Neumann approach. As such, this approach is not used by any dataflow projects.

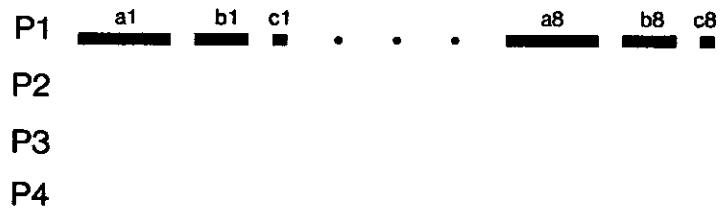
(2) *The use of a re-entrant graph is allowed, but an iteration is not allowed to start before the previous one has finished.* As they note, this solution requires more work for *less* parallelism.

(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.* As Gajski et. al. noted, this approach is used by Dennis[10]. Although it resolves all of the above problems, it reduces the amount of parallelism realized, possibly increases processor idle time, and increases communication and static memory overhead, since it requires a data acknowledgment scheme.

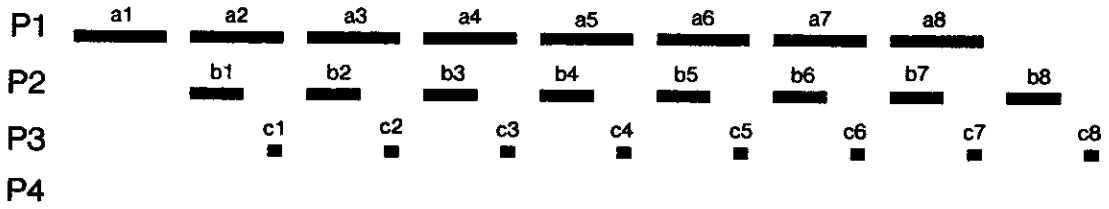
(4) *The tokens are queued on arcs in order of their arrival.* Again, this scheme solves the aforementioned problems. But it may increase processor idle time (and therefore decrease parallelism of computation) by forcing an operation to wait until an "earlier" computation (by program analysis) on that operation is complete.

(5) *The tokens are assumed to carry their index and iteration level as a label.* In the terminology of Arvind[3], this label is the *tag* of the *tagged token*. This scheme exploits maximum parallelism. However, tagging requires extra bits on every token, and additional time to compute tags for newly generated tokens. But it should be noted that the extra bits used for dynamic tagging save the large amount of static preallocated token storage necessitated by scheme (3) above. In addition, the computation for the new tag for a token is simple, and can be performed concurrently with the normal ALU operation that produces the token. Tag sizes may also be kept bounded by several "tricks" such as the *logical domains* and reusable *colors* suggested by Arvind.[3]

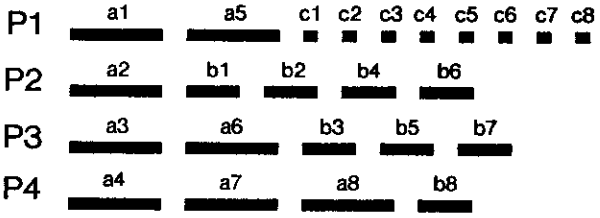
After the above outline, Gajski et. al. calculated some total time estimates for a simple program running on four parallel processors, using each of these approaches. In addition, they showed possible results of running the same program under a vector architecture, after the use of a vectorizing compiler. Their figure is repeated in figure 4.



(a) von Neumann. Time = 48, 25% utilization



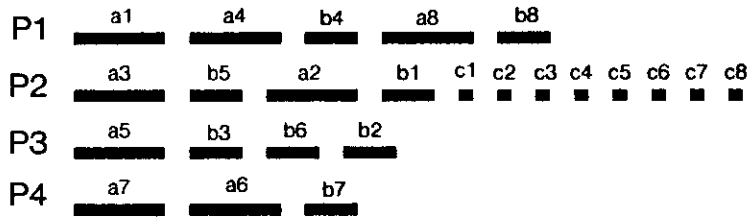
(b) LAU Dataflow. Time = 27, 44% utilization



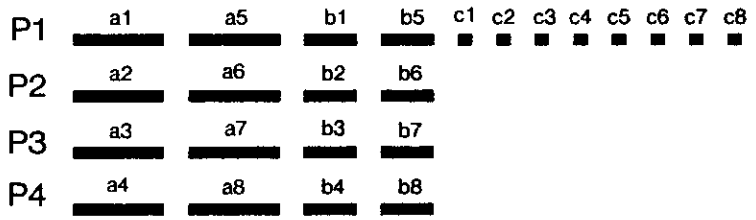
(c) Best Dataflow. Time = 14, 86% utilization

Figure 4

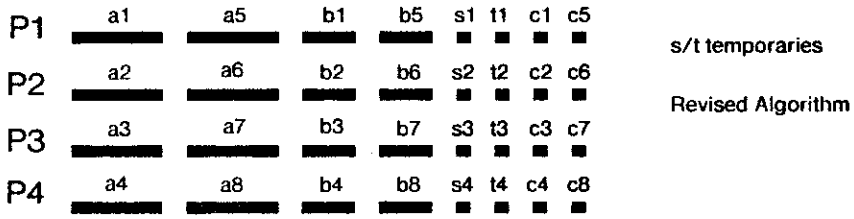
Scheduling and Timing



(d) Worst Dataflow. Time = 18, 67% utilization



(e) "Fair" Vectorized Code. Time = 18, 67% utilization



(f) "Good" Vectorized Code. Time = 14, 86% utilization

Figure 4, continued

Scheduling and Timing

Each block in figure 4 represents a computation; the longer the block, the longer the amount of processor time needed for that computation. These computations correspond to the program proposed by Gajski et. al., which is shown below. It included eight *a* operations, eight *b* operations, and eight *c* operations, which take (respectively) three, two, and one time units to complete on some hypothetical ALU. Each *b* operation is data-dependent on a corresponding *a* operation, and each *c* operation is data-dependent on a corresponding *b* operation *as well as all previous c operations*. This last, of course, is the most problematic from the viewpoint of scheduling this program on any computer, be it dataflow, vectorized, or "standard" von Neumann architecture. Their program is repeated here:

input d, e, f;
$c_0 = 0$;
for i from 1 to 8 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;

All of the patterns in figure 4 show possible computation schedules for this program when running in parallel on four processors P_1, P_2, P_3, and P_4, under different scheduling and architecture schemas. As can be seen in the figure, the timings in (a) and (b), which correspond to a von Neumann architecture and a dataflow architecture corresponding to approach (2) above are, as expected, quite high. The next two timings correspond, essentially, to random process scheduling on an Arvind style dataflow machine, as typified by approach (5) above. The last two timings represent what might be expected of code executing on vectorized machines generated by a "fair" and a "good" vectorizing compiler, respectively. As can be seen, timings (c) and (f) are the best, taking 14 time units, with a processor utilization coefficient of 12/14 = 86%. It is noted that under a random scheduling pattern, a dataflow machine may actually be left in pattern (d), with a total time commitment of 18 units, and a utilization of only 12/18 = 67%.

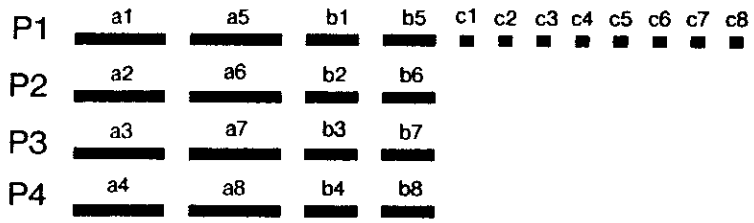
Some points are missing in this discussion, however. First, Gajski et. al. claimed that a mere "good" vectorizing compiler could come up with the required fourteen time unit code. At closer inspection, however, we find that the actual program *algorithm* has been changed in order to only *match* the best performance of a dataflow machine! The new algorithm introduces temporaries which reflect a complete change in the program's methodology, as well as assuming the addition operator to be associative. Their new program, ostensibly the output of only a "good" vectorizing compiler, looks like this:

input d, e, f;
$c_0 = 0;$
for i from 1 to 8 begin
  $a_i = d_i / e_i;$
  $b_i = a_i * f_i;$
end;

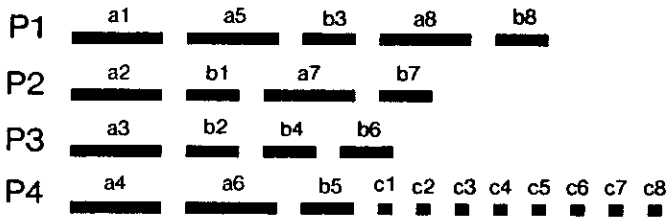
$s_1 = b_1 + b_2;$	$s_2 = b_3 + b_4;$	$s_3 = b_5 + b_6;$	$s_4 = b_7 + b_8;$
$t_1 = b_3 + s_1;$	$t_2 = s_1 + s_2;$	$t_3 = b_4 + s_3;$	$t_4 = s_3 + s_4;$
$c_1 = b_1 + c_0;$	$c_2 = s_1 + c_0;$	$c_3 = t_1 + c_0;$	$c_4 = t_2 + c_0;$
$c_5 = b_5 + c_4;$	$c_6 = s_3 + c_4;$	$c_7 = t_3 + c_4;$	$c_8 = t_4 + c_4;$

output a, b, c;

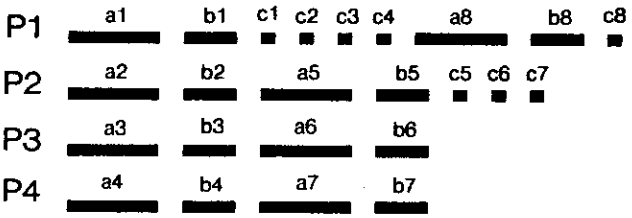
In addition, the 18 unit pattern (d) is a worst case scheduling pattern; even a simple FIFO operation queue can match that. The outcome of a simple FIFO scheduling algorithm is displayed in figure 5 (a). Even figure 4 (d) can be simply reworked by moving the column of c operations, leaving a total time pattern of 16 units, with a utilization of 12/16 = 75%, a noble percentage. This reworked display is shown in figure 5 (b). In addition, two other scheduling algorithms, designed "on the fly" by us and named *Lookahead* and *Lookahead with Priority,* result in the greatly improved scheduling outcomes displayed in figure 5 (c) and (d), with respective processor utilizations of 80% and 86%. Even without recourse to complicated and domain-dependent scheduling algorithms, we can generate a pattern such as figure 5 (e), with a utilization of 75%, without any central scheduling algorithm at all. This is the pattern that would be generated by the tagged-token machine using its current non-centralized activity "scheduling" algorithm with four *subdomains* and a mapping constant of two.[4]



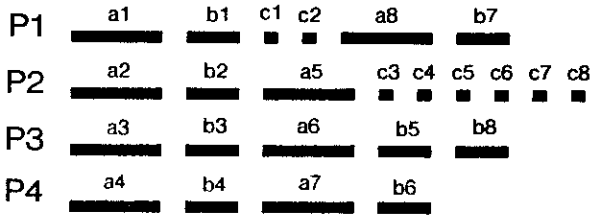
(a) FIFO Dataflow. Time = 18, 18% utilization



(b) Dataflow. Time = 16, 75% utilization



(c) Lookahead Dataflow. Time = 15, 80% utilization



(d) Lookahead/Priority Dataflow. Time = 14, 86% utilization

Figure 5
Better Scheduling Algorithms

Of course, these scheduling algorithms (which are cursorily outlined in figure 6) were designed for the problem at hand, and reflect the scheduling problems of the program example. In other words, They define processing element activity scheduling schemes that seem to optimize processor usage for this particular example. Therefore, they result in timings only slightly more than, and even equal to (in the case of Lookahead with Priority), that claimed by Gajski et. al. for a "good" vectorized program. This brings us to three important points:

(1) *Not much research has gone into dataflow scheduling algorithms.* No one ever claimed that random operation scheduling would match timings achievable on powerful vector machines; more research needs to be, and will be, done. It is possible that simple, non-random algorithms might go a long way to solving this problem; if not, specialized methods such as marking tokens on the critical path of a dataflow graph might be called for. The interested reader is directed to new work by Ho and Irani.[11]

(2) *There is no reason that a dataflow machine cannot choose a scheduling algorithm on a per-program basis.* A compiler for a dataflow language, or the programmer, can make some simple assumptions to help a dataflow machine choose a suitable scheduling schema. Again, any program analysis that can be used in a non-dataflow scheme, such as static program scheduling at compile time, can be included in a dataflow computing scheme.

(3) *Processor "idle" time on a dataflow architecture does not equal processor idle time on a vector machine.* Vector machines are by nature single-instruction multiple-data in organization; dataflow machines, in contrast, are *multiple-instruction* multiple-data architectures. Blank (or "idle") time blocks sketched for dataflow architectures in figures 4 and 5 need not be really idle, but could be working on different parts of the same program *or even a different program altogether.* Idle time

blocks sketched for vector architectures in figure 4 represent *actually idle hardware.* An interesting approach to filling the idle time of a dataflow machine, presented by Burkowski[12], suggests a multi-user approach to multiprocessing.

```
ADD-TO-QUEUE: (p):
    tail.next = p;
    tail = p;
END;
```

Queue Addition

```
SELECT:
    answer = head;
    head = head.next;
END;
```

FIFO Selection Algorithm

```
SELECT:
    LOOP FOR p IN queue DO
        IF p.projected-time < min
            THEN min = projected-time;
                answer = p;
        END;
END;
```

Lookahead Selection Algorithm

```
SELECT:
    LOOP FOR p IN queue DO
        IF p.projected-time < min
            THEN min = p.projected-time;
                answer = p;
        END;
        IF p.skip >= p.projected-time
            THEN min = -1;
                answer = p;
        END;
        p.skip = p.skip + 1;
END;
```

Lookahead with Priority Selection Algorithm

Note: Projected-time components are in units of scheduling cycles, not milliseconds.

Figure 6

Scheduling Algorithms

Gajski et. al. then continued with a discussion of the problem of data flow computing under a low degree of parallelism. Their contention, echoed by Abe et. al.[13], was that the overhead necessitated by the dataflow computation model would render it less useful while executing a program with less explicit and implicit parallelism (i.e., with a high amount of data dependency). It is important to note, however, that a dataflow machine is capable of exploiting parallelism on *any level.* The following program, which utilizes some function *f* that we assume has no exploitable parallelism, expresses this point:

```
total = 0;
for i from 1 to N begin
      total = total + f (x_i);
end;
output total;
```

This fragment, which simply sums up a mapped set of numbers, is the ultimate nightmare for a pipelined processor. Since such a machine is reduced to computing the mappings $f(x_i)$ sequentially, the total running time of this program on a standard pipelined processor would be $N * t_f + t_{SUM}$ where t_f is the execution time of the function *f*, and t_{SUM} is the time necessary to sum the *N* mapped values. In contrast, on a dataflow architecture, this program would execute in time $t_f + t_{SUM}$, since the *f* mappings may be performed in parallel. This is a good example of a program which seems bound to sequential execution methods, but actually contains enough parallelism for a dataflow machine to exploit.

In addition, even in the rare cases which might actually allow *no* exploitable parallelism in a program, the dataflow machine does not need to waste cycles idling. As was mentioned above, Burkowski's work involving usage of dataflow machine idle time,[13] in a multi-user system points out that "idle" hardware can be used for processing other programs simultaneously; a dataflow architecture, unlike more standard machine architectures has some realistic hope of "keeping all of its pipelines full."

In conclusion, in theory at least, the Arvind dataflow architecture might hold its own against a vector machine approach, contrary to the opinions of Gajski et. al.

Memory Management

Another problem that Gajski et. al. found in the dataflow approach to parallel processing is that the dataflow architecture implicitly and explicitly uses more storage than a corresponding von Neumann or vectorized program. In other words,

(1) A dataflow system, by its very architecture, enforces duplication of data and therefore memory use.

(2) The programmer is given no control over memory management.

Gajski et. al. directed most of their criticism at the major memory users, arrays and structures. They also discuss the problems of I-structures, the structure memory model advanced by Arvind and Thomas.[14] This model semantically presents a non-ordered list of pairs, somewhat reminiscent of Lisp's assoc-lists[15] in place of arrays and the like.

The I-structure model of dataflow machine memory, together with the portion of the dataflow machine named the *I-structure controller,*[16] implement a scheme for non-blocking concurrent access to data. The most important aspect of I-structure storage is the model, "read datum and deliver to processor *X*," the primitive non-blocking non-local read operator on the tagged-token dataflow machine.

In addition to supplying this solution to multiple contention-free memory access, the I-structure model provides a method of cutting down inter-processor (network) traffic. The semantics of functional languages, upon which the tagged token dataflow scheme is based, demand that *no memory* be resident in each dataflow processing activity; all machine *"state"* should be available only along the arcs of the dataflow graph. Although in the physical world this does wipe out the memory contention problems which plague most multi-processor architectures[17], it introduces more inter-processor overhead.

The I-structure solution is to provide, for large arrays of data and other carefully selected related data, array-like storage spread throughout the physical system. Although this scheme reintroduces memory contention, it can save large amounts of network traffic which might be unnecessary. For instance, let us consider the Id program:

```
x ← (initial x ← <>;          ! Initialize x to the empty structure. !
        i ← 1                 ! Initialize i to one. !
    while i < 11 do
        new x [i] ← f (i);    ! On each iteration, set an element of x. !
        new i ← i + 1
    return x);
```

This program calls a function *f* on each iteration, building a structure with what it returns. This structure can be thought of simply as an array of values. On a non-dataflow machine, this array would be a portion of some linear memory; each element would be one of the contiguous cells in this memory.

This model of structure storage would not suffice on a dataflow machine. One of the features of the tagged token dataflow machine is that separate iterations of a loop will be executed concurrently if data dependencies allow. Therefore, the separate processors executing this loop would contend for the central memory in which the structure *x* resided, greatly reducing performance. In the I-structure model, however, the structure *x* is allowed to encompass memory spread over multiple processors; thus, each processor executing an iteration of the loop may store a portion of *x* locally. This solution lessens network overhead *and* memory contention, while increasing effort only at compile time. In addition, I-structure memory solves the problem of attempting to read data before it is created by associating *presence* tag bits with each word of I-structure storage and deferring requests for memory reads until the requested memory address is filled by a write request.

Gajski et. al. found three inherent problems of the I-structure model, which we dispute here:

(1) *The request/data usage model of I-structure memory doubles communication traffic through the system.* This is untrue; in a von Neumann architecture, there is an address request answered by a piece of data returned to the processor. In the I-structure scheme, there is a request and a data return, the same amount of traffic as before. Deferred read requests (for items not yet written) do not increase communication overhead, only write time overhead. Perhaps Gajski et. al. confused the

hardware issue of conventional circuit switching and acknowledgment with the dataflow scheme of packet switching.

(2) *The problem of distributing I-structure storage over the multi-processor is introduced by this scheme.* Although this is true, we refer the reader to reference (3) for discussions of reduction of this problem by methods such as *physical subdomain* problem splitting. Conventional systems face this same problem when attempting to reduce memory contention; this is not a "new problem" introduced by dataflow computation.

(3) *The necessity of garbage collection is unnecessarily introduced into the system by this scheme.* The computing literature is rife with work related to complex schemes reducing the problems of garbage collection by various means. However, we needn't use *any* of these difficult schemes; since I-structures are by definition non-circular in nature, Arvind uses a reference-count storage collection and allocation method, which is well understood and low overhead in nature.[3] Dynamic storage allocation is *necessary* for a general, powerful programming model.[18] In addition, there is no reason that a dataflow machine cannot use preallocation where such schemes are notable at compile or load time. We must also point out the growing realization that languages that do not provide fully automatic memory management (and therefore necessarily a garbage collection scheme) are limited in expressive power.[18]

Data Flow Languages

Gajski et. al. then turned their criticisms toward the languages in use by data flow machine architectures, languages sculpted with the limitations and abilities of these architectures in mind. It is useful here to reiterate the basic properties of data flow languages as they currently exist, as stated by Ackerman:[9]

(1) *Freedom from side effects.* This requirement brings "functional" languages such as FP[19] and pure Lisp[15] into the picture.

(2) *Locality of effect.*

(3) *Data dependencies form the only scheduling constraints.* This property makes a language mirror its related data flow graph.

(4) *The single assignment convention.* As noted by Gajski et. al., this convention is already widely accepted as a programming methodology; it is enforced by data flow languages.

(5) *A different semantic (and therefore syntactic) model for looping.*

It is easy to see how these properties rely on, and are necessitated by, the data flow machine model. Freedom from side effect clearly simplifies the detection of parallelism in a program. Points (3) and (4) make data dependency the heart of operation scheduling, while point (5), a new loop model, is necessitated by points (1) and (4).[9]

In light of the long section of this paper dealing with scheduling data flow operations on a multiprocessor architecture, it is interesting to note that in data flow languages, data dependencies form the *only* scheduling constraints. It is important to note the difference between *constraints* and *optimal performance;* though a dataflow program will run just as *correctly* without any scheduling intelligence (due to this property of dataflow languages), it is possible that a more machine-cognizant schedule of operations might speed dataflow processor usage. This is analogous to the old FORTRAN problem of how to allocate and use a two-dimensional array; although the program will return correct results written in any particular way, if the array is stepped through in proper order (given the interleaving constraints of the machine itself), the program will return results more quickly.

Gajski et. al. used one theme while reviewing the above design issues; in their words, their "main objection to functional semantics is that it denies the programmer direct control of memory allocation." They further noted that "...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."

We find fault in this critique at every possible level. First, giving a programmer "direct control" of memory allocation means nothing in the context of a dataflow computer; since memory itself is partitioned among processors, there is no way for a programmer to specify a "block" wherein he would like to store some cherished variable's value. Data might be needed in widely disparate processors at highly indeterminable times; to "allocate memory" in the von Neumann style is a semantic void in the data flow world. In addition, placing this responsibility in the hands of the programmer has been called "too burdensome and too error-prone for general purpose systems."[18]

Of course, this means that we must rely on some sort of garbage collection scheme to manage memory, since it is highly implausible to give this job to the programmer. As has been mentioned above, however, we needn't resort to complex memory management techniques; the lack of circular data structures in languages like Id[4, 14] allows the use of simple reference count garbage collection schemes.

In addition, several researchers, notably Harrison[20], have been doing great amounts of work in the direction of removing storage overhead needs in functional language situations. Harrison, in his preface, notes "We propose techniques to remove, or at least to reduce, [memory usage] inefficiencies and inadequacies in the implementation of functional languages." Given the above-noted ability to use low-overhead reference count garbage collection algorithms and the work by Harrison, it seems clear that memory usage problems will not be a major overhead factor in functional language implementation. We conclude that the lack of confidence in functional semantics professed by Gajski et. al. is therefore ill founded.

Implicit Parallelism

Gajski et. al. launched just one more major attack; they found fault in data flow languages' reliance on implicit specification of program parallelism. They mention two problems with this reliance: (1) that not enough compiler technology is in use by data flow compilers for unearthing all of the available parallelism of programs, and (2) there are not enough methods for representing explicit, programmer-specified parallelism in data flow languages. As for the first point, it is true in the context of concurrent operations on data structures; more attention could be, and undoubtedly will be, focused on finding more

implicit parallelism by advanced compiler techniques. That all such techniques are not currently found in the data flow literature is not an attack on the method, but on specific implementations. Dataflow compilers are already using the type of subscript analysis advocated by Kuck; as a matter of fact, this analysis is greatly simplified by the functional nature of languages like Id.

The second point, however, is a patent mis-statement about current language technology. In particular, Gajski et. al. noted that Id "...has no form of explicit parallelism." A glance at some of the Id literature belies this idea; the Id **for all** and **for each** constructs support a scheme of explicit parallel computation and composition of functionality almost unmatched in current language technology.[3] Id *streams* and *managers*[21] support a functional method of pipelining operations, as well as controlling parallel use of system resources. In addition, the basic parallelism apparent in a dataflow language, the freedom to concurrently evaluate functional arguments, might in any case obviate any need for explicit programmer-noted parallelism.

Gajski et. al. then went on to state that "...implicit parallelism requires translation techniques as complicated as those used to extract parallelism from imperative languages." This statement is simply incorrect; the basic concurrent argument evaluation structure of dataflow languages, as mentioned above, does not have to be extracted; it is immediately apparent from the surface of the dataflow graph. We acknowledge that there are translation techniques available for imperative languages that may be useful for even better implicit parallelism recognition in dataflow languages; however, the important parallel component of dataflow languages is evident by simple inspection.

Additional Points

The most interesting, and correct, points noted by Gajski et. al. concerned marketability and usability. We address them on a point-by-point basis:

(1) *Does the conservatism of the high-speed processor market leave data flow languages unmarketable?* This is a tough question without market research and timings from actual data flow machines. Nascent dataflow techniques must be tested far more thoroughly before this question can be evaluated. However, it is clear that if dataflow machines are shown to be

scalable (i.e., doubling the number of processors approximately doubles performance), it is reasonable to assume that even the most conservative of high-speed computer users would be willing to switch to dataflow machines and languages.

(2) *Do data flow languages increase programmer productivity?* Most data flow languages approximate extant languages in look, feel, and functionality; although there is scant data on the subject, we believe that such languages are not so great a departure from current languages that they would lower programmer productivity. In fact, given the functional nature of dataflow languages and such languages' insistence on modularity and other good programming practices, it is likely that programmer productivity will greatly *increase.*

(3) *How are input/output and debugging affected by data flow architectures?* Great amounts of research are necessary in this area, especially the problem of debugging a program that is running essentially nondeterministically on many processors, split at the operation (i.e., a very low) level. We find this area to be the most interesting and challenging, though not insurmountable, part of data flow research; some pioneering work in the field has been done by Bauman and Iannucci.[22]

Conclusion

We hope that this paper has presented arguments that clear, or at least soften, the doubts left by Gajski and his colleagues. The data flow multiple computational model, though new, radical, and substantially untested, is a budding and promising field which may some day revolutionize, or at least help to revolutionize, high-speed (and perhaps medium-speed) computing; we hope that its proponents and detractors can join forces to break through the hardware and software limits which hold back computing speeds as we know them. That the classic von Neumann style of computing is out of date is clear, and we find the data flow model a very acceptable and desirable alternative.

Acknowledgments

We wish to thank Isabel Szabó and Poh C. Lim, as well as the anonymous reviewers, for their careful comments and criticism.

References

1. D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, A Second Opinion on Data Flow Machines and Languages, *Computer*, February 1982.

2. T. Agerwala and Arvind, Data Flow Systems, *Computer*, February 1982.

3. Arvind, V. Kathail, and K. Pingali, *A Data Flow Architecture with Tagged Tokens*, Laboratory for Computer Science, Technical Memo 174, MIT, Cambridge, MA, September 1980.

4. 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, December 1978.

5. G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *Proc. 1967 AFIPS Spring Joint Computer Conf.*, March 1967.

6. J. E. Thornton, *Design of a Computer, The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill., 1970.

7. 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*, January 1981.

8. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Co., Reading, Mass. 1980.

9. W. B. Ackerman, Data Flow Languages, *Computer*, February 1982.

10. J. B. Dennis, Data Flow Supercomputers, *Computer*, November 1980.

11. L. Y. Ho and Keki B. Irani, An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment, *Proc. 1983 Int'l. Conf. Parallel Processing*, August 1983.

12. F. J. Burkowski, A Multi-User Data Flow Architecture, *Proc. Eighth Annual Symp. Computer Architecture,* May 1981.

13. S. Abe, R. Hiraoka, Y. Fukunaga, T. Bandoh, K. Hirasawa, and Y. Kawamoto, Preliminary Performance Evaluation of Data Flow Computing, *Proc. Compcon Spring 1982,* February 1982.

14. Arvind and R. H. Thomas, *I-Structures: An Efficient Data Type for Functional Languages,* Laboratory for Computer Science, Technical Memo 178, MIT, Cambridge, MA, September 1980.

15. J. McCarthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, *Communications of the ACM,* April 1960.

16. S. K. Heller and Arvind, Design of a Memory Controller for the MIT Tagged Token Dataflow Machine, *Proc. Int'l. Conf. Computer Design '83,* Portchester New York, November 1983.

17. Arvind and R. A. Iannucci, A Critique of Multiprocessing von Neumann Style, *Proc. Tenth Int'l. Symp. Computer Architecture,* Stockholm, Sweden, June 1983.

18. G. L. Steele, Multiprocessing Compactifying Garbage Collection, *Communications of the ACM,* September 1975.

19. J. Backus, Can Programming be Liberated from the von Neumann style? A Functional Style and Its Algebra of Programs, *Communications of the ACM,* August 1978.

20. P. G. Harrison, Efficient Storage Management for Functional Languages, *The Computer Journal, 25* (2), May 1982.

21. Arvind, J. D. Brock, *Streams and Managers,* Computation Structures Group Memo 217, Laboratory for Computer Science, MIT, Cambridge, MA, June 1982.

22. N. B. Bauman and R. A. Iannucci, A Methodology for Debugging Data Flow Programs, Computation Structures Group Memo 219, Laboratory for Computer Science, MIT, Cambridge, MA, October 1982.