UC Irvine

ICS Technical Reports

Title

A process-oriented model for efficient execution of dataflow programs

Permalink

<https://escholarship.org/uc/item/73h196xc>

Author

Bic, Lubomir

Publication Date

1986-11-21

Peer reviewed

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES

Z

699

C3

no. 86-23

C. 2

A Process-Oriented Model for Efficient Execution of Dataflow Programs[†]

Lubomir Bic

Department of Information and Computer Science
University of California, Irvine
November 21, 1986

Technical Report 86-23

ABSTRACT

In a dataflow program, an instruction is enabled whenever all of its operands have been produced; at that time, the instruction packet is eligible for execution by a free processor. Compared to a von Neumann computer, the major sources of overhead are (1) the need for matching of token destined for the same instruction, (2) routing of tokens among processors, and (3) the fact that instructions are scheduled for execution individually, one at a time. In this paper, we present an execution model that reduces much of this overhead. A dataflow program is broken into sequences of instructions that must be executed sequentially due to their data dependencies. Each sequence is loaded into execution memory as a whole, where it forms a very simple process. A processor is then multiplexed among the ready processes in its local memory. The states of these processes change between running, ready, and blocked, depending on the arrival of operands. The main advantage is that operands produced and consumed within the same sequence are stored directly in one memory operation, thus bypassing the token matching and routing units. Consequently, when executing highly sequential programs, the dataflow machine "degenerates" to an efficient von Neumann computer.

[†] This work was supported by the NSF Grant DCR-8503589: The UCI Dataflow Databases Project.

Contents

	Page
Introduction	1
Existing Dataflow Architectures	4
The Execution Model	6
Sequential Code Segments	6
Translation of Dataflow Graphs into SCSs	10
The Machine Architecture	11
Activity Names and Token Matching	12
The Fetch/Execute Unit	16
I-Structure Storage	17
Comparison of the Process-Oriented Model to Other Approaches	18
Conclusions	21
References	23

1. Introduction

Dataflow computation was proposed as an alternative to the conventional von Neumann style of computing to exploit parallelism [DEN75]. A dataflow program is a directed graph where nodes represent *operators* and arcs show the *data dependencies* among operators. Operands are carried on packages called *tokens* along these arcs. An operator is enabled when all its input operands have arrived on incoming arcs. It executes by consuming these values and produces results sent along its output arcs to other operators expecting these values as inputs. In this way, computation proceeds in a *data-driven* manner without the need for any centralized *control* (program counter), used in von Neumann computers to decide when an operation should be executed, and without the concept of *memory* to store operations and data.

Example: Figure 1 shows the dataflow graph to evaluate the two roots of a quadratic equation, $(-b \pm \sqrt{b^2 - 4 * a * c}) / 2 * a$. The graph expects three input values to arrive on the lines labeled a , b , and c . The arrival on b triggers the execution of the unary “ $-$ ” and the “ $\uparrow 2$ ” operators. The resulting values are sent to the “ $+$ ” and “ $-$ ” operators, respectively. These, however, must wait for the arrival of the other operands produced by preceding operators. In particular, the “ $+$ ” operator must wait for the completion of the square root operator, which, in turn, must wait for the completion of the “ $-$ ” operator, and so on. In this way, execution is sequenced automatically by the availability of intermediate results. Similar data dependencies apply to the remaining operators.

All dataflow systems obey the basic principles of data-driven computation described above. With respect to executing loops, however, two basic approaches can be distinguished. In the first, referred to as *static* interpretation [DEN75], there is only one instance of the subgraph representing a loop body at any given time. Tokens cycle through the same instance of the graph for the duration of the loop. To guarantee that a program executes correctly, it is essential that tokens from different

Figure 1

A Dataflow Graph

iterations do not overtake one another. This is ensured by feedback signals, which inhibit the execution of an operator until all its output arcs have no more tokens. This approach is called *feedback interpretation*.[†]

The second approach [AGP78] uses a concept called *loop unraveling*, where a separate copy of the graph is created for each iteration of the loop. This makes the use of feedback signals unnecessary, since tokens produced during one iteration and destined to the next are passed to a *separate instance* of the graph.

Example: Consider the following simple loop in Id (the *Irvine dataflow language* [AGP78]), which iteratively applies the function f to x 5-times:

[†] Even more restrictive approaches can be found in older dataflow machines, e.g., LAU [CoSy80], where only one instance of any loop may be active at any given time. That is, only tokens belonging to the same iteration may exist in the loop graph concurrently.

Figure 2

Example of a Loop

for $i = 1$ to 5 do

$x := f(x)$

The corresponding dataflow graph is shown in Figure 2.[†] Under static interpretation, the operators L , L^{-1} , d and d^{-1} , shown using dashed lines, would not be present. The loop operates as follows. The initial input values x and i are sent to the respective switch operators. A copy of i is also sent to the predicate operator (≤ 5), which produces a boolean value and sends it to both switch operators. When this value is true, the values x and i are output on the arc designated by T (for true) and hence reach the function operator f and the increment operator ($+1$), respectively. The resulting values are recirculated into the switches and the predicate operator; this process is repeated until a false value produced by the predicate operator routes the x token out of the loop.

The presence of the L , L^{-1} , d and d^{-1} operators implements the unraveling interpretation. L first establishes a new context for the loop; this permits arbitrary

[†] Dennis' base language uses a different set of operators, but the basic principles are the same.

Figure 3

Operation Cycle of Dataflow Computers

loop nesting. The function of the d operator is to route the received token into a *different* copy of the same dataflow graph, corresponding to the next iteration. The result token produced by the last iteration of the loop passes through the d^{-1} and L^{-1} operators, which reset the iteration count and restore the original context. These principles permit possibly many instances of the same loop to be active simultaneously — the loop is unraveled in time and space. For example, the value i could circulate through the loop at a faster rate than the value of x . This results in a greater degree of parallelism than the feedback interpretation.

1.1. Existing Dataflow Architectures

The original motivation of dataflow was to build machines consisting of very large numbers of asynchronously operating processing elements (PEs), communicating with one another through an interconnection network. The best known conceptual architecture to support the feedback interpreter is that of Jack Dennis [DEMI75]. Two representatives of architectures built to support the unraveling interpretation are those proposed by Arvind, Gostelow, and Plouffe [AGP78, ARGO82] and Gurd and Watson [WAGU82, GKW85]. While quite different in their design, all these architectures are based on the following operational principles. There are four basic tasks to be performed repeatedly during execution: (1) matching of operand tokens destined for the same instruction, (2) fetching of enabled instructions, (3) instruction execution, and (4) routing of tokens. These four tasks are performed in a cycle depicted graphically in Figure 3. In the feedback architecture, each arriving operand token is stored immediately into the operand slot of

the appropriate instruction. When all operands for that instruction have arrived, it is fetched and passed to a free PE for execution. The resulting tokens are then routed to the instructions expecting these as operands. In the unraveling dataflow machine, operand tokens are not stored immediately in instructions. Rather, an explicit *matching store* is provided, in which operands destined for the same instruction are segregated. Only when all have arrived is the corresponding instruction fetched and sent to a PE for execution. Resulting tokens are then routed back into the matching store to find their counterparts for subsequent operations. This basic cycle is repeated until the program terminates.

In both realizations of the dataflow concept, there is significant overhead in executing each operation: First, every data token must travel considerable distances between any two instructions as it passes through the basic cycle of token matching, instruction fetch, and execution. At the hardware level, this involves passing the token through sequences of separate hardware units between any two instructions. Typically, there are two levels of routing — local and global. If the two instructions exchanging a token reside in the same PE, the routing is local; in this case, the token passes through several hardware units and token queues within the PE. If they reside in different PEs, the routing is through the global interprocessor network. In machines implementing unraveling interpretation, the second major source of overhead is the matching store, necessary to collect tokens and determine if any two are destined for the same instruction.

In comparison, the passing of an operand from one instruction to the next in a von Neuman computer is accomplished through a simple memory store operation. This simplicity is a major factor in the efficiency of instruction execution. When a program contains significant amounts of potential parallelism, the overhead incurred in dataflow machines is tolerable, since different parts of the computation may be overlapped in time. When parallelism is low, however, performance necessarily degrades [GPKK82].

In this paper, we propose a different approach to executing dataflow programs which eliminates much of the overhead associated with executing each operation in isolation. We also describe a computer architecture capable of supporting this execution model.

2. The Execution Model

2.1. Sequential Code Segments

Any two operators connected by an arc in a dataflow program show a data dependency between the two operators. Consequently, the execution of one must necessarily precede the execution of the other in time. Since they can never be executed in parallel, there is no reason for mapping them onto two different PEs. When both are mapped onto the same PE, the operand produced by the first operator can be stored directly into the second, thus bypassing the normal cycle of token flow (matching store, communication network, etc.). This concept can obviously be extended to sequences longer than two operators. In general, any path that follows the arcs of a dataflow graph yields a sequence of operators, each of which is dependent on the completion of its predecessor and thus may be executed by the same PE.

The execution model proposed in this paper is based on the above observation. A given dataflow graph is broken into chains of operators which must be executed sequentially; these will be called *sequential code segments* or SCSs. Figure 4 shows one possible set of SCSs resulting from the program of Figure 1. Instead of treating individual operators of a dataflow graph as the smallest units of computation scheduled for execution whenever they are enabled, we increase the granularity to the level of SCSs. An SCS is *passive* as long as its *first* operator is disabled, i.e., is still missing some operands. A passive SCS resides on secondary storage. When all operands for the first operator have arrived, the SCS becomes *active*. It

Figure 4

Sequential Code Segments

is loaded as a whole into a contiguous portion of memory and it remains there until all operators constituting that SCS have been executed.

Each enabled SCS may be viewed as a very simple *process*. It has a "Process Control Block" (PCB) that contains the following information: (1) starting address of the SCS in memory, (2) a program counter pointing to the current instruction (the one to be executed next), and (3) a status field indicating whether the process is running, ready, or blocked. The three states are defined as follows. An SCS is said to be *running* when a PE is currently fetching and executing instructions from that sequence. An SCS is *ready* when its current instruction is enabled (has all its operands) but there is no free PE to execute that sequence. Finally, an SCS is *blocked* when its current instruction is not enabled.

Figure 5

Process State Changes

The possible state changes are illustrated in Figure 5. Initially, an SCS is loaded into memory in its ready state. Whenever a PE becomes free, it begins executing one of the ready SCSs in its memory; at that time, the status of the selected SCS changes from ready to running. The PE continues executing the SCS until it reaches the end, at which time the SCS is destroyed, or until it encounters an operator that does not yet have all its operands present. In the latter case, the SCS is blocked and the PE is switched to some other ready SCS. The blocked SCS changes its status to ready as soon as the last operand for the current instruction arrives.

This process-oriented point of view permits us to implement the execution of a dataflow program as a collection of *communicating SCSs*. A given program is transformed into one or more SCSs, which are mapped onto the available PEs. Each SCS continues executing as long as it has all the operands necessary to perform its current operation. When an operation produces a result token destined for a subsequent operation within the same SCS, it is stored directly in the appropriate operand slot using only one simple memory operation. Only when the token is destined for some other SCS does it have to travel through the dataflow routing network (within the same PE or to another PE) and pass through the matching store.

SCS1	0	-			*, 1(<i>l</i>)	
	1	+			*, 2(<i>l</i>)	
	2	/			...	...
SCS2	0	↑		2	*, 1(<i>l</i>)	
	1	-			*, 2(<i>l</i>)	
	2	√			*, 3(<i>r</i>)	SCS1, 1(<i>r</i>)
	3	-			*, 4(<i>l</i>)	
	4	/			...	...
SCS3	0	*		4	*, 1(<i>l</i>)	
	1	*			SCS2, 1(<i>r</i>)	
SCS4	0	*		2	SCS1, 2(<i>r</i>)	SCS2, 4(<i>r</i>)

Figure 6

Internal SCS Representation

Internally, an SCS is represented as a sequence of “machine-level” instructions, each consisting of an opcode, operand slots, and destination addresses for results. Figure 6 shows the same four sequences of Figure 4 in the form of a “machine program”. For simplicity, we assume that an instruction has only one or two operands; if only one is expected, the other is shown shaded in the figure. Constant operands do not travel through the graph but reside permanently in the appropriate instruction slots. For example, SCS4 comprises only one instruction with a constant operand 2 and a slot for the value of a .

A result token may be routed to one or two other instructions. Instructions are identified by the name of their SCS and the offset within that SCS. The letter l or r indicates which operand slot (left or right) the value should be placed in. For example, the one instruction constituting SCS4 sends its result to the right slot of the instruction in SCS1 at offset 2 (i.e., the division operator), and also to the right slot of the instruction in SCS2 at offset 4 (also a division operator). Asterisk denotes ‘itself’, i.e., the current SCS; for example, the first instruction of SCS1

stores its result into the next instruction in the same sequence (i.e., SCS1 at offset 1, left operand slot).

2.2. Translation of Dataflow Graphs into SCSs

In general, there will be more than one way to divide a given dataflow graph into SCSs. Different choices will obviously yield different performance results. For the purposes of this paper we ignore the issues of optimality and limit ourselves to only showing a simple method for translating basic non-iterative blocks and loop constructs into SCSs. We shall use the basic operators of Id (Irvine dataflow); similar methods could be developed for other dataflow base languages.

A simple non-iterative block statement is a directed acyclic graph consisting of unary and binary operators. An example of such a statement is the graph of Figure 1. Such a graph is translated into SCSs as follows: Starting with each incomming arc, traverse the graph by following arcs in their forward direction until reaching an exit arc; mark all operators encountered along the path. If an operator has more than one outgoing arc, choose one arbitrarily. The marked operators make up an SCS. If there are some unmarked operators after all incomming arcs have been tried, find those that have no unmarked operators leading into them. Starting with each of these, traverse the graph in the same manner as before. Stop when encountering an already marked operator or when reaching an exit arc. Each such sequence becomes another SCS. The algorithm stops when all operators have been marked. The SCSs of Figure 4 have been derived in this way from the graph in Figure 1.

Loop statements of Id always have the basic form shown in Figure 7. We translate each such (possibly nested) loop schema as follows. Starting with the inner-most loop, the loop body, including the operators L , d , d^{-1} , and L^{-1} , is translated into one or more SCSs using the algorithm described above. If the loop is nested inside another loop, the translation process continues in a similar manner.

Figure 7

A Loop Schema in Id

Each of the SCSs resulting from translating the inner loop is now treated as a *single* operator and the same principles are applied as with the inner loop. This process may be repeated until arbitrarily nested loops have been translated into SCSs.

3. The Machine Architecture

The proposed machine architecture is a collection of PEs interconnected through a communication network. While the choice of the network is important to ensure adequate performance, it has no bearing on the operational principles of individual PEs and hence need not be specified here.

The organization of each PE is shown in Figure 8. The major active functional units are a *fetch/execute* unit, a *matching store*, a *code segment loading* unit, and an *I-structure* unit. There are also three major passive units: an *execution memory*, a *token pool*, and a *program memory*. The following sections explain the functions of each of the units in more detail.

Figure 8

Organization of a PE

3.1. Activity Names and Token Matching

In a dataflow program, each activity (operator) has a unique identification called *activity name*. This provides the necessary matching information between tokens and their corresponding operators during execution. In the unraveling interpreter, each activity name is of the form $\langle u.c.s.i \rangle$, where the four component have the following meaning: i is the *iteration* number; if an operator is part of a loop, i designates which iteration the operator belongs to. (This component is absent in a feedback interpreter.) s is the *statement* number; this uniquely identifies an operator within a given procedure; c then identifies the *procedure*. Finally, u gives the current *context*; when a new procedure is called, the system records the activity name of the calling procedure in the context field u of the called procedure. The same is done when a new loop construct is entered. At the time the loop or procedure terminates, the original context is restored. In addition to the activity

name, which uniquely identifies each operator within a dataflow program, tokens destined for non-commutative binary (or n -ary) operators carry a tag called *port*; this designates the operand slot (left or right) into which the arriving token should be placed.

In our approach we adopt the same scheme of activity names as in Id. The only modification is to divide the statement number s into two components, $s1$ and $s2$, where $s1$ identifies an SCS and $s2$ is an offset within that SCS. Hence, $s1$ and $s2$ together identify a single operator in the same way as the statement number s .

Token matching is then based on the following two basic principles:

- As in any dataflow approach, two tokens are destined for the same operator if and only if their activity names show an exact match; the port tag distinguishes among multiple tokens in the case of binary or n -ary operators.
- If the activity names of two tokens differ in only the component $s2$, then both tokens are destined for the *same* SCS.

The operation of the matching store and the code loading unit of the proposed architecture is as follows (refer to Figure 8). Each entry in the matching store has three components: (1) a unique identification of an SCS, (2) a missing token count t , and (3) a pointer p to a list of tokens destined for that SCS. The identification of the SCS has the form $u.c.s1.*i$; i.e., it is an activity name where the component $s2$ matches any value, as indicated by the asterisk. The missing token count indicates how many tokens with $s2 = 0$, i.e., tokens destined for the first operator of the sequence are still missing. As long as this number is greater than zero, the SCS is *passive*. During this time, all tokens destined for that SCS are accumulated in the token pool in the form of a linked list; the pointer p points to this list. When the missing token count reaches zero, the SCS becomes *active*. This results in the creation of a new process for the SCS, which involves the following tasks: all tokens from the list pointed to by p are placed into the appropriate operand slots of the SCS; the SCS is loaded into execution memory and a "PCB" for the new process

is created and entered into one of the PCB registers of the fetch/execute unit; *p* is changed so that it points to the SCS in execution memory, rather than to the list of tokens in the pool; this list is now empty. The following algorithms specify in more detail the operations performed by the matching store and the code loading unit when a token arrives; we assume the token's activity name to be *u.c.s1.s2.i*:

Matching:

```
search the matching store associatively for the occurrence of
  an entry that matches the token's activity name;
if no match is found
then make a new entry in the matching store with the activity
  name u.c.s1.*.i;
  if s2=0 and the first instruction of the sequence is unary
  then go to Loading
  else store the token in the token pool and point p to it;
    store the number of tokens still outstanding for the
    first instruction into the missing token count t
else if s2=0 (i.e., entry exists and token is for first instr.)
  then decrement t;
    if t=0 then go to Loading
  else if t=0 (i.e., SCS is already active)
    then follow pointer p to find SCS in execution memory;
      place token into operand slot at offset s2;
      if this was the last operand missing
      then mark operation as enabled;
        if program counter points to that instruction
        then change status of SCS to ready
    else (i.e., SCS remains passive)
      append token to the list in the token pool
```

Loading:

```
get SCS for activity u.c.s1.*.i and load it into exec. memory;
store all tokens from the list pointed to by p in the appropriate
  operand slots in the SCS;
change p to point to SCS in execution memory;
create a new PCB comprising:
  (1) a pointer to the SCS's starting address
  (2) a program counter pointing to its first instruction
  (3) a status field with the value 'ready'
find a free PCB register and store in it the new PCB
```

Figure 9

Snapshot 1 of Execution

Example:

Consider again the four SCSs shown in Figure 6 and assume that the values b and c have arrived. Figure 9 shows a snapshot of the execution when all instructions enabled at that time have been executed. This execution involved the following steps: The arrival of b caused the sequences SCS1 and SCS2 to be loaded into memory. At the same time, two entries were made in the matching store with the respective activity names $u.c.SCS1.*.i$ and $u.c.SCS2.*.i$; the missing token count t is zero in both cases and the pointers point to the respective code segments in memory. SCS1 and SCS2 are ready and hence will start executing whenever a PE becomes free; both will become blocked after executing their first respective instructions, since the operands for the subsequent instructions are still missing. The program counters point to the instructions to be executed next.

In an analogous way, the arrival of c causes a new entry for the sequence SCS3 to be created in the matching store. However, since the first instruction ($*4$) is still missing its operand (a), SCS3 is not loaded into memory. Instead, the operand c

Figure 10

Snapshot 2 of Execution

is stored in the token pool and p is pointed at it. At this time, execution cannot proceed until the value of a has arrived (Figure 9 show this situation).

The arrival of a causes the remaining sequences SCS3 and SCS4 to be loaded, as shown in Figure 10. Note that SCS3 contains, in addition to the operand a , also the operand c , retrieved from the token pool. Both SCS3 and SCS4 are ready to execute. When SCS3 terminates, it stores the resulting value into SCS2 at the offset 1. This changes the status of SCS2 from blocked to ready and thus permits SCS2 to continue. SCS2 then generates the operand for SCS1, on which SCS1 is blocked, and so on. For the remainder of the execution, control is switched among the ready segments until all have terminated.

3.2. The Fetch/Execute Unit

This unit is equipped with a conventional processor responsible for executing active code segments in execution memory. As mentioned in Section 2.1, each active SCS is viewed as a simple process, that can be running, ready, or blocked. The “PCBs” of all processes are kept in an array of registers as shown in Figure 8.

Each PCB contains the program counter (PC), the status, and a starting address in execution memory. The execution cycle of the processor is as follows:

```
Begin: find the next ready process;
       fetch the instruction pointed to by PC;
Exec:  execute the instruction;
       for each resulting token tk do
         if activity name of tk differs in only the s2 field from
           the current activity
           then store tk directly in the current SCS at offset s2
           else send it to the routing unit;
       increment PC;
       if end of SCS is encountered
       then destroy process and go to Begin
       else fetch next instruction;
         if instruction is enabled
         then go to Exec
         else set process status to blocked and go to Begin
```

All tokens destined for an SCS different from the current one are sent to the *routing unit*. This unit determines, based on the token's activity name, which PE is holding the destination instruction. If this is the same PE as the current one, the token is sent back to the matching store; otherwise it is output to the network.

3.3. 1-Structure Storage

Dataflow systems use tree structures to represent arrays [DEN75]. A one-dimensional array is a tree of height one, a two-dimensional array is a tree of height two, etc. At each level, all subtrees are uniquely identified by a value called the selector. Two operators are provided to manipulate such structures. A *select* operator receives a structure S and a selector value i , and it returns the highest-level subtree of S labeled i , i.e., $S[i]$. An *append* operator must receive a structure S (possibly empty), a selector value i , and some other value v . It creates a new structure which is an exact copy of S except that the subtree labeled i has now the value v ; the latter could be a simple scalar value or another structure.

Conceptually, the entire structure S is always carried on a token between consecutive operators of the dataflow graph. To reduce the overhead in an actual implementation, the structure is kept in memory and only a pointer to it is carried on a token. In the architecture described in [ARIA83], the memory holding structures enforces the necessary synchronization between requests accessing the same structure, as these, in general, will not be arriving in the order of their generation. This memory is called the *I-structure memory*.

In the architecture shown in Figure 8, the same principles are applied to handle structured data. When an append operation is executed, it forms a request packet containing the pointer to the structure, the selector, and the value to be appended, and sends it to the appropriate I-structure storage (local or in some other PE) via the routing unit. The I-structure storage performs the actual append operation by creating the new structure and sending a pointer to it to the matching store, where it is eventually consumed by the logically next operator. Similarly, the execution of a select operator by the processor causes a request token to be routed to the appropriate I-structure storage which retrieves the value specified by the selector. When the value arrives at the PE, it is passed through the matching store directly into the SCS that issued the select operation.

4. Comparison of the Process-Oriented Model to Other Approaches

The proposed model has three main advantages over other approaches that implement data-driven computation:

1. The number of tokens passing through the routing mechanisms (both local and global) in the course of executing a dataflow program is reduced.
2. The number of token matchings that must be performed is reduced.
3. The overhead associated with executing each operator is reduced.

Consider first a non-iterative code segment such as the block statement in Figure 1. Let us distinguish between tokens destined for unary and for n -ary ($n > 1$)

operators. Under regular interpretation of dataflow programs, only tokens for n -ary operators must pass through the matching store; tokens for unary operators may be sent directly to the code fetch unit. Furthermore, all tokens, regardless of the arity of their operators, must pass through the routing mechanisms of the architecture (within each PE and possibly through the global communication network between different PEs).

	Other	Process-oriented
Tokens through Routing	17	10
Tokens through Matching Store	12	10

Table 1: Operators from Figure 1

Table 1 summarizes the token flow for the sample block statement. Under normal interpretation, “Tokens through Routing” is equivalent to the total number of tokens created in the course of executing the program (i.e., 17). In the process-oriented scheme, this number is reduced by those tokens stored directly into currently active SCSs in execution memory. Hence only 12 tokens need to be routed. “Tokens through Matching Store” corresponds, in the first case, to the number of tokens destined for n -ary operators (12). In the process-oriented scheme, all tokens that are not stored directly pass through both routing and the matching store. Hence the two values are always identical (10, in this example).

In general, a sequence of n binary operators requires $n + 1$ tokens to pass through the matching store – two for the first operator and one for each subsequent one. Other approaches always require all $2n$ tokens to pass through the matching store. The same ratios are obtained for the number of tokens that need to pass through the routing network, rather than being stored directly in the same SCS: with other approaches, all $2n$ tokens must pass through the routing network (local or even across different PEs); with the process-oriented approach, only $n + 1$ tokens

pass through routing; the remaining $n - 1$ are stored directly using one memory operation. Table 2 summarizes these results.

	Other	Process-oriented
Tokens through Routing	$2n$	$n + 1$
Tokens through Matching Store	$2n$	$n + 1$

Table 2: n Binary Operators

From the above it follows that, with long sequences, the reduction in tokens passing through the matching store as well as the number of tokens passing through the routing network approaches 50%. With short sequences, the reduction is less significant, however, it is never worse than in other architectures. In the worst case, when $n = 1$, the number of tokens passing through the routing network and through the matching store is the same (2 tokens) as with other approaches.

	Other	Process-oriented
Tokens through Routing	n	1
Tokens through Matching Store	0	0

Table 3: n Unary Operators

A sequence of n unary operators (Table 3) requires in both cases zero tokens to pass through the matching store, regardless of the value of n . The number of tokens through routing, however, is reduced from n to 1, i.e., the reduction is directly proportional to the sequence length. In the worst case, when $n = 1$, the reduction is zero, with $n = 2$ it is 50%, with $n = 10$ it is 90%, etc.

	Other	Process-oriented
Tokens through Routing	$n + n/2$	$1 + n/2$
Tokens through Matching Store	n	$1 + n/2$

Table 4: n Alternating Operators

Table 4 compares the number of tokens routed and tokens matched for a sequence of n alternating binary and unary operators. In terms of tokens passing

through the matching store, the results are similar to those of purely binary sequences: the reduction approaches 50% with large n and becomes zero with the shortest possible sequence of 2 operators. In terms of tokens passing through the routing network, the reduction is even more significant – approaching 66% with large n (and remaining zero for $n = 2$).

In addition to reducing the number of tokens passing through the routing network and the matching store, the process-oriented approach also reduces the overhead associated with fetching and executing instructions. In existing dataflow machines, each instruction is fetched, loaded, and executed individually whenever the necessary operands have been generated. With the process-oriented approach, entire sequences of instructions are loaded at once and each remains active until all instructions comprising that sequence have been executed. This facilitates the use of standard architectural techniques, such as caching, which are being applied successfully in von Neumann machines to increase their performance.

Since the information constituting the status of each active SCS (process) is quite minimal (a program counter, a memory pointer, and a status field), the switching of control among the different active sequences can be implemented with little overhead. The storing of operands into instruction slots, regardless of whether they are generated in the same PE or arrive through the communication network, is accomplished with one simple memory store operation. Consequently, efficiency in instruction execution withing each processor could approach the efficiency of a von Neumann machine executing a sequential program. The main advantage, of course, is that many such efficiently operating processors may be engaged simultaneously in executing a given dataflow program.

5. Conclusions

We have presented a scheme for efficient execution of dataflow programs and sketched an architecture capable of supporting this scheme. Our approach preserves

the basic principles of data-driven computation, where instructions are executed when and only when all of their input values have been produced and arrived at the operator. The expected benefit is a high degree of parallelism during execution, extracted from a high-level language program implicitly by transforming it into a dataflow graph. In addition, the proposed architecture attempts to reduce the overhead resulting in other dataflow architectures from three major sources: (1) routing of *all* tokens generated during the execution of a program, (2) matching of *all* tokens destined for *n*-ary operators in the matching store, and (3) the loading and execution of instructions on a one-to-one basis, which prevents the use of techniques such as caching. These problems are alleviated by loading into memory entire sequences of instructions that cannot be executed concurrently due to their data dependencies; the processor is multiplexed among the active sequences residing in its execution memory. Note that with a purely sequential code (which is generated when a given problem is of an inherently sequential nature), existing dataflow machines experience the same overhead in instruction execution as with highly parallel code. The proposed machine, on the other hand, "degenerates" to a von Neumann machine, that executes a regular sequence of code using only simple read/write operations in its local memory; no message passing is necessary in this case.

The price paid for the gained advantages is increased hardware cost, resulting from adding an execution memory to hold the active instruction sequences. This, however, is (at least partially) offset by a smaller matching store, since the number of activities is reduced — it corresponds to the number of distinct code sequences rather than the number of distinct operators, as is the case in other dataflow machines.

REFERENCES

- [ARGo82] ARVIND, GOSTELOW, K.P. The U-Interpreter. *Computer V15*, n2 (February, 1982), 42-49.
- [AGP78] ARVIND, GOSTELOW, K.P., PLOUFFE, W. An Asynchronous Programming Language and Computing Machine. In *Advances in Computing Science and Technology*, Ray Yeh, ed., 1978.
- [ARIA83] ARVIND, IANNUCCI, R.A. A Critique of Multiprocessing von Neumann Style. *The 10th International Symposium on Computer Architecture*, SIGARCH Newsletter V11, n3 (1983), 426-436.
- [CoSy80] COMTE, D., SYRE, J.C. The data-driven LAU multiprocessor system: results and perspectives. *IFIP Congress* (October, 1980), 175.
- [DEMi75] DENNIS, J.B., MISUNAS, D.P. A Preliminary Architecture for a Basic Data Flow Processor. *Proc. 2nd Int'l Symp. on Computer Architecture* (1975), 126-132.
- [DEN75] DENNIS, J.B. First Version of a Dataflow Procedure Language. *Mac Tech. Memorandum 61* (1975), MIT, Cambridge, Mass..
- [GKW85] GURD, J.R., KIRKHAM, C.C., WATSON, I. The Manchester Prototype Dataflow Computer. *CACM 28*, 1 (January, 1979), 34.
- [GPKK82] GAJSKI, D.D., PADUA, D.A., KUCK, D.J., AND KUHN, R.H. A Second Opinion on Data Flow Machines and Languages. *IEEE Computer* (February, 1982), 58-69.
- [WAGu82] WATSON, I., GURD, J.R. A Practical Dataflow Computer. *Computer* (Feb., 1982).