LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures

Computation Structures Group Memo 278
June 1988

Arvind
David E. Culler
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Kattamuri Ekanadham
IBM Research
T.J. Watson Research Center
Hawthorne, New York

To appear in the *Proceedings of CONPAR*, Manchester, England,
September 12-16, 1988.

This research was done at the Massachusetts Institute Technology Laboratory for Computer Science and at the IBM T.J. Watson Research Center. Funding for the Massachusetts Institute Technology project is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Resaerch contract N00014-84-K-0099. This paper is also published as IBM RC 13889.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures[1]

Arvind
David E. Culler
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Kattamuri Ekanadham
IBM Research
T.J. Watson Research Center
Hawthorne, New York

Abstract:

A cost analysis is presented for the dataflow model of execution, in which instructions are executed in an asynchronous manner to take advantage of all the fine grain parallelism in a declarative program. Comparisons are drawn with the execution costs of equivalent imperative programs on traditional sequential and parallel machines. The cost of program execution is measured in terms of the number of instructions executed. It is argued that the complexity of dataflow instructions is comparable to that of traditional load/store architectures and that pipelining is very effective for dataflow instructions; thus, instruction counts provide a meaningful comparison.

It is shown that the total number of instructions executed in fully parallel, asynchronous dataflow execution of programs is 2 to 3 times that of equivalent programs on the IBM 370 and close to that on load/store architectures. The overhead of dataflow execution is identified with parallel conditionals, iterations, and function invocations. This overhead appears to be of the same magnitude as the *basic work* in the program. A comparison is made with selective parallelization of programs for conventional multiprocessors. Finally, a brief analysis of the hardware cost of an architecture which exploits fine grain asynchronous parallelism is offered.

1 Introduction

Parallel execution has been the primary focus of recent research in the arena of high speed computation. Although performance is the driving force, many researchers have recognized that it is essential to abstract the underlying parallel hardware and provide a configuration-independent programming model. The aim of the dataflow project at MIT is to achieve both goals: performance and programmability. Programs are expressed in a declarative language called *Id* , which specifies only the data dependencies between subcomputations, allowing for maximally parallel execution. Execution is performed in a data-driven manner in order to exploit all the potential parallelism in a program. Our studies have demonstrated that the language is powerful enough, and the language tools mature enough, to write "real" applications [6], not just toy programs. Moreover, programs expressed in this form exhibit a great deal of parallelism, even using conventional algorithms [5]. A parallel program invariably performs more work than a sequential one for the same problem due to generation and synchronization of parallel activities. Since dataflow architectures try to exploit all the parallelism in a program, it is important to quantify the overhead incurred by such an approach, in absolute terms and in comparison with alternative approaches.

In order to quantify this overhead, we must first settle upon a cost metric. In our view, the cost of executing a program involves three aspects:

[1] This research was done at the MIT Laboratory for Computer Science and at the IBM T.J. Watson Research Center. Funding for the MIT project is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

1. the number of instructions executed,
2. the time required to execute these instructions, and
3. the complexity of the associated hardware.

The primary focus of this paper is on the first aspect — we present detailed data on the number of instructions executed for a variety of programs on dataflow and conventional machines. These measurements are made by executing the program on a properly instrumented interpreter or, in the case of Cray machines, with hardware monitors. We separate the portion which is *basic work* from that which is overhead, and show that the overhead in dataflow execution is directly attributable to asynchronous parallel execution.

The other two aspects of the cost of program execution are more difficult to quantify. Hardware monitors and interpretation have been used to measure elapsed cycles in sequential machines, primarily RISC machines, but these techniques do not deal well with the asynchronous events that are common in parallel machines due to synchronization, memory contention, etc. Thus, we can provide only qualitative assessments. To place instruction counts on a firm footing, we argue that the instruction set of the dataflow machine is roughly comparable to that of modern load/store architectures — instructions are simple and all take roughly the same number of cycles [22]. Furthermore, at least in concept, dataflow architectures can pipeline instructions more heavily than sequential architectures available today, so it should be possible to dispatch an instruction every cycle. One reason for this is that in dataflow architectures, all memory references and synchronization operations are "split phased", and thus do not block the processor pipeline[8]. In addition, independent "threads" of computation can be interwoven on a cycle by cycle basis. Although we are not ready to predict the absolute performance of a multiprocessor dataflow machine based on instruction counts, given this qualitative assessment of dataflow instruction execution mechanisms and results presented elsewhere[5] on the potential parallelism in programs, we believe that instruction counts are a first-order indicator.

The hardware cost of a novel architecture in an early developmental stage is extremely hard to quantify, and we make only a qualitative assessment here. Because synchronization and generation of parallel activities are integral parts of the basic instruction execution mechanism, dataflow machines are likely to require more hardware to deliver the same MIPS in a single processor. The primary criticism of dataflow architectures has centered on the complexity of the wait-match unit (discussed below)[14]. However, recent advances in compiling have made it possible to replace the content-addressed wait-match store by a directly addressed store[2, 8]. This approach is the basis of the Monsoon dataflow machine, currently under construction at MIT [21].

Aside from architectural differences, we must contend with differences in programming languages. The dataflow programs we consider are written in *Id* , while the conventional versions are in FORTRAN. The value oriented approach of declarative programming, together with features such as higher order functions, infinite objects, and the capability of passing functions and data structures around freely, not only facilitates a higher level of programming but also exposes parallelism implicitly [6]. However, this power brings additional implementation costs[26] due to dynamic storage allocation, automatic storage reclamation, lexical closures, partial evaluation of functions, lazy evaluation, etc. In this paper we do not embark upon an analysis of costs accrued due to differences in programming style *per se*, rather we look at a set of examples which are so easily expressed in either declarative or imperative languages that language issues are of lesser significance, other than in the generation of parallel code.

Regardless of the programming style, optimizations performed by a compiler can have substantial impact on cost of program execution, so we only consider code generated by sophisticated compilers. Most of optimizations performed by the *Id* compiler are present in traditional optimizing compilers; these include common subexpression elimination, constant propagation, inline substitution, fetch elimination, and loop invariant analysis. Other optimizations are specific to functional languages, including arity analysis and tuple elimination. Although the *Id* compiler is a relatively young research tool, it is competitive with production optimizing compilers, partly because optimizations are more effective in the *Id* context. We further confine ourselves to programs for which code optimizations under the declarative and imperative styles are comparable.

Our intuition of the costs of dataflow execution versus that of more traditional methods is reflected in Figure 1. Since a dataflow model always exploits all the parallelism in a program, the work is constant; whereas, when a sequential program is "parallelized", either explicitly or via automatic translation, the extra work increases as more parallelism is exploited. The empirical data presented here suggest:

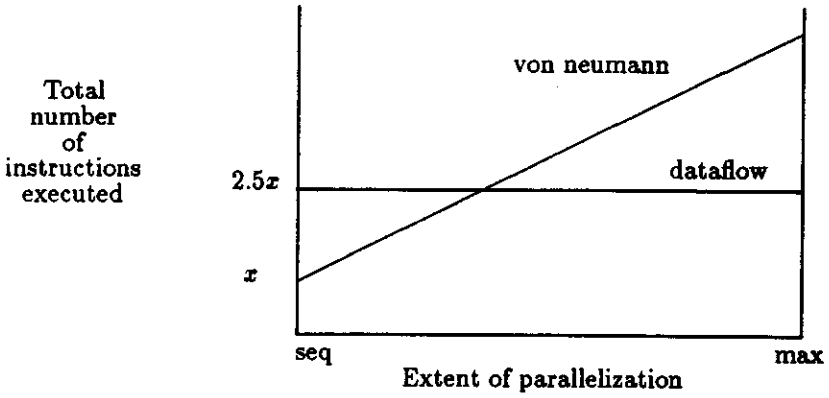

Figure 1: Work Done in Dataflow and Conventional Executions

1. The Tagged-Token Dataflow Machine executing an *Id* program performs two to three times as many instructions as an IBM 370 executing an equivalent FORTRAN program.

2. The overhead in dataflow programs is directly attributable to asynchronous parallel execution, and is typically the same magnitude as the basic work.

3. Load/Store architectures experience overhead that is also roughly the magnitude of the basic work due to register management. The dataflow architecture we consider has effectively an infinite register set, but eventually dataflow architectures may have to contend with register management overhead as well.

4. Selective parallelization on conventional multiprocessors may generate less overhead than that observed for dataflow execution by exposing only a small amount of parallelism, but as more parallelism is exposed on these machines, the overhead rapidly approaches and will surpass the dataflow level

This paper is organized as follows. Section 2 briefly describes the instruction processing pipeline and instruction set of the the MIT Tagged-Token Dataflow Architecture. Section 3 presents and analyzes experimental data on dynamic instruction counts. Section 4 discusses the costs of implementing the kind of fine grain synchronization implied by the dataflow instruction scheduling mechanism. Finally, Section 5 summarizes our conclusions and presents a comprehensive view point on the costs of parallel execution.

2 Dataflow Architecture

Traditionally a program is specified by sequence, *i.e.*, a *total order*, of operations against the machine state. In many cases this over-specifies the program, because it may be permissible for a pair of operations to be performed in any order, yet there is no way to express this. Machines such as the Cray attempt to determine dynamically where the sequential ordering is artificial and relax it on-the-fly through simple dependence analysis, in order to execute operations in parallel. In these machines, instructions are dispatched in order, although they may overlap and complete out of order. The IBM 360/91 was even more aggressive in this direction and allowed instructions to be dispatched out of order. For dataflow machines, programs are represented as a graph, rather than a sequence, of instructions so that the partial order is not obscured. Any two nodes not connected by a path in the graph represent an opportunity for parallel execution. Where an operation has multiple operands, the presence of all the operands must be detected before the operation can proceed. Thus, synchronization of parallel activity and initiation of these activities are intrinsic to the instruction execution cycle of a dataflow machine.

This section is intended as a review of the features of a dataflow architecture that are relevant to the discussion in this paper. Readers interested in further details on this subject may refer to [3, 8]. For details on the declarative language *Id* , see [6, 20, 1].

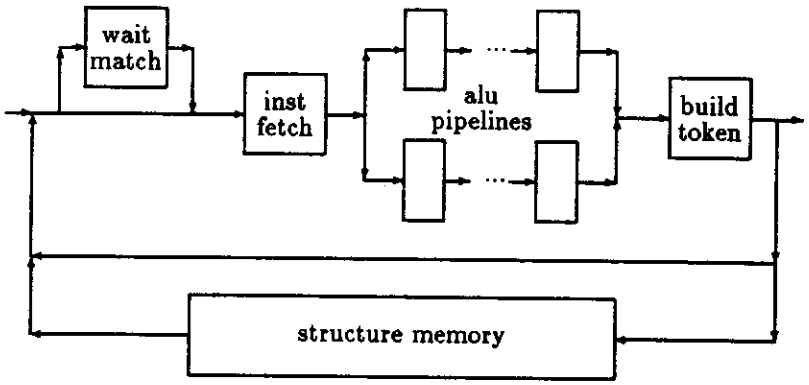

Figure 2: Pipeline of a Dataflow Processor

2.1 Pipelining Dataflow Instructions

Since the program is a graph, rather than a sequence, the locus of control is a collection of instructions, rather than a single point in the sequence. An instruction is to be enabled when the operands it requires are available, and to accomplish this each item of data, called a *token*, carries with it the equivalent of a program counter, called a *tag*, which identifies the instruction for which it is an operand. Instructions can be processed in a pipelined fashion consisting of 4 stages: wait-match, instruction fetch, arithmetic, and build-token, as shown in Figure 2.

When a token enters the wait-match stage, the tag it carries is compared against the tags on tokens present in the token-store in order to locate the other operand for the instruction specified by the tag. If present, the matching operand is fetched from the store and the instruction is enabled for processing; otherwise the incoming token is stored to await its partner. The matching operation is the basic means of synchronizing independent activities. Operands for monadic instructions, which account for approximately 40% of the dynamic mix, bypass this stage. [2]

The tag provides the address for instruction fetch, which is similar to that in a conventional machine, although the instruction format is somewhat novel. We show instruction-fetch following wait-match, but a prefetch can be initiated when a token enters wait-match; if the instruction is not enabled the prefetch is flushed[17]. Instruction-fetch may even precede matching, as in the Monsoon dataflow machine[21], in which case the instruction can specify the slot in the wait-match store to be examined and the matching function.

After instruction fetch, just as in a conventional pipelined machine, the operands and the operation code are be delivered to the arithmetic stage, which can be pipelined in a conventional manner and can perform several parallel functions: normal ALU functions, floating-point functions and tag-manipulation functions. The result is combined with successor instruction addresses specified in the instruction to yield $< tag, data >$ pairs for the wait-match stage. An instruction can have multiple successor instruction addresses, and parallel computations are initiated by generating a token for each address.

Normally, instruction pipelining is used to overlap the execution of consecutive, yet independent, operations in a single instruction sequence, which often occurs when an expression tree is linearized to form an instruction sequence. In a few machines[19], instruction pipelining is used to overlap independent sequences, which may represent distinct tasks. The dataflow pipeline serves both roles, as no distinction is made between these forms of parallelism, and a very high pipeline utilization can be achieved.

[2]The large fraction of monadic instructions results from placing values which are operands of a loop, but constant over all the iterations, in a special loop-constant area.

Data structures are stored in the I-structure store, and data is moved between this store and the processor pipeline by explicit I-fetch and I-store instructions, exactly like load and store instructions in a conventional architecture. The I-fetch does not block the pipeline, however. The address to be read, computed in the arithmetic stage, is sent as a token to the I-structure store, along with a tag for the instruction which is to receive the result. The data is returned from the store as a token, which is processed by the wait-match stage. Thus, a uniform synchronization mechanism is used for all levels of parallel activity.

The set of tokens that circulate through the processor pipeline including the tokens stored in the wait-match unit and the set of constant values stored in the constant area set up for each loop, correspond, roughly, to the register set in a load/store architecture. All ALU operations must have their operands ready in the processor (tokens in TTDA and registers in load/store architectures) and no operands can be in memory. Typically, load/store architectures use the registers as synchronization points for concurrent operations within a thread and overlapped memory requests. The size of the register set limits the amount of parallel activity, whereas the wait-match unit provides a large set of such synchronization points. This permits, for example, a large number of outstanding memory requests to be issued, masking the latency incurred by each request.

The wait-match stage of the instruction pipeline is fairly complex, but it is complex on conventional pipelined machines as well when hazard resolution and pipeline scheduling are accounted for. Typically, we find that result-store, at the end of the pipeline, must arbitrate with operand-fetch, at the beginning, for access to the registers. In wait-match, either the result is stored or the operand is fetched. We discuss the complexity of the wait-match stage further in Section 4 below. Whatever, the hardware complexity, it should be clear that a complete dataflow instruction accomplishes about the same amount of computation as an instruction on a load/store architecture. The addressing modes are primitive, the operations are typical ALU functions, and no implicit indirection is allowed. A single instruction may generate a small number of copies of a data item, not an entire vector. In fact, the dynamically scheduled pipelines in some of the very high performance machines reflect many aspects of dataflow machines.

2.2 Dataflow Instruction Set

The instruction set of a dataflow machine is novel, because it represents a graph, rather than a sequence, of instruction, but is fairly simple. A program is represented by an adjacency list of its graph; thus, each instruction specifies an operation code and a list of successor instruction addresses. Operand addresses are implicit, since the operand data tokens carry the instruction address.

The TTDA has conventional arithmetic and logical instructions. The two operands are available in the input tokens and the result goes directly into the output token. The data tokens carry types with them and hence there is some logic to examine the type of operands and direct the data to the appropriate functional unit, *e.g.,* fixed or floating point unit. For structure accesses, address computation for elements is done by an instruction that has the same complexity as an add instruction. The effective address is sent to the structure memory where the memory access takes place asynchronously, as described above.

There is no branch instruction, since there is no single locus of control. Instead, a *switch* instruction is used for conditional expressions. It takes two inputs, a data item and a boolean value, and has two successor lists. The data item is sent to one or the other of the destination lists, based upon the boolean value. In general, a branch instruction of a conventional machine corresponds to a set of switch instructions that direct the values into a target subcomputation.

There are a few tag manipulation instructions. Each of them takes a token and produces a token with an altered tag. The complexity of any of these instructions is comparable to a register-to-register arithmetic/logical operation.

Resources, such as program and data memories, are allocated by executing explicit instructions to invoke a manager program, which returns the address of the allocated resource. Low level resources, such as contexts, are managed directly by hardware, much like the stack in a conventional machine. The processing within the manager is completely asynchronous and its overhead is not accounted for in this paper. (We also did not consider the overhead in operating system calls in conventional machines for services like memory allocation.) Much of the overhead of resource management, such as termination detection, is compiled into the graph itself, and hence is accounted for explicitly.

3 Instructions executed in a program

In this section, we examine the execution of declarative programs on a dataflow machine. Among the instructions executed, we identify those that are essential to the computation, *i.e.*, that must be executed in any model of computation. those executed for control and synchronization of parallel computations. We contrast these overhead instructions with corresponding overheads in traditional machines. We illustrate the overheads through the familiar program that multiplies two given matrices. An *Id* version of this program is shown in Figure 3. It takes two matrices, A and B and returns their product matrix C. In the following discussion we assume $l = m = n$, for simplicity.

3.1 Basic Work

Figure 4 shows the basic work to be performed in this problem. It includes the n^3 multiplications and n^3 additions. Assuming that matrices are stored in memory in a conventional manner, the program must also perform $2n^3$ *load* operations for fetching elements of the two input matrices and n^2 *store* operations to store the elements of the result matrix. The control of each loop (whether performed sequentially or in parallel) requires 3 operations in each iteration to increment the loop index, compare it with the maximum and some sort of a branch to divert the control. Since there are three loops with respectively n, n^2, n^3 iterations, the basic work includes $3*(n+n^2+n^3)$ operations for this. Finally, the compare and branch operations must be executed one more time in each loop to determine the end of the loop and this gives rise to $2*(1+n+n^2)$ more operations.

3.2 Parallel Loops

The instructions for parallel loop control in a dataflow machine are summarized in Figure 5. In place of a branch, every value circulated from one iteration to another must be passed through a switch that directs the value either to the next iteration or to the result expression based on the loop condition. Values circulate through the loop asynchronously, and potentially all iterations of a loop may execute concurrently, thus tokens for every iteration must be colored distinctly. Hence, circulating values also pass trough a recoloring instruction. In the matrix multiplication example the circulating values are i in the outer loop, j in the middle loop and k, s in the inner loop. However, each loop must also circulate a termination signal which is used to deallocate resources. The compiler was able to optimize away the termination signal for the innermost loop, thus two values are circulated in each loop. Hence, $2*(n+n^2+n^3)$ coloring operations and $2*(n+n^2+n^3)$ switch operations are performed. Since the switch operations for the loop indices have already been accounted for as branches under basic work, Figure 5 shows only the remaining switch operations. Each value returned by the loop must be passed through an extra switch operator and values that appear in the result expression must be decolored. In this way, some of the results of a loop may be available while others are still being computed. In this example only the value s is used in the result. Finally, to keep resources under control, the bounded loop schema [12, 25] introduces three throttling operators per iteration to permit only a given number of iterations at any time. Since these instructions are executed in each iteration, we see the corresponding overhead.

3.3 Other Overheads Due to Asynchronous Execution

Dataflow model incurs further overhead in order to provide for maximal asynchrony in function calls and conditional expressions. We briefly comment on these overheads. Instructions executed in the matrix multiplication example under various categories are shown in Figure 6.

Function Calls: The function call mechanism in a dataflow model facilitates not only concurrent execution of instantiations of one or more functions, but also asynchronous passing of arguments and results between them. There are special instructions to instantiate a function, which can commence parts of its execution that are not dependent on any argument. The caller sends each argument (using a change-tag instruction) directly to the corresponding node in the body of the newly instantiated function. Execution within the called function body proceeds as each argument arrives, so parts of the function execute in parallel with the computation of some of the arguments. In a similar manner, results are returned asynchronously, each result triggering the execution of the corresponding part of the caller. Since all the function call

```
Def matmult A B l m n =
  { C = matrix ((1,l),(1,n));
    {For i From 1 To l Do
       {For j From 1 To n Do
          s = 0;
          C[i,j] = {For k From 1 To m Do
                      Next s = s + A[i,k] * B[k,j];
                    Finally s}}};
  In C};
```

Figure 3: Matrix Multiply Program in *Id*

Additions, Multiplications	$2n^3$	8192
Matrix element loads, stores	$2n^3 + n^2$	8448
Loop index increment, compare, branch	$3 * (n + n^2 + n^3)$	13104
Zero-trip compare, branch	$2 * (1 + n + n^2)$	546
Total basic work		30290

Figure 4: Basic Work in Matrix Multiply ($n = 16$)

Recolor circulated values	(2 *values/iter*) $* (n + n^2 + n^3)$	8736
Switch circulated values	(1 *value /iter*) $* (n + n^2 + n^3)$	4368
Decolor result values	(1 *value in inner loop*) $* n^2$	256
Switch result values	(1 *value /loop*) $* (1 + n + n^2)$	273
Loop bounding	(3 *ops /iter*) $* (n + n^2 + n^3)$	13104
Total loop control		26737

Figure 5: TTDA Instructions for the Control of Parallel Loops in Matrix Multiply ($n = 16$)

	normal	inner loop unrolled once
Basic Work	30290	23890
Parallel Loop Control	26737	15430
Asynchronous arg passing	4110	4496
Termination detection	5730	6676
Condition handling	0	768
Structure handling	4128	4128
Other Miscellaneous	1349	2189
Total	72344	57577

Figure 6: TTDA Dynamic Instruction Counts for Matrix Multiply ($n = 16$)

overhead is due to the execution of special instructions, it is easy to isolate this from an execution trace. Line 3 of Figure 6 gives the total number of these instructions executed in the matrix multiply program. Since functions are used as basic building blocks in declarative languages, the function call overhead can be substantial. The compiler can optimize them away in many cases by performing inline substitution for the body of a function. The use of higher-order functions can incur additional overhead in building closures, but this too can be reduced through compiler optimizations.

Termination Detection: The asynchronous nature of functions described above calls for dynamic deallocation of the activation frames as each function terminates, so that they can be reused. Since each function itself has many parts executing in parallel, its termination must imply the termination of all sink nodes, *i.e.*, nodes with no useful output, in the function body. In the current implementation, this is accomplished by forming an inverted tree of *identity* operators which collect completion signals from all the sink nodes of a function body. Once again, this overhead can be isolated by counting these instructions as reported in line 4 of Figure 6. A sophisticated compiler can optimize away several of these operations by analysis of the graphs.

Conditionals: Conditionals are another source of overhead due to parallel execution. In a data-driven execution, the computation in an arm of a conditional is not initiated until the predicate evaluation is completed and the arm is selected. This way no wasteful computation is performed. However, this implies that the values needed for the two arms of a conditional must be guided to the appropriate arm using a switch instruction for each value. If a value is used in only one arm, the corresponding switch instruction discards the value when that arm is not selected. (Actually, it gets fed into a termination tree.) Once the predicate of the conditional is evaluated, portions of the selected arm may begin execution as soon as any of the inputs are available. This is an important source of parallelism in many recursive programs. In contrast, a traditional machine uses a single jump instruction for a conditional and the selected arm accesses whichever variables it requires. The overhead due to conditionals can be isolated by counting the number of switch operations performed (in addition to the switch operations accounted for loop control). The matrix multiply example does not involve any conditionals, as indicated by line 5 in Figure 6.

Structure Handling: Although in this paper we consider only examples which do not exhibit storage overheads due to the functional programming style, we would like to point out one peculiarity of the current implementation of matrices. Matrices are implemented as arrays of arrays. Hence, to access an array element, 2 *load* operations are needed, one to obtain the address of the row and the other to obtain the element within it. In the matrix multiply example, in addition to the normally expected *loads* shown in Figure 4, $2n^3$ more load operations must be executed. Moving the constant row address out of the middle loop halves this overhead, as shown in line 6 of Figure 6.

3.4 Loop unrolling

From Figure 6, we observe that the dataflow model pays a substantial loop control overhead of roughly 37% of the total number of instructions of Matrix Multiply ($n = 16$). It may seem wasteful to pay the overhead of potential parallel execution in situations like the inner loop of matrix multiplication, which has a linear recurrence. While this is partly true, consider the behavior of this loop under the assumption that the array accesses generate remote memory references involving high latency. Under asynchronous execution, a number of iterations will execute concurrently, causing memory requests to be issued to fetch the matrix elements for a number of iterations. As these complete, the values will be multiplied and accumulated. Thus, the latency for each individual request. This effectively pipelines the memory requests to mask the latency of any individual request.

One can also perform loop unrolling at compile time to reduce the control overhead. For example, the second column of Figure 6 shows the instruction counts when the inner loop of the matrix multiply program is unrolled once. The basic work is reduced because the number of iterations in the inner loop is reduced by half. The parallel loop control is reduced by almost 40%. The minor increases in instructions for argument passing and termination detection are due to the increase in constants to be passed and switches used in conditionals. Each time the inner loop is initiated, the sum is initialized to zero or to the product of the first two elements, depending on whether n is even or odd. Three values have to be switched each time this is done, which accounts for the $3n^2$ switch operations as condition handling overhead.

	dataflow	IBM 801	IBM 370
Basic Work	30290	30290	18002
Parallel Loop Control	26737	0	0
Asynchronous arg passing	4110	0	0
Termination detection	5730	0	0
Structure handling	4128	4640	4896
Other Miscellaneous	1349	1354	3274
Total	72344	36284	26172

Figure 7: Dynamic Instruction Counts on Various Machines for Matrix Multiply ($n = 16$)

3.5 Comparison with Sequential Machines

To gauge the significance of this overhead, we contrast the number of instructions executed by a dataflow machine in the matrix multiply problem, with sequential execution on conventional machines. Figure 7 shows this comparison. First, consider the second column which gives the instruction counts on a simulator of the IBM 801 architecture [23]. The basic work is the same as for a dataflow machine. There are no overheads associated with the next three categories as the execution is sequential and has no asynchrony. Structure accesses involve some additional computation, since pointers into the matrices are kept in registers and are updated in each iteration. We have $2*(n+n^2+n^3)$ updates for pointers into the two input matrices and n^2 updates for the pointer into the result matrix. Usually one of the pointers is used for controlling the loop as well. In Matrix Multiply ($n = 16$), 4640 instructions were executed for this purpose. Finally there are 1354 miscellaneous operations for moving data in and out of registers for various reasons, giving a total of 36284.

The last column in Figure 7 gives the counts for sequential execution on the IBM 370 machine. This machine has a powerful instruction BXLE which performs an increment, compare and branch all in one instruction. The inner loop is controlled by a BXLE and this reduces the basic work by $2n^3$ operations. Also, the program accesses the elements of one matrix as a memory operand of the multiply instruction. This eliminates n^3 load operations from the basic work. Hence the basic work is accomplished in only 18002 instructions. The overhead for structure accesses is similar to the IBM 801 machine. The machine has more miscellaneous operations for setting up the registers and other address constant constants for efficient use of its more complex addressing modes. The larger setup costs are offset by the saving in the bulk computation under basic work.

3.6 Other Examples

Figure 8 contrasts the total number of instructions executed for a set of programs on the dataflow machines with corresponding counts for IBM 370 . Although the programs are simple, we believe they are fairly representative of a broader class of programs. We observe that the instruction counts on a dataflow machine are about 2-3 times the counts on IBM 370 machine. The first program sums two pairs of vector and computes the inner product of the resulting vectors. Conventional parallelizations easily cause the vectors sums to proceed in parallel, but it is quite hard to pipeline it along with the inner product. The dataflow machine does this automatically and still takes only 2.5 times as many instructions for this as the IBM 370 . The second row suggests that ratio for the matrix multiplication example changes little as the dimensions increase. The third program, LU decomposition, is a classic example where conventional programs heavily capitalize on storage reuse by updating in place. Declarative languages are known to cause inefficiencies in these situations. The counts indicate that the situation is not as bad as one might expect. [3] Fourthly, multiple table lookups using binary search was chosen to illustrate the overhead of conditionals. As discussed earlier, dataflow computations generally incur overhead to switch values to the appropriate arms of a conditional. Here, three values (low,middle,and high) are steered into a conditional which selects two as the new values of low and high. Surprisingly, the dataflow machine is at parity for this example. Although conventional

[3] These counts include the instructions for invoking and synchronizing with a dynamic memory allocator, but not the instructions to actually perform the allocation.

	dataflow	IBM 370
Vector sum and inner product (size 50)	2816	1266
Matrix multiply (size 30 × 30)	443561	156840
LU decomposition (size 16 × 16)	35552	13876
Table lookup (size 50)	9541	10757
Simple 1 iteration (size 10 × 10)	196848	108117

Figure 8: Total Dynamic Instruction Counts on Various Examples

	cray-xmp	MIPS
Vector sum and inner product (size 50)	1843	
Matrix multiply (size 30 × 30)	413946	374808
LU decomposition (size 16 × 16)	33801	
Table lookup (size 50)	10629	9904
Simple 1 iteration (size 10 × 10)	220014	154429

Figure 9: Total Dynamic Instruction Counts on Load-Store Architectures

machines can use a single jump instruction to implement a conditional, they still have to move the values to the right place to be used by the subsequent computations. This, suggests that overheads in conditional need more careful study. Finally the last program is a large hydrodynamics code called SIMPLE and exhibits a complex mix of constructs. This range of 1.9 to 2.8 times the number of IBM 370 instruction is typical of the various programs and problem sizes we have studied.

3.7 Load Store Architectures

Since the instruction set of a dataflow machine is closer to that of a load-store architecture, comparisons with machine of this class would be more meaningful. Although we were unable to obtain detailed data as on the IBM machines, we present some preliminary data on two architectures. Figure 9 shows the instruction counts for our five examples. The first column shows total instruction counts observed through a hardware monitor on the Cray-XMP machine. The programs were compiled using the CFT77 compiler version 2.0.0, with the vectorization turned off. The second column gives the instruction counts obtained from the MIPS[16] simulator at Stanford University using the F77 compiler, version 1.2.1. The MIPS count for matrix multiply includes some initialization code and should be reduced by approximately 20000 instructions.

The fact that the instruction counts fall in between the corresponding numbers for dataflow and IBM 370 is to be expected. However, one might be surprised that they are closer to dataflow counts than the IBM 370 counts. Examination of the matrix multiply example on the Cray machine showed that 202500 instructions out of the total of 413946 are instructions that move data from one register to another register. This was partly due to having many kinds of registers (A, B, S, and T) and partly a result of trying to expose local parallelism. A dataflow machine conceptually has infinite registers, although a practical machine will have to deal with this finite resource problem. The dataflow counts do not reflect this aspect, and the subject needs further study. The Cray version unrolled the inner loop 16 times; as indicated above, similar optimizations would improve the dataflow numbers.

3.8 Conventional Parallelizations

Most conventional multiprocessor systems engage in concurrent computation only for selected portions of a program. This selection is made either by a compiler or by the programmer through annotations. While the dataflow model incurs the costs for parallel execution independent of the degree of parallelism, a conventional system might tune these costs selectively. It would be very interesting to see the relationship between the utilized parallelism and the associated costs in a conventional multiprocessor *vis. a vis.* the overheads in

a dataflow system that always uses all the parallelism that is present. Unfortunately, it has proved very difficult to obtain reliable, detailed data on existing multiprocessing systems. Below we discuss results from EPEX [24], at IBM, which simulates an RP3 style multiprocessor employing the SPMD (Single Program Multiple Data) mode of execution. Typical segments of code in this system are illustrated below:

```
@serial-begin
 .... (serial segment)
@serial-end
 .... (replicated segment)
@do-begin i=1,n
 .... (parallel loop)
@do-end
```

All processors execute the same program and special annotated code segments can be executed either serially or concurrently. *Replicated* segments are those that are outside any annotation and are executed by all processors. They typically contain local state initializations and the control flow through the program which every processor must follow. Serial segments are executed by the first processor reaching the @serial-begin and all other processors skip around it. Appropriate lock and unlock mechanisms are executed at the beginning and end of such a serial segment. Similarly, when a processor reaches a @do-begin statement, it picks up the next available loop index from the parallel loop and executes the corresponding iteration. Upon completing that iteration, the processor gets another index and so on until all indices are exhausted. Usually, at the end of a serial or a parallel loop segment, processors must wait and synchronize with others before they can proceed further.

Under this approach a variety of overheads arise:

1. Synchronization - instructions for locking, unlocking, etc. at segment boundaries;
2. Busy waiting - instructions executed while waiting for other processors to reach a barrier;
3. Instructions to initialize local state, synchronizing variables *etc.* (replicated segments);
4. Instructions due to loss of optimizations because of parallelization.

The cost of synchronization can be modeled as follows. Ideally, each of the annotations shown above translates to a sequence of 3 or 4 instructions like replace-add, test, and branch. To simplify the analysis, let c be the average number of instructions taken by each of these constructs and let p be the number of processors in the system. Then a serial segment takes $(p+1)*c$ instructions for synchronization, as all p processors must execute the prologue and one processor must execute the epilogue. Setting up the necessary counters for a parallel loop takes $p*c$ instructions. (Although only one processor sets up the loop, all the other processors must check and skip over the initialization.) Executing a parallel loop of n iterations takes $(n+p)*c$ synchronizing instructions (c for getting the index of each iteration and c for each processor to determine that there are no more indices).

For both serial and parallel segments, all processors busy wait at the end to synchronize. The cost of busy waiting is difficult to determine based on an EPEX simulation because it is sensitive to the scheduling policy of the operating system underlying the tasks that represent EPEX processors. In a dataflow machine a processor busy waits only when there is no further work to be done. It is not clear how we can fairly account for these busy waits in the two systems. Thus, we will exclude instructions executed in the busy-wait loop from our instruction counts.

Finally, the last two factors are program dependent and difficult to quantify. The replicated computation arises because, generally, global variables used in a loop will be moved to local storage in the prologue; such copies will be performed in each of the p processors. To illustrate the last factor, consider a loop which traverses a row of a matrix stored in column-major order. In compilation for a sequential machine, the multiplication used to compute the element address is eliminated by adding column length in each iteration. This optimation is not possible if each processor is performing a random collection of the iterations.

Figure 10 gives the instruction counts observed on an EPEX simulation on an IBM 370 machine for the same set of problems, with varying number of processors. Unfortunately the numbers are unrealistic, as the instructions for simulating c discussed above varies between 60-200 IBM 370 instructions. But the table still has some interesting aspects. The instruction counts for executing with one processor are sometimes significantly higher than the corresponding counts of the same problems on the same machine as shown in the

	$p=1$	$p=4$	$p=8$	$p=16$
Vector sum and inner product (size 50)	11376	13654	26622	22920
Matrix multiply (size 30 × 30)	157121	158028	159252	161676
LU decomposition (size 16 × 16)	33872	55874	88469	157655
Simple 1 iteration (size 10 × 10)	143313	166887	199413	269137

Figure 10: Total Instruction Counts on EPEX Simulator Excluding Busy-Waits

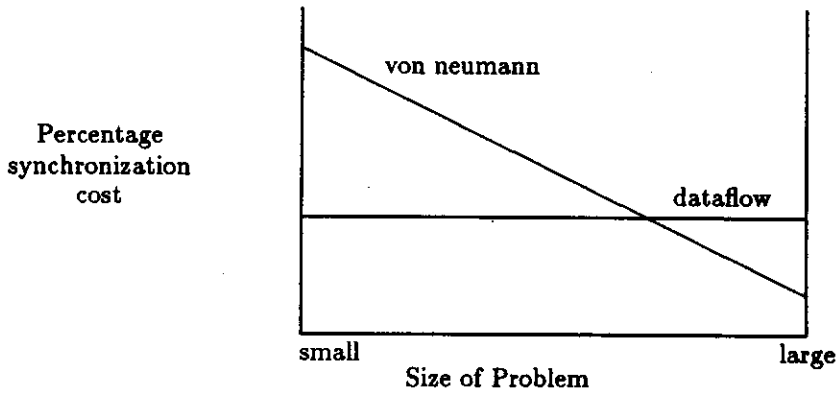

Figure 11: Synchronization Costs in Dataflow and Conventional Executions

last column of Figure 8. This is due to one time execution of replicated segments as well as synchronization cost for each parallel iteration. When number of processors increases, the counts go up in accordance with the formulae described above, but since the value of c in the epex simulation is not realistic, we cannot draw any strong conclusions.

We would like to draw attention to differences in synchronization costs in the two models of computation. For simplicity, consider a single loop whose body takes b instructions for one iteration. Let n be the number of iterations. Usually, n depends on the size of the problem. When this loop is concurrently executed, let c be the number of additional synchronization instructions executed in each iteration. The percentatge overhead of this synchronization cost is given by the ratio $(c * n)/(b * n) = c/b$. In dataflow executions, all loops are parallelized uniformly and, hence, the percentage synchronization cost, c/b, is a constant for each loop. The parallelization strategy is to parallelize all loops, independent of the number of processors and the size of the problem.

For conventional multiprocessors, parallelization is done selectively. Usually, in a nesting of loops, only one loop can be parallelized. When an outer loop is parallelized, the percentage ratio of synchronization cost, c/b, becomes a function of n. Usually as the problem size grows, b increases and the ratio diminishes. Often, parallelization strategies are tuned for very large problems so that small synchronization overheads are achieved (*e.g.*, see [13]), but this number will not be met for smaller problems. Figure 11 reflects our intuition on how synchronization costs vary with problem size for a fixed parallelization strategy in dataflow and von Neumann systems. Conventional systems often quote the synchronization cost at the far end of the x-axis and the same strategy for smaller problem sizes is usually prohibitive. Adaptive parallelizations can be devised for von neumann systems, but we do not know of any existing systems that employ them.

Thus, dataflow models adopt a uniform parallelization strategy and hence have their synchronization costs increase uniformly with the problem size. Conventional parallelizations, must tailor their strategy for a given problem size to keep the costs down. This paper suggests that the uniform strategy in the dataflow model increases the synchronization costs at most by a factor of 2-3 compared to a sequential execution. Any partial parallelization has only this limited margin for improvement.

4 Comment on Dataflow Hardware Costs

The hardware implementation of the dataflow instruction scheduling mechanism has been refined by a number of research groups over the years [4, 15, 3, 17, 21]. While it is not yet possible to assess quantitatively the costs of dataflow and conventional multiprocessors, we believe that the dataflow hardware costs are not inordinately greater than the conventional counterparts. In this section, we briefly relate some of the reasons for this belief. We focus on the special hardware features in a dataflow machine, which are often thought of as complex, and discuss some less alarming alternatives that are underway.

The most striking complexity in a dataflow machine is in the wait-match stage. Synchronization of two data tokens bearing the same tag seems to require the equivalent of a large associative memory; the token store is effectively addressed by its contexts. However it is implemented, the token store must be large, for if it overflows the program will deadlock[11]. Large associative memories are obviously expensive, but also are too slow to be used effectively as a stage in the instruction processing pipeline. All existing tagged-token dataflow machines employ hashing to get the functionality of associated memory at reasonable cost, but still to keep the pipeline beat reasonably fast, a pipelined hashing strategy must be pursued. This implies a rather long pipeline, which degrades performance in situations of low parallelism. Even so, to achieve effective hashing, a substantial fraction of the store must be left unused. A more effective job of hashing can be employed if the compiler plans the usage of the token store for portions of the program, *i.e.,* if the usage of the activation frame is determined statically. A block of storage is allocated when a function is invoked to hold any tokens that must wait during the function activation [21]. The compiler walks the dataflow graph and assigns slots so that no two tokens that can potentially co-exist use the same slot; this is essentially like register allocation through graph coloring[10]. The compiler can then translate the graph into machine code in which the tags on data tokens are offsets into the activation frame. A presence bit in each memory cell can be used to detect the presence or absence of data in the cell. Thus, a matching operation involves first accessing the presence bit in fast memory and then either a read or a write to the token store. This preallocation scheme uses more memory in general, because the sharing of the memory is not as effective as in the associative memory scheme, but the cost and access time are much less. Hybrid machines[18] take this direction farther and employ presence bits only at points where sequential threads must synchronize.

As suggested above, circulating tokens and tokens in the wait-match memory are analogous to registers in a conventional machine. However, if this token storage is quite large it can hardly be considered high-speed processor state, and instead intimates that the dataflow machine resembles more a memory operand machine rather than a register machine. Two factors mitigate this concern. First, the token store has a dual role; it serves as registers for short lived values and also as stack storage for longer lived ones. The majority of tokens find their partner after a small number of instructions [9]; this means that a variety of caching and scheduling techniques can be employed so that only a small portion of the token store need be maintained in fast storage. Secondly, on the average only 40% of the tokens require matching at all (*e.g.*, see statistics in [15]).

We believe it is essential that multiprocessors support a large number of logical register names, however these are mapped into processor state, in order to tolerate long, unpredictable memory latency. Take, for example, a load-store architecture such as the Cray. It can mask memory latency to a small degree by issuing multiple requests to memory; each request has associated with it a register that will receive the data value. Thus, at most eight scalar references can be outstanding. As memories become more distant from the processor, the latency increases and more register names are required to mask this latency successfully. In the dataflow machine, addresses in the token store serve precisely this role as synchronization points. As a result, dataflow machines have high tolerance to memory latency as expounded, in [7].

In addition, the above scheme facilitates a uniform treatment to all memory components in a dataflow machine. I-structure implementation requires tagged memory any way. Now I-structures, waiting tokens, programs, constants can all be stored in the same memory, thereby reducing the complexity of the memory management so that it can deal with only one class of memory. Another added advantage of this scheme is that input output processing can progress concurrently with other computation, once again at a fine granularity of individual elements. Conventional systems usually synchronize at a higher granularity of i/o blocks. While providing high-bandwidth i/o and fast communication networks are problems common to all multiprocessors, we believe that asynchronous i/o fits naturally into the dataflow model and facilitates asynchrony at a fine granularity. We would like to point out that tying i/o in with dataflow methods may

compromise the elegance of the language, but not the hardware.

Another oft-cited hardware complexity in dataflow machines is the need for wide data paths. Token processing in a processor does require wide paths of the order of 128 bits. But we believe that the logic involved is so simple that it presents no problems in comparison with the complexity of interlock detection in a high speed conventional processor, for instance.

5 Conclusions

In this paper we provided a cost/benefit analysis of a dataflow model, in which all instructions are executed in an asynchronous manner to take advantage of all the fine grain parallelism expressed by an algorithm. The framework for the analysis is the *Id World* programming model at MIT, which uses a declarative programming language called *Id* and a tagged-token dataflow architecture for execution. Although the details of the analysis are derived from this model, we believe that the conclusions apply to many computation models.

In order to evaluate the costs of an approach, we must settle upon a cost metric. Ultimately, we should like to run the program and count the seconds that elapse and the dollars involved, but this is obviously not an option in the early stages of development. Moreover, it provides very little information on the reasons for the differences. The overhead due to additional work is indistinguishable from time spent idling because some processors had no work to do and time lost due to inadequacies in the architecture or implementation. In this paper we have examined the total number of instructions executed in programs under dataflow execution and compared this to instruction counts on conventional machines. We believe that the number of instructions executed is a meaningful cost metric in this context. Overhead introduced by attempting to execute programs in parallel does manifest itself in terms of this metric and can be traced back to particular aspects of the program and its machine representation. Of course, this assumes that the instruction set is relatively simple, *i.e.*, it is implemented directly, without a level of microcode interpretation.

Since we are concerned with high-performance machines, we expect instruction execution to be pipelined, so perhaps we should count cycles rather than instructions. However, like counting elapsed seconds, this obscures the point of study because the number of cycles will depend heavily on the particular way the instruction set is implemented and the technology used. It also complicates the task of tracing the source of overhead back to particular aspects of the program. Focusing on cycles does, however, raise an important point; the numbers of cycles per instruction may increase in a multiprocessor context due to increased latency in memory accesses. Dataflow machines maintain an effective instruction processing pipeline in spite of long, unpredictable memory and communication latency, whereas conventional processors do not, so in fact counting instructions gives a conservative comparison.

Based on empirical results, we conclude:

1. The total number of instructions executed in fully parallel, asynchronous dataflow execution of programs is 2 to 3 times that of equivalent programs on the IBM 370 and close to that on load/store architectures.
2. Parallel programs for conventional multiprocessors may encounter less overhead if only a small amount of parallelism is utilized, but rapidly approach the overhead for dataflow execution as more parallelism is exploited.
3. The overhead in dataflow execution is directly attributable to asynchronous parallel execution and is typically of the same magnitude as the basic work.

The last point is made by showing that the instructions executed in a dataflow model can be partitioned into basic work and overheads. The overheads can be subdivided further and associated with different synchronizing events such as conditionals, parallel iterations, parallel function invocations *etc.* This is useful for cost/benefit trade-offs at various levels of parallelization. The other points are made by providing a brief comparison with conventional machines. The relative costs for partial and complete parallelizations do not appear to be as disparate as commonly believed. The argument for fine grain parallelism may be further strengthened if the costs of programming and hardware can be quantified in a similar manner.

Acknowledgements

We gratefully acknowledge the contributions of Olaf Lubeck and his group at Los Alamos National Laboratory, who collected data on the Cray Xmp, and John Hennessy at Stanford University, who collected data on MIPS. Thanks also to Natalie Tarbet for assistance on the final draft.

References

[1] Arvind. Programming Generality and Parallel Computers. Technical Report CSG Memo 287, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1988. To appear in the Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy.

[2] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England.* North-Holland Publishing Company, July 15-18 1985.

[3] Arvind and David E. Culler. *Dataflow Architectures*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, IEEE Computer Society Press, 1987.

[4] Arvind, David E. Culler, Robert A. Iannucci, Vinod Kathail, Keshav Pingali, and Robert E. Thomas. The Tagged Token Dataflow Architecture. Technical Report FLA, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1983. Revised October, 1984.

[5] Arvind, David E. Culler, and Gino K. Maa. Assessing the Benefits of Fine Grain Parallelism in Dataflow Programs. Technical Report CSG Memo 279, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, June 1988. To appear in the International Journal of Supercomputer Applications, MIT Press.

[6] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987. To appear in the Journal of Parallel and Distributed Computing.

[7] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987.

[8] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259).* Springer-Verlag, June 15-19 1987. To appear in IEEE Transactions on Computers, revised June, 1988.

[9] Stephen A. Brobst. Organization of an Instruction Scheduling and Token Storage Unit in a Tagged Token Dataflow Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 40–45, August 1987.

[10] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[11] David E. Culler. Resource Management for the Tagged-Token Dataflow Architecture. Technical Report TR-332, Massachusetts Institute for Technology, Laboratory for Computer Science, January 1985.

[12] David E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Fifteenth Annual International Symposium on Computer Architecture*, pages 141–150, Hawaii, May 1988.

[13] F. Darema-Rogers. Parallel applications development for shared memory systems. Technical Report RC 12229, IBM T.J.Watson Research Center, Yorktown Heights, New York, 1986.

[14] D.D. Gajski, D.A. Padua, David J. Kuck, and R.H. Kuhn. A second opinion of data flow machines and languages. *Computer*, 15(2):58–69, February 1982.

[15] John R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[16] John L. Hennessy. VLSI Processor Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[17] K. Hiraki. Instruction set reference manual for sigma-1. Technical Report Tech report, Electro Technical Laboratory, Japan, 1983.

[18] Robert A. Iannucci. *A Dataflow/von Neumann Hybrid Architcture*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988.

[19] Harry F. Jordan. Performance Measurement on HEP - A Pipelined MIMD Computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden*. North-Holland Publishing Company, June 1983.

[20] Rishiyur Sivaswami Nikhil. ID (Version 88.0) Reference Manual. Technical Report CSG Memo 284, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 1988.

[21] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988 (expected).

[22] D.A. Patterson. Reduced instruction set computers. *Communications of the Association for Computing Machinery*, 28(1):9–21, January 1985.

[23] G. Radin. The 801 minicomputer. In *Proceedings of symposium on architectural support for programming languages and operating systems*. ACM, March 1-3 1982.

[24] J.M. Stone, F. Darema-Rogers, V.A. Norton, and G.F. Pfister. The vm/epex fortran preprocessor reference. Technical Report RC 11408, IBM T.J.Watson Research Center, Hawthorne, New York, September 1985.

[25] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, Massachusetts Institute for Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

[26] Kenneth R. Traub. *Sequential Implementation of Non-Strict Languages*. PhD thesis, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988.