

DATA FLOW ON A QUEUE MACHINE[†]

Bruno R. Preiss

Department of Electrical Engineering
University of Toronto
Toronto, Ontario, Canada

V. C. Hamacher

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, Canada

Abstract

An execution model that supports program reentrancy, recursion, and automatic run-time loop unravelling is described. This execution model is based on queue machines that execute acyclic data-flow graphs. The use of separate instruction and data token spaces allows program reentrancy. Execution environments called contexts execute acyclic data-flow graphs associated with high-level code blocks. Iteration and function activation are implemented by the dynamic creation of contexts and do not require the use of tagged tokens.

A multiprocessor architecture that supports this execution model is proposed. The system architecture is based on a partitioned ring in which each partition of the ring is a conventional processor/memory bus.

The proposed architecture has been simulated in software. A number of test programs have been developed and their execution on the proposed architecture has been evaluated. The performance of the proposed architecture with various numbers of processing elements is described. In addition, a number of task scheduling algorithms are presented and evaluated.

1. Introduction

In this paper, we describe an approach to the organization of a multiprocessor system. First, an execution model that uses a prioritized queue to handle operands is described. It is shown that a certain class of data-flow program graphs can be used as generators of instruction sequences for queue machines.

Each program is subdivided into contexts, in order to facilitate procedure activation, iteration, and conditional execution. Contexts may execute in different processors. Mechanisms for context generation and intercontext communication will be described. The data-flow paradigm is used to support the automatic exploitation of medium-grain (context-level) parallelism.

We propose a multiprocessor architecture based on a partitioned memory and bus scheme. The bus partitions are connected in a ring topology. The ring is used both for data transfers as well as for context generation.

[†]This research was partially supported by NSERC (Canada) Grant A5192.

The proposed architecture has been simulated in software. A number of test programs were derived to demonstrate the performance of the architecture when executing programs containing certain specialized forms of parallelism. These simple programs were used to evaluate the performance of a number of context scheduling algorithms. Finally, the execution of two more complex programs containing the kinds of parallelism modelled by the simple programs was simulated.

2. Queue Machines

A queue machine uses implicit referencing to an operand queue in a manner analogous to the way a stack machine uses an operand stack. Operands for the current instruction are to be found at the front of an operand queue. Results of instructions are placed at the rear of the operand queue. This simple queue machine execution model is extended by associating with the result of each instruction a priority. The destination of the result is a position in the operand queue determined from the priority of the result. Thus, a result can be inserted anywhere in the queue. Operands are only deleted from the front of the queue.

A problem with the prioritized queue is that insertions in the middle of the queue require the relocation of a potentially large number of operands. However, this can be avoided. Let the operand queue occupy contiguous memory locations. The address of the front of the queue is held in a front of queue (*FOQ*) register. An operand fetch involves reading the contents of the memory location specified by the *FOQ* register and then incrementing the *FOQ* register. Instructions shall consist of two parts, an operation code part, and a result priority part. The result priority part consists of a number of destination offset fields (the result of an instruction can be inserted into the operand queue in more than one position). Each destination address for the result is computed by adding the destination offset field of the instruction to the contents of the *FOQ* register. Thus, the task of insertion into the operand queue has been simplified from that of searching a prioritized queue to that of performing a simple offset calculation. However, there is no guarantee that the result stored by one instruction does not overwrite that of an earlier instruction. Furthermore, there is no guarantee that there are no empty spaces in the queue. That is, there is no guarantee that the content of the memory location indicated by the *FOQ* pointer is a valid operand. A technique for resolving these difficulties will be given later.

As an example of the prioritized queue execution model consider the following instruction sequence, which computes the expression $(x + y) \cdot (x - z)$.

	Instruction	Destination Offsets
1	QUEUE: x	0,2
2	QUEUE: y	1
3	ADD	2
4	QUEUE: z	1
5	SUB	1
6	MUL	0

Assume that the queue is initially empty. The first instruction fetches x and places it into the operand queue at offsets 0 and 2 from the *FOQ*. The second instruction fetches y and places it in the queue at offset 1 from the *FOQ*. The third instruction removes two operands (x and y) from the front of the queue (incrementing *FOQ* each time), adds them, and stores the result at offset 2 from the new *FOQ*, which points at the second copy of x . Note that an empty place has been left in the queue. This space must be filled before an operand fetch from the space is attempted. The fourth instruction fetches z and stores it at offset 1 from the *FOQ*, filling the

empty space. The fifth instruction removes two operands from the front of the queue (incrementing FOQ each time), subtracts the second from the first, and stores the result at offset 1 from the new FOQ , which points at $x + y$. Finally, the sixth instruction removes two operands from the queue (incrementing FOQ each time), computes their product, and stores the result at offset 0 from the new FOQ . The result $(x + y) \cdot (x - z)$ is now the only element in the queue.

It has been assumed that the queue resides entirely in memory. Memory reference overhead can be reduced by using the concept of a sliding register window for holding a number of queue elements at the front of the queue.

Suppose there are n CPU registers, called *window registers*, that correspond to the first n elements of the operand queue. That is, the window registers are associated with the memory locations $FOQ, FOQ + 1, \dots, FOQ + n - 1$. As FOQ is incremented, the register window slides through memory. A front of window (FOW) register indicates which window register corresponds to the front of the operand queue, with the remaining $n - 1$ registers, enumerated in wrap-around order, corresponding to the next $n - 1$ queue elements. (If $n = 2^k$ for some k , then the FOW register is just the least significant k bits of the FOQ register). In order to indicate whether the operand contained in a window register is valid, each window register is augmented with a presence bit. When a result having destination offset d is computed, and d is less than n , the result is placed in the window register selected by $(FOW + d) \bmod n$ and the presence bit is set. If $d \geq n$, the result is stored in memory address $FOQ + d$. When an operand is to be removed from the front of the queue, the presence bit of the window register specified by FOW is examined. If it is set (the operand is in the window register), the operand is removed, the flag is cleared, and the FOQ and FOW registers are incremented. If the bit is not set (the operand is in memory), the operand is fetched and FOQ and FOW are incremented.

The window register bank acts as a cache for the operand queue. Values are not written through to memory. One might consider prefetching data from the memory into the window register bank. However, it is not possible to know whether or not a value fetched from the queue is valid. This problem can be resolved by using presence bits in memory as well as in the window registers. Such bits are required to support multiple contexts. This will be shown in a later section.

3. Executing Data-Flow Graphs on a Queue Machine

In the previous section, we introduced the prioritized-queue model of execution. An implementation requires that the code sequence correctly manages the operation of the queue. In this section, we show how a certain class of data-flow program graphs can be used to generate code sequences for a prioritized-queue machine.

A data-flow program graph is a directed graph in which the nodes correspond to instructions (called actors) and the arcs specify the movement of operands (tokens) [6]. In this section, we shall consider only acyclic data-flow program graphs composed of actors that produce results on all of their output arcs. Let an n -node graph consisting of actors A_1, A_2, \dots, A_n be represented by an $n \times n$ connectivity matrix C , where $C_{ij} = 1$ if there is a directed arc from A_i to A_j , and $C_{ij} = 0$ otherwise. Furthermore, define the predicate P_{ij} to be true if and only if there exists a path of directed arcs from node i to node j . It is easy to show that for $A_i \leq A_j$ iff P_{ij} and $A_i \leq A_i$, the relation \leq is a partial order.

It can then be proved that: Given an acyclic data-flow program graph, all (actor) instruction sequences $\{(I_1, I_2, \dots, I_n)\}$ suitable for execution on a prioritized-queue machine have the property that $A_i \leq A_j$ for all $i < j$. Therefore, acyclic data-flow graphs can be viewed as generators of valid instruction sequences for prioritized-queue machines.

There are two classical approaches to the representation of data-flow program graphs. In static data-flow architectures, the program is represented as a multiply-linked list of instruction cells, each of which contains an instruction and memory locations reserved for operand data-tokens. In dynamic data-flow architectures, instruction cells have no storage reserved for operands. Instead, each data token is tagged with information that associates it with an instruction cell and with other data tokens executing in the same context [13].

We propose a new data-flow program representation scheme which is based on instruction sequences for a prioritized-queue machine. An instruction cell shall occupy a range of addresses in two address spaces—instruction space and data token space. The operation code and destination offset parts of an instruction cell (which constitute a code sequence for a queue machine) occupy the instruction space. The data tokens associated with a given instruction cell are stored in data token space (which is the operand queue of a queue machine).

The layout of an instruction cell corresponds to the class of its corresponding actor in the data flow graph. We classify actors according to the number of input arcs and whether the arcs carry variable or constant data tokens. This classification of actors excludes the number of output arcs as this is a property of the data-flow graph and not of the actor itself. The classes of actor that we shall be dealing with are: one-operand actors; single-constant, one-operand actors; two-operand actors; and single-constant, two-operand actors. (We will also introduce a three-operand, single-constant actor later).

Consider a data-flow graph that evaluates the expression $\{(1-w) + 1\}/2$ (Figure 1a). A possible instruction sequence is shown in Figure 1b. The destination offsets were determined using an algorithm equivalent to that of a two-pass assembler.

The layout of an instruction cell depends on the class of the corresponding actor. A one-operand actor occupies one word of instruction space, and one word of data space. A single-constant, one-operand actor uses two words of instruction space, and none in data space. (Note the use of a colon to indicate that a constant follows). A two-operand actor uses one word in instruction space, and two words in data-token space. Finally, a single-constant, two-operand actor uses two words in instruction space, and one word in data space.

An important attribute of this method of representing data-flow graphs is the purity of the code. The separation of instructions from data permits reentrancy for code segments. In static data-flow architectures, program reentrancy is severely restricted due to the impurity of code. Complex mechanisms are needed to support reentrancy (e.g. see [11]). Dynamic data-flow architectures allow reentrancy at the cost of adding context-specifying tags to all operands (e.g. see [3]). In the queue machine model, the context of a data token is implicit in its memory address. Thus the overhead of tag manipulation is eliminated.

The restriction that data-flow graphs must be acyclic precludes the possibility of iteration. In addition, it has not yet been shown how subroutines may be invoked. Techniques for achieving these functions are discussed in the next section.

4. Contexts and Parallelism

In this section, we will describe a technique for implementing subroutines, conditional execution, iteration, and recursion based on the notion of contexts. The basis for the method is a mechanism called *dynamic data-flow program splicing*. It will be shown that this method can support parallelism at the context level. Thus, we place the emphasis on intercontext parallelism at the expense of intracontext parallelism.

A context shall consist of two address spaces: (1) an instruction space containing a queue machine instruction sequence corresponding to an acyclic data-flow graph, and (2) a data space containing an operand queue which holds data tokens. Note that there may be several contexts

associated with a particular instruction sequence. However, the data spaces of all contexts are unique.

Since the instruction sequence in a context is restricted to one that corresponds to an acyclic data-flow graph, only limited forms of computation are possible. In terms of high-level languages, such graphs correspond to sequences of assignment statements in a single assignment language. Conditional execution, iteration, and function activation can be implemented using special instructions for context generation and intercontext communications. These instructions connect the data-flow graphs in different contexts at execution time. Thus, we call the technique dynamic data-flow graph splicing. This technique is a generalization of the similar function activation mechanisms presented in both [1] and [4]. The advantage of our technique is the complete flexibility of the intercontext communications possible. The same mechanism is used for iteration, conditional execution and function activation. Functions can return partial results when available and can begin computing as soon as one argument is available.

We shall now introduce three new queue machine instructions—*FORK*, *SENDD*, *SENDA*—that are used to implement dynamic program splicing. The *FORK* instruction is a single-operand instruction. Its purpose is to create a new context. The *FORK* instruction expects as its single operand the address of an instruction sequence corresponding to an acyclic data-flow graph. It returns a pointer to the initial *FOQ* of the data space of a newly created context. The specified instruction sequence becomes the instruction space for the new context. The context executing the *FORK* shall be called the *calling* context, whereas the newly created context shall be called the *called* context. The *FORK* instruction is a non-standard data-flow actor in two senses: (1) It is not free from side-effect (it creates a new context when invoked); (2) it is not a function (each time it is invoked, a different context is created). The *FORK* instruction is not a system call since it does not cause a context switch nor does it involve operation system intervention.

The *SENDD* and *SENDA* instructions are used for intercontext communication of data. The *SENDD* instruction sends data between contexts. It is used to send arguments from a calling context to a called context; it is used to return results from a called context to the calling context. The *SENDD* instruction is a two-input instruction. The first operand is the address of the initial *FOQ* of a called context as returned by a *FORK* instruction. The second operand is a data item to be transferred between contexts. The destination offset fields are offsets from the initial *FOQ* in the operand queue of the called context at which the data item is to be stored.

The *SENDA* instruction is used to allow a called context to return a value to the calling context in a specified operand queue location. The *SENDA* transmits to the called context the address of the operand queue location in which the calling context expects a result to be stored. The *SENDA* instruction is a one-operand instruction whose input is the initial *FOQ* of a called context as returned by a *FORK*. The *SENDA* instruction has two destination offsets. The first is an offset from the initial *FOQ* in the called context. The second offset is added to the current *FOQ* in the calling context to compute the address that is sent to the called context.

We will now describe an example program that illustrates the basic mechanisms of dynamic data-flow program splicing. This example consists of a trivial function, namely $f(a) = a + 1$, and a main program that evaluates $f(5)$. A possible instruction sequence is shown in Figure 2. In this example, the program text is loaded starting at address 0. The main context data space begins at address 100. The *FORK* instruction generates a new context which begins executing instructions at address 6. We shall assume that the data space associated with this context begins at address 200. The *SENDD* instruction stores the constant 5 at memory location $200 + 0$. The *SENDA* instruction computes the address $102 + 0$ and stores it at $200 + 1$. The called context evaluates the addition and stores the result at address $201 + 1$. The *SENDD* instruction stores the result at address $102 + 0$. Finally, when a *HALT* instruction is reached, the context terminates its execution and releases the memory used for the data space of the context.

The previous example demonstrates the basic mechanisms involved in data flow-program splicing. The broken arcs in the data-flow graph of Figure 2a correspond to dynamically created arcs. These arcs carry intercontext data. Note also that the output arc from the *SENDA* actor is a virtual arc. Data is actually carried by a dynamic arc created in another context to the node specified by the virtual output arc from a *SENDA* actor.

Dynamic data-flow program splicing can also be used to implement conditional execution and iteration. The sample program of Figure 3 illustrates both iteration and conditional execution. This example introduces comparison instructions and the *SELECT* instruction. The comparison instructions produce a Boolean result which is placed in the operand queue. The *SELECT* instruction represents a new class of instruction. It has three inputs, the first of which is the select input. The result of the *SELECT* instruction is either the second or the third operand depending on whether the Boolean-valued select input is true or false, respectively. In Figure 3, the *SELECT* instruction is used to select the address of the loop body (i.e. address 8) or the address of the loop terminator (i.e. address 21) for input to the *FORK* instruction. Thus, a new context is created for each iteration of the loop. Similarly, a new context is generated for each branch of conditional execution. Finally, instructions 21 – 23 constitute a loop terminator sequence which introduces the *SKIP* instruction. The effect of *SKIP n* is to skip *n* operands in the operand queue. This is accomplished by adding *n* to *FOQ*. The loop terminator context sends the result computed in the last iteration of the loop back to the main context.

Since a new context is generated for each iteration of a loop, it is possible to execute several iterations of a loop in parallel. Thus, the context-based queue machine naturally supports automatic loop unravelling. This method of iteration differs from a recursive implementation of a loop in that when an iteration terminates, it executes a *HALT* instruction, releasing memory resources allocated to that iteration.

In order to synchronize the communication between contexts, a semaphore mechanism is required. We propose to use the usual data-flow technique of using presence bits. Associated with each memory location in data space is a presence bit. When a data space is allocated to a context, all the presence bits are cleared. The presence bit is set when an operand is stored in the operand queue. A context executing an operand fetch becomes blocked if the presence bit of the operand is not set. In a correct program, a context can only become blocked at instructions with dynamic input arcs. The blocked context becomes enabled for execution when some other context stores a data token at the blocking address.

A physical processor shall be responsible for the execution of many contexts. In order to organize this, it must maintain a list of enabled contexts and blocked contexts. The processor must detect when a blocked context becomes enabled for execution. This can be easily done using an associative context store. A context is completely specified by a program counter and an *FOQ* pointer. A context becomes blocked when the presence bit in the memory location specified by *FOQ* is not set. When a context blocks, its *PC* and *FOQ* are placed in an associative store. The entry is selected by the most significant $w - n$ bits of *FOQ*, where w is the number of bits in *FOQ*, and 2^n is the size of the data space associated with a context. The least significant n bits are used as the key in associative searching of the store. Since every data space is unique, there will never be a collision when inserting a blocked context into the context store.

Every time a data token is stored into a memory location by an intercontext *SENDD* or *SENDA* instruction, the set associative context store is searched. This is done by using the most significant $w - n$ bits of the data token address to select the set, and then comparing the least significant n bits of the data token address with the key stored in the set. If a match is found, the context becomes enabled for execution.

The execution model supported by this machine is like the communicating sequential process (CSP) model [13] in that contexts are explicitly enabled for execution by the *FORK*

instruction. However, once a context is activated, its execution is controlled by the availability of operands and processors (data flow). The data-flow paradigm is used here to support medium-grain parallelism.

This architecture has a number of similarities with the HEP machine [9]. Both architectures have tagged memory locations. The HEP architecture uses these tags to interlock pipeline stages. In the queue machine, the tags are used to synchronize contexts. The HEP processing element executes a number of tasks simultaneously, effectively doing a context switch on each instruction. The queue machine processing element executes within a context until it is blocked or finished. Both architectures provide special hardware for the management of context queues.

5. A Multiprocessor Queue Machine Architecture

The basis of the context-based data-flow execution model is the partitioning of the computation into a number of concurrently executing contexts, leading to an implementation based on a multiple processor architecture.

The context-based model of execution relies on two forms of communication. They are intracontext and intercontext communication corresponding to the flow of data tokens over static and dynamic arcs, respectively. Furthermore, most of the arcs in a data flow graph are static. This leads naturally to a two-level hierarchy of communications. A processor executing within a given context should have fast access to the data space associated with that context. However, the same processor must be able to establish dynamic arcs with any other context. Hence, the processor must be able to access all data token memory spaces. Since dynamic arcs occur less frequently, such cross-context references may proceed more slowly.

Finally, since each context can execute a *FORK* instruction, each processor should be able to create a new context with equal efficiency. This means there is no natural hierarchy among processors—they are all equal.

We propose a partitioned memory and bus scheme to provide a two-level memory hierarchy. Each processor has associated with it a memory and a processor bus. The processor buses are connected together in a ring topology by bus switches called node controllers (Figure 4). A processor has fast access to its own memory by means of its partition of the bus. However, a processor can also access the memories of other processors. The mechanism by which a processor accesses a remote memory is based on the transfer of a request from node controller to node controller. This organization is derived from two similar architectures. The first is the partitioned linear bus used in MP/C [2]. We use a ring topology to remove the processor hierarchy in MP/C. The second is a ring/bus multiprocessor called FERMTOR [10]. We have restricted each bus segment to have a single processor.

Each node controller is the master of its partition of the bus. The node controller grants the use of the bus to the associated processor. If the associated processor wishes to access a remote memory, it addresses the next downstream node controller and requests a memory access. The node controllers forward the request in the same manner until the request reaches the desired bus segment. The memory access is performed and a response travels around the ring to the requesting processor. A processor can have at most one request pending at any time. This guarantees that all processors have equal access to all memories. Furthermore, since the node controller is the master of the bus partition, a request is guaranteed to circulate (i.e., it cannot be indefinitely postponed).

In addition to memory references, the ring is used to carry *FORK* requests. When a processor executes a *FORK* instruction, it must select a processor according to some scheduling algorithm, and request the selected processor to create a context associated with a specified instruction sequence. The selected processor must acknowledge receipt of the *FORK* request, and it

must return to the processor executing the *FORK* the address of the data space of the newly created context. Since many contexts will be generated during the course of a computation, many *FORKs* will be executed. The scheduling algorithm must be very easily computed while still maintaining efficient processor utilization. We shall discuss suitable algorithms in the next section.

We now perform a simple derivation of the total execution time on a queue machine multiprocessor of a program consisting of a loop such as that of Figure 3. Let L be the time required to execute a single iteration of a loop. We claim that $L \approx \alpha_L + \beta_L n$, where α_L is the constant time to execute intracontext instructions, and $\beta_L n$ is the time to execute intercontext instructions (*FORK*, *SENDD*, *SENDA*), and n is the number of processors. Intercontext instructions require execution time proportional to n since they involve requests that circulate around the ring. (We shall ignore the effect of ring utilization in this simple analysis). Let D be the delay from the start of execution of one iteration of the loop until the start of execution of the next. Then, $D \approx \alpha_D + \beta_D n$. That is, α_D corresponds to the computation in a context before it is able to execute a *FORK* instruction, and $\beta_D n$ is the time required to execute the *FORK*. There are now two cases to consider: $n \leq \frac{L}{D}$ and $n > \frac{L}{D}$.

In the first case, loops unravel quickly enough to achieve continuous utilization of all processors. The average execution time for an iteration of the loop is $T \approx \frac{1}{n} L = \frac{\alpha_L}{n} + \beta_L$. Note that the execution time varies as $1/n$.

The second case occurs when loops do not unravel quickly enough. There are times during the computation when some processors are idle. The average time for an iteration of the loop is thus $T \approx \frac{1}{n} \cdot nD = \alpha_D + \beta_D n$. Note that the execution time varies as n . These results are shown graphically in Figure 5.

There are two important observations to be made from this analysis. First, when all processors are being fully utilized, execution time approaches a non-zero constant as processing elements are added. Second, eventually the interprocessor communication delays become the dominating factor in execution time. The minimum occurs for $n \approx L/D$. Thus, more processors can be utilized if the amount of computation in a context is increased or if the delay between the start of one context and the start of the next is decreased.

6. Simulation of the Proposed Architecture

A simulation of a multiprocessor queue machine was written in the *Concurrent EUCLID* language [8]. The purpose of the simulation was two-fold. First, The simulation was used to evaluate a number of simple context scheduling algorithms. Second, the simulation was used to examine the dependence of total execution time on the number of processors for certain representative kinds of parallelism. In the simulations, it was assumed that the instruction space was uniformly distributed among the processors and that each processor had access to all of the program text.

When a processor executes a *FORK* instruction, it selects a processor (possibly itself) to which it assigns the new context. Since the amount of computation within a context will be small (due to the acyclic nature of the graph), *FORKs* are expected to occur very frequently. Hence, the processor selection process must not have a great deal of overhead associated with it. We shall examine three selection algorithms that are designed to be amenable to simple hardware implementations.

The first algorithm we call *neighbour scheduling*. When a processor i ($0 \leq i < n$) executes a *FORK*, it selects processor $(i + 1) \bmod n$ (where n is the number of processors). The second

algorithm we call *cyclic scheduling*. Associated with each processor is the variable $last_i$. When a processor executes a *FORK*, it selects processor $(last_i + 1) \bmod n$, and then sets $last_i$ to the selected processor number. The third algorithm we call *least recent scheduling*. This algorithm takes advantage of the fact that *FORK* requests circulate around the ring of processors. When a processor i executes a *FORK*, it selects processor $(last_i + 1) \bmod n$. When a *FORK* request passes processor i , it assigns to $last_i$ the processor number specified in the request.

The behaviour of each of the scheduling algorithms was evaluated by simulating the execution of several test programs. Two of these test programs were specifically designed to evaluate the ability of the multiprocessor queue machine to support specific kinds of parallel context execution.

The first program, called *Loop*, was designed to evaluate the automatic loop unravelling performance of the proposed architecture. The program consists of a loop similar to that of Figure 3. A special instruction, *DELAY*, was included in the loop body as a simulator directive. When a processor executes the *DELAY* instruction, it delays the execution of the next instruction by the specified number of clock cycles (Figure 6a). The *DELAY* instruction was introduced to facilitate the simulation of a processor performing strictly intracontext computation requiring no intercontext communication nor intercontext synchronization. By setting the duration of the delay to zero, the minimum overhead associated with iteration can be determined. If the delay is large, loop unravelling should take place, reducing total execution time.

A graph of total execution time versus number of processors is shown in Figure 6b for the case of executing a 32 iteration loop with a delay of 32 clock cycles per iteration. Note that the shape of the curve is qualitatively in agreement with the simple analysis of the previous section. An exact correspondence cannot be expected, since the simple analysis ignores the effects of selection algorithms, the effects of the size of the window register set (8 in this case), and the effects of the detailed behaviour of the program.

The results obtained for the *neighbour* and *least recent* selection algorithms are identical. This is as expected for an unravelling loop program. The *cyclic* selection algorithm was less efficient than the other two. This is because a processor occasionally assigns a new context to itself. However, as the number of processors is increased, the performance of the *cyclic* scheduling algorithm approaches that of the other two. This is because the likelihood of a processor assigning a context to itself is inversely proportional to the number of processors.

The second test program, called *Split*, was designed to evaluate the performance of divide-and-conquer style, binary recursive algorithms on the proposed architecture. The main intent of the *Split* program is to generate as many contexts as quickly as possible. Again, a *DELAY* instruction is used to control the duration of execution of each context. Since the *Split* program was designed to maximize potential parallelism, it is a good test of the processor selection algorithms.

The expected performance is a strong function of the processor selection algorithm. Ideally, all processors will be active for the duration of the computation. This corresponds to the first case in the simple analysis of the *Loop* program. Thus, the expected total execution time is expected to behave like $\frac{\alpha}{n} + \beta$.

The results of simulating the *Split* program with argument 32 and a delay of 32 clock cycles are shown in Figure 7b. The *neighbour* algorithm has the poorest performance of the three selection algorithms. This is because each processor always selects the same neighbour processor. Since each call of the *Split* program calls itself recursively twice, this selection algorithm is inefficient. In particular, if the argument to *Split* is N , then at most $\log_2 N$ processors will be involved in the computation. Thus, for a number of processors $n > \log_2 N$, execution time will increase linearly due to added ring delay. This effect is evident in Figure 7b.

The best processor selection algorithm was the *least recent* algorithm. This selection algorithm exhibits the desired performance. That is, total execution time behaves like $\frac{\alpha}{n} + \beta$, approaching a broad minimum as the number of processors is increased. Eventually increasing the number of processors past the broad minimum results in increases in the total execution time.

We shall briefly discuss two additional programs that were simulated as practical programs exhibiting the kind of parallelism demonstrated by the *Loop* and *Split* programs.

The first of these, called *Integrate*, performs a numerical integration of a function using the trapezoidal rule. It is an example of a program containing a loop that can be automatically unravelled. The *Integrate* program is like the *Loop* program. Instead of a *DELAY* instruction, each iteration of the loop involves a non-trivial computation, namely a call to the function being integrated. Since several calls to the function can proceed in parallel, loop unravelling can take place. The performance of the *Integrate* program is expected to be similar to that of the *Loop* program. The essential difference is that the latter generates only one context per iteration, whereas the former generates two.

The execution time of the *Integrate* program is plotted against number of processors in Figure 8a. As in the case of the *Loop* program, the *cyclic* processor selection algorithm is least efficient. The performance of the *neighbour* and *least recent* selection algorithms is essentially the same for large numbers of processors. For small numbers of processors, *least recent* selection suffers because it does not take into account the duration of computation within a context. If the number of processors is even, half the processors execute the loop body while the other half evaluate the function activation. Since the two contexts don't take the same time, the computational load is unevenly distributed. On the other hand, if the number of processors is odd, the load is distributed evenly and execution time is reduced.

The last program we shall consider is a binary recursive *FFT* algorithm. This algorithm is based on the manipulation of streams of operands. Streams are used to implement data structures such as arrays. The implementation of streams is beyond the scope of this paper (see [5], [7], and [12]). We shall merely present the results of simulating the *FFT* program as an example of a non-trivial program that contains both iteration and binary recursion.

The results of computing a 32-point *FFT* for varying numbers of processors are shown in Figure 8b. Note that the behaviour of the *FFT* program is essentially the same as that of the *SPLIT* program. The important result is that for programs with a non-trivial amount of parallelism, total execution time varies with the reciprocal of the number of processing elements and approaches a constant as the number of processing elements is increased.

Finally, on the basis of these four programs, we conclude that the best selection algorithm of the three is the *least recent* processor selection algorithm. It produced the best results for both automatic loop unravelling as well as for binary recursive algorithms.

7. Conclusions

A multiprocessor architecture based on the queue machine execution model has been presented. Data-flow graphs are used both to generate valid instruction sequences and to support medium-grain parallelism. By associating different execution contexts with high-level language blocks, the program is partitioned into a number of tasks that can be distributed among the processors.

Simulation of the proposed architecture shows that even for problems of small inherent parallelism, significant reductions in execution time can be achieved (without reprogramming) by adding processors. The execution of model programs that exhibit specialized forms of parallelism was simulated in order to evaluate the performance of the multiprocessor system. Using these model programs, a number of processor selection algorithms have been evaluated.

REFERENCES

- [1] Makoto Amamiya, Ryuzo Hasegawa, Osamu Nakamura and Hirohide Mikami, "A List-processing-oriented Data Flow Machine Architecture," *AFIPS Conference Proceedings, 1982 National Computer Conference*, Vol. 51, June 1982, pp. 143-151.
- [2] Bruce W. Arden and Ran Ginosar, "MP/C: A Multiprocessor/Computer Architecture," *IEEE Transactions on Computers*, Vol. C-31, No. 5, May 1982, pp. 455-473.
- [3] Arvind and Kim P. Gostelow, "The U-Interpreter," *Computer*, Vol. 15, No. 2, February 1982, pp. 42-49.
- [4] L. J. Caluwaerts, J. Debacker and J. A. Peperstraete, "A Data Flow Architecture with a Paged Memory System," *Conference Proceedings of the 9th Annual Symposium on Computer Architecture*, April 1982, pp. 120-127.
- [5] L. J. Caluwaerts, J. Debacker and J. A. Peperstraete, "Implementing Streams on a Data Flow Computer System with Paged Memory," *Conference Proceedings of the 10th Annual Symposium on Computer Architecture*, June 1983, pp. 76-83.
- [6] Jack B. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science 19*, April 1974, pp. 362-376.
- [7] Jack B. Dennis and Ken K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," *Proceedings of the 1979 International Conference on Parallel Processing*, August 1979, pp. 35-45.
- [8] R. C. Holt, *Concurrent Euclid, The Unix System, and Tunis*, Addison-Wesley, Reading, Massachusetts, 1983, 323 pp.
- [9] Harry F. Jordan, "Performance Measurements on HEP—A Pipelined MIMD Computer," *Conference Proceedings of the 10th Annual Symposium on Computer Architecture*, June 1983, pp. 207-212.
- [10] Wayne M. Loucks, *FERMTOR: A Flexible Extendible Range Multiprocessor*, Ph.D. Thesis, University of Toronto, Department of Electrical Engineering Computer Group, Toronto, 1980, 188 pp.
- [11] Glen Seth Miranker, "Implementation of Procedures on a Class of Data Flow Processors," *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977, pp. 77-86.
- [12] Bruno R. Preiss, *Design and Simulation of a Data-Flow Multiprocessor System*, M.A.Sc. Thesis, University of Toronto, Department of Electrical Engineering Computer Group, Toronto, 1984, 164 pp.
- [13] Philip C. Treleaven, David R. Brownbridge and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, Vol. 14, No. 1, March 1982, pp. 93-143.