

# Semi-Static Dataflow

*Bruno R. Preiss*

Department of Electrical Engineering  
University of Waterloo  
Waterloo, Ontario, CANADA, N2L 3G1

*V. Carl Hamacher*

Computer Systems Research Institute  
University of Toronto  
Toronto, Ontario, CANADA, M5S 1A4

## ABSTRACT

In this paper we present a new dataflow execution model called semi-static dataflow. This model incorporates aspects of conventional static and dynamic dataflow architectures. Programs are partitioned into a collection of dataflow graphs. The execution of each of these graphs is the responsibility of a low-level process called a *context*. The static dataflow execution model is used to evaluate each of these graphs. Separate instruction and data spaces are used to allow program reentrancy. Function invocation, iteration, and conditional execution are accomplished by dynamically creating new contexts.

The process of creating new contexts and moving data tokens between contexts is called *dynamic dataflow graph splicing* and is the motivation for calling the whole system semi-static. We present a number of programming paradigms for function invocation, sequential iteration and parallel iteration that are based on dynamic dataflow graph splicing.

We have simulated the execution of a semi-static dataflow multiprocessor. In this paper some of the simulation results obtained for several benchmark programs are presented.

July 5, 1994

# Semi-Static Dataflow

*Bruno R. Preiss*

Department of Electrical Engineering  
University of Waterloo  
Waterloo, Ontario, CANADA, N2L 3G1

*V. Carl Hamacher*

Computer Systems Research Institute  
University of Toronto  
Toronto, Ontario, CANADA, M5S 1A4

## 1. Introduction and Motivation

Dataflow architectures are classified as either static or dynamic. Static and dynamic architectures differ in the following ways: First, in static architectures, the program graph is loaded into memory in completed form before the program begins execution (i.e., no run-time loader); whereas in dynamic architectures, nodes can be created at run time (e.g., to support loop unravelling and recursion). Second, in static architectures, at most one instance of an actor may be enabled for firing at a time. Dynamic architectures support several instances of an actor firing simultaneously[1]. Finally, static architectures use the same storage space for instructions (actors) and data (tokens) (i.e., impure code). Dynamic architectures use physically separate memories for instructions (i.e., pure code) and data[2].

In static dataflow architectures, the dataflow program graph is represented as a multiply-linked collection of items called *activity templates*[3]. Each activity template corresponds to an actor of the dataflow program graph. An activity template is a triple consisting of i) an operation code, ii) a set of operand slots, and iii) a (destination) pointer list. The operand slots correspond to the input arcs of the actor. Each slot is a reserved memory location into which one of the actor's predecessors stores a token. Thus, arcs have a maximum token-carrying capacity of one. The elements of the destination pointer list correspond to the output arcs of the actor. These pointers indicate the operand slots in other activity templates into which the result of this actor is to be stored. The operation code field of the actor specifies the function computed by the actor. It also directly or implicitly specifies the number of operand slots, the number of destination pointers, and the firing rule for the given actor. The basic execution cycle involves: identifying an activity template whose firing rule is satisfied (i.e., all its operand slots are full and the operand slots of its successors are empty); computing the function specified by the operation code; and storing the result in the operand slots specified by the destination pointers.

The finite arc capacity and impurity of code in the static dataflow systems result in dataflow program graphs that are not reentrant. This constraint affects both iteration and function (subroutine) calling mechanisms. Automatic run-time loop unravelling is not supported by static dataflow systems. Functions can be made reentrant by either code-duplication [4] or code-sharing mechanisms[5].

In dynamic dataflow architectures, dataflow program graphs are also represented as a collection of multiply-linked activity templates. In this case, activity templates consist of i) an operation code and ii) a (destination) pointer list. No space is reserved in the activity template for storing (input) data tokens. Instead, tokens are stored elsewhere. Each token is tagged with information that identifies the arc on which the token conceptually resides. Thus, a token consists of a data field and a tag field. As a result, the token-carrying capacity of the arcs is not constrained. As in static architectures, the elements of the

destination pointer list correspond to the output arcs of the actor. These pointers specify the address of the successor activity template and also indicate to which input of the actor the token is to be sent. (In general, the functions computed by the actors need not be commutative). In addition, the pointer may also specify how many tokens are required by the successor actor in order to fire (to facilitate efficient evaluation of its firing rules). The basic execution cycle involves locating actors whose input arcs are full (by matching tag fields of tokens); computing the function specified by the operation code (using the value fields of tokens); and forming the value parts of the output tokens from the result of the computation and the tag parts of the output tokens from the elements of the destination list.

Since there are no operand slots in the activity templates, the code is pure. Furthermore, as explained above, the token-carrying capacity of arcs is not constrained by the representation of the dataflow graph. As a result, loop unravelling and reentrant functions are easily implemented in dynamic dataflow architectures.

To support reentrancy and unravelling, the tag field of tokens is augmented with information that identifies the context in which the token is executing. The augmented tag field is called an *activity name*. The activity name specifies i) the actor (and which of its inputs) to which the token is destined, ii) the iteration number of the loop to which the token belongs (if it is inside a loop), and iii) the activity name of the calling function if the token belongs to a called function[6]. Note that the activity name is recursively defined. In effect, every token carries with it information that is analogous to the processor stack in conventional, Von Neumann architectures.

Iteration and function calls are accomplished in dynamic architectures by using actors that manipulate the tag fields of tokens[7]. For example, iteration is accomplished using an actor that takes tokens from the bottom of a loop, increments the iteration number field of the activity name of the token, and injects the token back into the top of the loop. Arguments are transmitted to functions and results are received from functions using actors that, in effect, push and pop contexts from activity names.

In this paper we present a new dataflow execution model that attempts to exploit the capabilities of both static and dynamic dataflow architectures without incurring the tag matching overhead of the dynamic model or the difficulties arising from the impure code of the static model. We call this execution model semi-static dataflow. In semi-static dataflow architectures, the dataflow program graph is represented as a multiply-linked collection of activity templates. As in static architectures, the activity templates consist of i) an operation code part, ii) operand slots, and iii) destination pointers. Semi-static architectures differ from static ones in that the activity template is not stored in contiguous memory locations. Instead, the operation code and destination pointers are stored in an instruction space and the operand slots are stored in a data space. By associating several data spaces with one instruction space, a graph can be made reentrant. The advantage of this scheme is that reentrancy is accomplished without the overhead of code copying or tagged tokens. Furthermore, since there are fixed memory locations for data token storage, tags are unnecessary. Consequently the tag matching program has been eliminated.

Typically, a dataflow program consists of a collection of separate dataflow graphs (e.g., a separate graph is constructed for each procedure). Each instance of the evaluation of a dataflow graph requires a new data space. In order to simplify the management of memory resources, we restrict the size of an instruction or data space to the size of a memory page (e.g., 1 kbyte). A separate “process” is invoked to evaluate each {instruction space, data space} pair. We call these processes *contexts* to emphasize their small size. (We reserve the term *process* for the execution of a family of related contexts. I.e., the term *process* carries its usual meaning.)

## 2. Contexts

A context is a small to medium granularity process. A context evaluates a static dataflow graph. The execution of a program requires the dynamic creation and execution of many contexts. In order to perform useful computation, these contexts must communicate. We have adopted a simple communications model based on the use of unidirectional communication channels.

A context requires (1) a static dataflow graph, (2) a data token space, and (3) a pair of communication channels called the *input* and *output* channels of the context. A context receives data tokens over the input channel, evaluates its dataflow graph using the data token space for token storage, and sends its results over the output channel.

In dynamic dataflow machines, conditional execution, iteration, and function invocation are accomplished using special actors that alter the tags associated with tokens. In effect these actors change the “context” in which the tokens are executing. By analogy, in semi-static dataflow, conditional execution, iteration and function invocation are all accomplished by the dynamic creation of contexts.

The essence of the semi-static dataflow approach is the partitioning of the program into a collection of dataflow graphs. Each of these graphs can be evaluated using the static dataflow execution model. That is, data token storage addresses are statically allocated for each of these dataflow graphs. By associating several virtual data spaces with a single graph, the dataflow graph can be made reentrant. These associations are established dynamically as the program is being executed. Each of these associations is executed by a low-level process called a context. Special actors are used for the creation of contexts and for the communication of data tokens between contexts. Thus the semi-static dataflow approach is a cross between the static and dynamic dataflow execution models

## 3. Communication Primitives

Communication between contexts is accomplished over communication channels. A channel is an abstract entity whose purpose is to provide a unidirectional communications path between two contexts. Communication channels can be established in two ways:

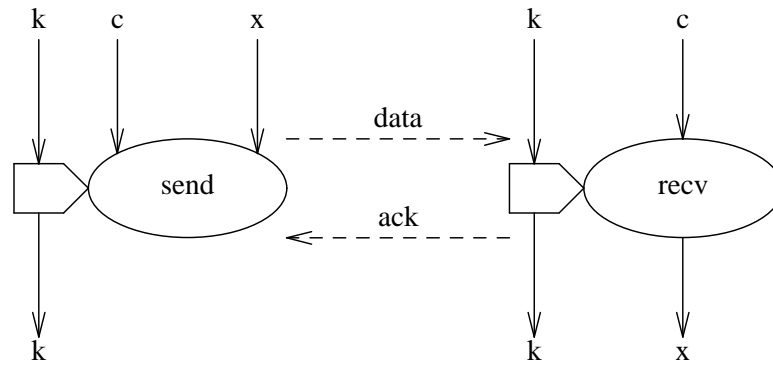
- Communication channels can be opened dynamically during program execution. These channels are created when contexts are created to provide the necessary communication paths between parent and child contexts.
- Communication channels can be allocated statically at compile time. These are channels explicitly declared and used by the programmer.

A channel is used to transfer a fixed length message from one context to another. Each message can carry exactly one (scalar) data token. Consequently, the length of the data portion of a message is equal to the word size of the machine.

Channels use an unbuffered, synchronous communications strategy sometimes called a rendezvous. Intercontext communication is accomplished using two simple primitives called *send* and *receive*. Each channel has a unique channel identifier. This identifier is used by channel primitives to select the desired communications channel. This is essentially the mechanism used in the Occam language for interprocess communication[8], and will be discussed later in relation to our simulation study.

The dataflow actors used to represent the two communication primitives are shown in Fig. 1. The send actor is a three-input, one-output dataflow actor. The first operand, *k*, is a control token. It is used to sequence communications primitives. Its use will be described below. The second operand, *c*, is a channel identifier. It specifies the channel to use for the communication. The third operand is the value, *x*, to be transmitted on the channel. The output of the send actor, *k*, is a copy of the input control token.

The receive actor is a two-input, two-output dataflow actor. The first operand, *k*, is a control token. The second operand, *c*, is a channel identifier. The first output of the receive actor, *k*, is a copy of the input control token. The second output of the receive actor is the value, *x*, received on the channel.



**Fig. 1. Communication primitive dataflow actors.**

The send and receive actors are non-standard dataflow actors in two senses:

- They are not free from side effects. Both send and receive affect the execution of other contexts.
- They are not strictly functional in the mathematical sense. That is, the result of their execution is not strictly a function of their operands.

Consequently, the order in which send and receive actors are executed will affect the outcome of the computation. To ensure deterministic results, these actors use control tokens. The sole purpose of these tokens is to sequence actors having side effects. The send and receive actors only produce control tokens on their output arcs after successful communication has occurred. This involves the exchange of two messages between contexts. The first message carries the input data token from the send actor to the receive actor. The second message acknowledges receipt of the data token.

The effect of the execution of a {send, receive} pair is the establishment of dynamic arcs between the two actors. These arcs must be established dynamically since, in general, it is not possible to predict which two send and receive actors in a given program will communicate. This uncertainty can arise in two ways. First, the channel identifier used by a primitive need not be statically determined. Second, the invocation of a particular instance of a communication primitive may be conditionally determined. The net effect of the dynamic establishment of arcs is that separate dataflow graphs are “spliced” together at execution time. For this reason, we call this technique dynamic dataflow graph splicing.

#### 4. Function Invocation by Dynamic Dataflow Graph Splicing

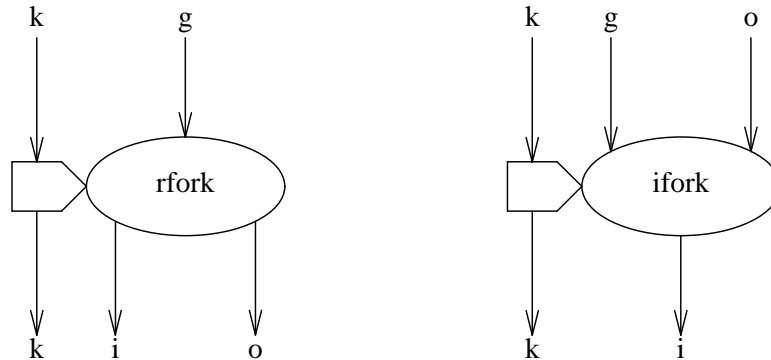
In this section, we describe a function invocation paradigm based on the method of dynamic dataflow graph splicing. The function invocation paradigm consists of four phases of execution: context generation, parameter passing, concurrent execution, and result passing.

##### 4.1. Context Generation Phase

In the context generation phase, the calling context (i.e., the parent context) creates a new context for the execution of the function. We have currently defined two primitives for the generation of contexts. These primitives are called recursive fork (rfork) and iterative fork (ifork). Function invocation is accomplished using the rfork primitive. (The use of the ifork primitive to implement sequential iteration is described in a subsequent section.)

The dataflow actors used to represent the context generation primitives are shown in Fig. 2. The rfork actor is a two-input, three-output dataflow actor. The first operand is a control token,  $k$ , which is used to sequence actors with side effects. The second operand is a pointer to the dataflow graph to be evaluated by the child context (i.e., the entry point of the subroutine). The effect of the rfork actor is to generate a new context together with two new channels. These channels become the input and output channels of the child context. The first output of the rfork actor is a control token,  $k$ , which is a copy of

the input control token. The second and third outputs of the rfork actor, *i* and *o*, are the channel identifiers of the input and output channels of the child context. The input channel is used by the parent context to send data tokens to the child context. The output channel is used by the child context to send data tokens to the parent context.



**Fig. 2. Context generation primitive dataflow actors.**

The ifork actor is a three-input, two-output dataflow actor. The first operand is a control token, *k*, which is used to sequence actors with side effects. The second operand is a pointer to the dataflow graph to be evaluated by the child context. The third operand is a channel identifier. The effect of the ifork actor is to generate a new context together with one new channel. This channel becomes the input channel of the child context. The output channel of the child context is specified by the third operand of the ifork actor. The first output of the ifork actor is a control token, *k*, which is a copy of the input control token. The second output of the ifork actor, *i*, is the channel identifier of the input channel of the child context.

#### 4.2. Parameter Passing Phase

In the parameter passing phase, the parent context sends the parameter values to the child context. The parent context does this by sending data tokens over the child context's input channel. The channel identifier for this channel is obtained as a result of the rfork or ifork primitive.

In the event that the child context requires more than one parameter, those parameters must be transmitted to the child sequentially. Consequently, the parent and child contexts must use a prearranged parameter sequence to ensure correctness. This sequence can be specified statically at compile time.

#### 4.3. Concurrent Execution Phase

After the child context has received all its parameter values from the parent context, the concurrent execution phase begins. During the concurrent execution phase, both the parent and child contexts may execute in parallel.

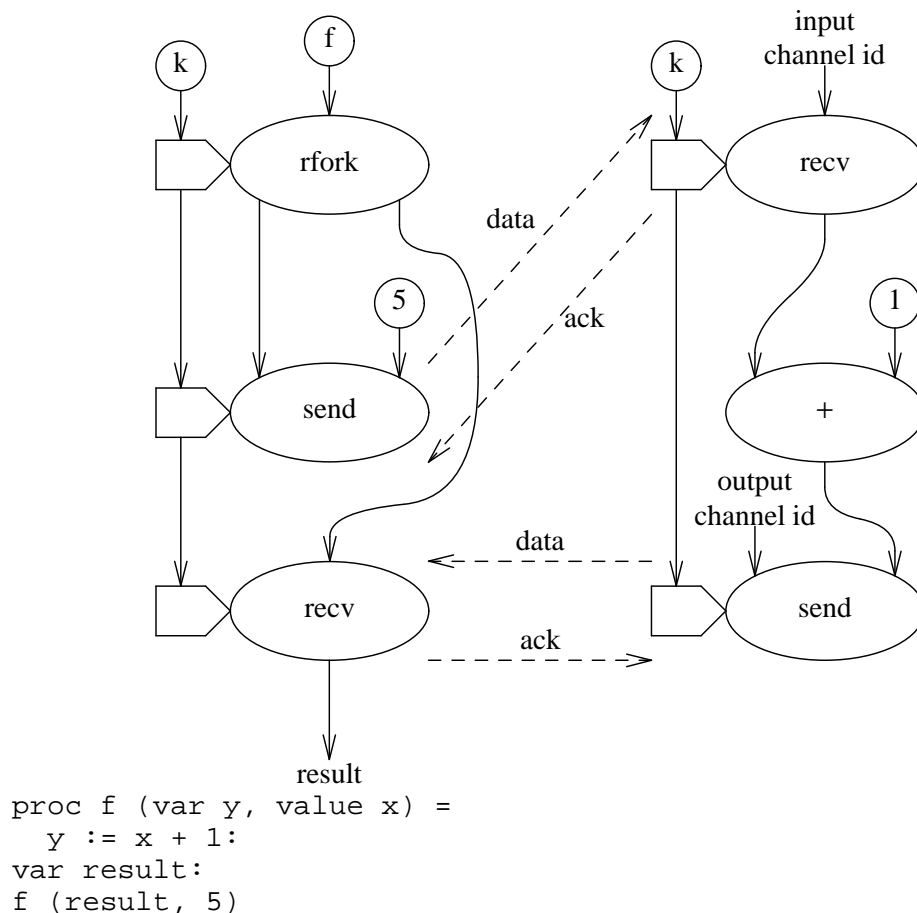
#### 4.4. Result Passing Phase

During the result passing phase, the child context sends its computed results back to the parent context. The child context does this by sending data tokens over its output channel. The parent context can receive the result data tokens using the channel identifier returned by the rfork actor or specified to the ifork actor.

In the event that the child context produces more than one result, those results are returned in a predetermined sequence just as in the parameter passing phase.

#### 4.5. Example

The dataflow graphs of Fig. 3 illustrate the basic function invocation paradigm. This example consists of a trivial function, namely  $f(x) \leftarrow x+1$ , and a main program that evaluates  $f(5)$ . The main program has three actors: (1) The rfork actor creates a new context to execute the graph associated with the function  $f$ . (2) The send actor transmits the argument, 5, to the new context. (3) the receive actor receives the result from the called context. The graph corresponding to the function  $f$  has three actors: (1) The receive actor receives the arguments to  $f$  on the input channel. (2) The + actor adds 1 to the argument. (3) The send actor transmits the result of the function to the parent context via the output channel. The dashed arcs in Fig. 3 correspond to the dynamically created arcs that splice the two dataflow graphs together.



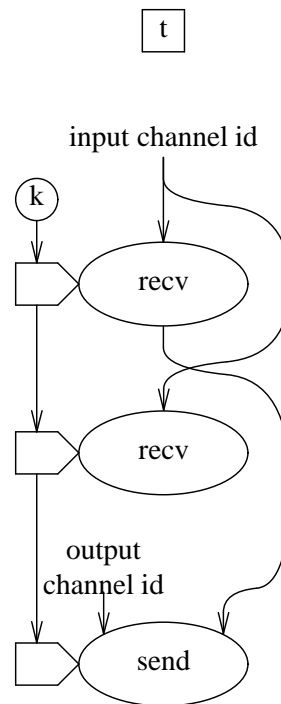
**Fig. 3. Basic function invocation paradigm.**

#### 5. Sequential Iteration

In this section we describe an iteration paradigm based on the method of dynamic dataflow graph splicing. The essential feature of this paradigm is the creation of a separate context for each iteration. This is done to allow for dynamic loop unravelling, i.e., to allow the parallel evaluation of multiple iterations.

The program of Fig. 4 illustrates the iteration paradigm. The iteration paradigm requires three separate dataflow graphs that are spliced together at execution time. These graphs are: the main program graph (m), the loop body graph (b), and the loop terminator graph (t).





```

var sum, result:
seq
  sum := 0
  seq i = [1 for 10]
    sum := sum + i
  result := sum

```

**Fig. 4. Sequential iteration paradigm.**

### 5.1. Main Program Graph

The main program graph uses the basic function invocation paradigm described in the preceding section to initiate the execution of a loop. In effect, the main program graph calls the loop body graph as if it is a function. The main program uses the rfork primitive to create the new context. The main program graph then transmits to the loop body the value of the loop counter and any values used in the body of the loop. In the example of Fig. 4, the main program graph invokes the loop body and transmits the initial values of the variables *sum* and *i*. A consequence of using the basic function invocation paradigm is that the main program graph may execute concurrently with one or more loop iterations.

### 5.2. Loop Body Graph

The loop body graph performs the calculations involved in a single iteration of the loop. It then tests the termination condition and creates a new context. This new context will execute the next iteration of the loop if the termination condition is not satisfied. Otherwise, it executes the loop terminator graph.

The loop body graph uses the ifork primitive to create the new context. Recall that the ifork primitive passes on the specified output channel to the child context. This output channel is the channel over which the main program graph expects to receive its results. In this way, the final iteration of the loop can return its results directly to the main program graph without having to pass through all the intermediate loop iterations. This is similar in effect to tail recursion. The advantage of this approach is that it allows loop unravelling yet at the same time permits the recovery of resources allocated to earlier iterations as they terminate.

In the example of Fig. 4, the loop body receives the values of *sum* and *i*. It computes new values for *sum* and *i* and tests the loop termination condition. If the loop termination condition is not satisfied, it invokes a new instantiation of the loop body graph. If the loop termination condition is satisfied, it invokes the loop terminator. In both cases, it transmits the new values of *sum* and *i* to the child context.

### 5.3. Loop Terminator Graph

The loop terminator graph is used to return the results from the final iteration of the loop back to the main program graph. The loop terminator graph is invoked by the loop body graph when the loop termination condition has been satisfied. Note that the loop body graph and loop terminator graph must have the same call format. That is, the sequences in which the arguments are transmitted must be identical. However, the loop terminator graph will discard most of the input values except for the final results. These results are transmitted via the output channel. Since the output channel has been inherited from preceding loop iterations, the values transmitted on this channel are returned directly to the main program graph.

In the example of Fig. 4, the loop terminator receives the values of *sum* and *i*. It discards the value of *i* and returns the *sum* to the main program graph. The value received by the main program graph thus becomes the result.

## 6. Dynamic Process Creation by Parallel Iteration

In this section we describe a modification of the sequential iteration paradigm described above which allows the dynamic creation of parallel processes (or context families). It is assumed that these processes are to execute concurrently. Furthermore, any synchronization or interprocess communication required uses channels explicitly declared by the programmer.

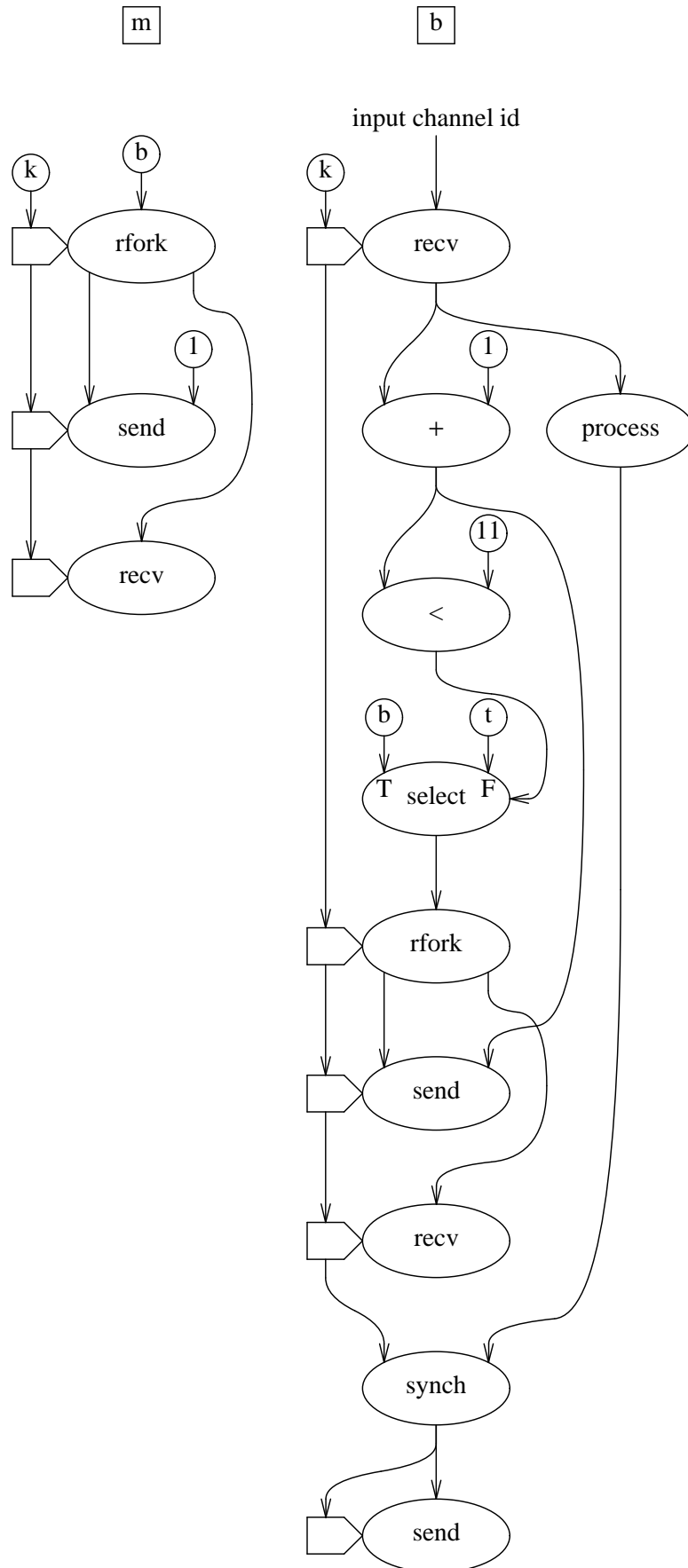
The program of Fig. 5 illustrates the parallel iteration paradigm. The structure of this program is very similar to the sequential iteration paradigm. Each iteration of the loop is responsible for the execution of one process. This technique exploits the concurrent execution phase of the basic function invocation paradigm to achieve the parallel execution of a number of processes.

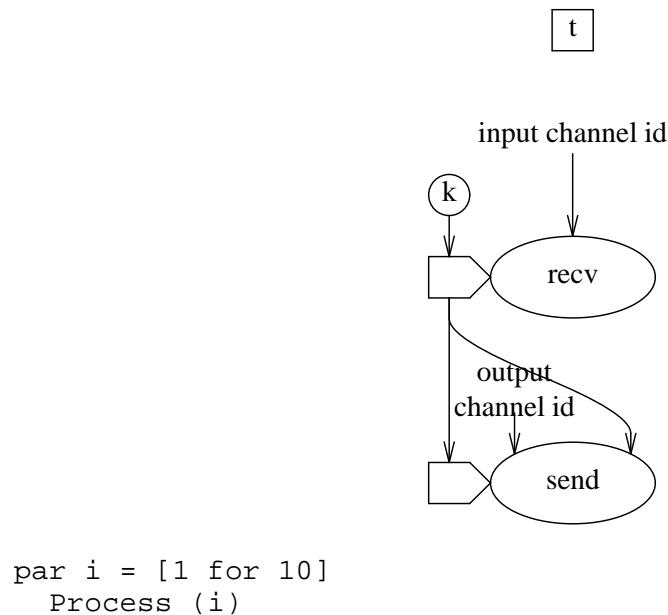
There is an important difference between the sequential iteration and parallel iteration paradigms. Whereas in the sequential iteration paradigm the main program context only needs to synchronize with the final iteration of the loop, in the parallel iteration paradigm the main program graph needs to synchronize with all iterations. This is required to preserve the “par” semantics. That is, subsequent computation in the main context cannot proceed until all parallel process instantiations have terminated.

Synchronization is explicitly accomplished by message passing. Each iteration *i* waits until its process instantiation has completed and until it has received a message from iteration *i*+1 indicating that all higher number iterations have completed. When both conditions have been met, iteration *i* returns a message to iteration *i*-1 indicating that it has completed. In effect, parallel iteration is analogous to conventional recursion except that all instantiations proceed in parallel.

## 7. Performance Analysis

In order to characterize the potential performance of the semi-static dataflow execution model, we have designed a shared-bus multiprocessor architecture called a *Queue Machine Multiprocessor* which uses the semi-static execution model[9]. In addition, we have implemented a preliminary run-time environment that supports the execution of programs compiled using a prototype Occam compiler. In this section we will present the results obtained by simulation of the execution of a number of benchmark programs.





**Fig. 5. Parallel iteration paradigm.**

### 7.1. Architectural Model

The Queue Machine Multiprocessor is a shared-bus multiprocessor which consists of a number of processing elements each with their own local memory connected by a shared interprocessor communication bus. The processing elements in this system are called *Queue Machines*[10]. A queue machine is a processing element that has been optimized for the efficient evaluation of static, acyclic dataflow graphs[11]. In contrast to typical dataflow processing elements, a queue machine has a single locus of control. However, we have shown that a queue machine can efficiently utilize a pipelined ALU to exploit the parallelism in a static, acyclic dataflow graph (intracontext parallelism).

We have also implemented a *Queue Machine Multiprocessing Kernel* for the Queue Machine Multiprocessor[9]. This is a collection of software routines in which queue machine programs generated by the Occam language compiler described below can be executed. The purpose of the kernel is to manage contexts (i.e., small processes) and resources (memory and channels). In particular, this kernel provides the context generation primitives `rfork` and `ifork`.

### 7.2. Occam Compiler

The benchmark programs used in this performance analysis are written in the Occam programming language[8]. We have constructed a prototype compiler that translates Occam source programs into dataflow graphs for execution on the Queue Machine Multiprocessor.

The compiler uses the basic function invocation, sequential iteration, and parallel iteration paradigms described above. It automatically partitions the source program into multiple contexts for parallel execution. The current partitioning algorithm is very simple (and, consequently, suboptimal). The compiler generates a separate dataflow graph for every procedure, for every outcome of a conditional branch, and for the body of every loop (sequential and parallel iteration). (The compiler also constructs loop terminator graphs as needed.)

The Occam compiler automatically emits the context generation and intercontext communication primitives as required by the function invocation, and sequential and parallel iteration paradigms. In order to emit the intercontext communication primitives, the compiler must choose a sequence of the input arcs to each dataflow graph. The compiler automatically chooses that sequence which maximizes

the amount of computation possible within a context before another input is required. This heuristic has been found, experimentally, to produce good overall throughput results on the multiprocessor[9].

### 7.3. Benchmark Programs

We have simulated the execution of a number of benchmark programs on the Queue Machine Multiprocessor. In this paper we present the results for four different Occam programs. The tasks performed by the benchmark programs are: 1) Matrix Multiplication, 2) Fast Fourier Transform, 3) Cholesky decomposition, and 4) Congruence transformation.

#### 7.3.1. Matrix Multiplication

The Matrix Multiplication program computes the product of two  $N \times N$  matrices.

#### 7.3.2. Fast Fourier Transform

The Fast Fourier Transform program computes the discrete Fourier transform of  $N$  data points, where  $N$  is a power of 2. The program is based on the binary recursive FFT algorithm described in [12].

#### 7.3.3. Cholesky Decomposition

The Cholesky Decomposition program is an algorithm for the factoring of a symmetric positive definite matrix  $A$  of order  $N$  into the product  $LL^T$  of a lower triangular matrix and its transpose. This algorithm is a transliteration of the dataflow-based algorithm presented in [13].

#### 7.3.4. Congruence Transformation

The Congruence Transformation program computes the congruence transformation of a matrix  $A$ . That is, given two  $N \times N$  matrices  $A$  and  $Q$ , it computes the matrix  $C=QAQ^T$ . This algorithm is based on an algorithm presented in [13].

Table I  
Benchmark program statistics

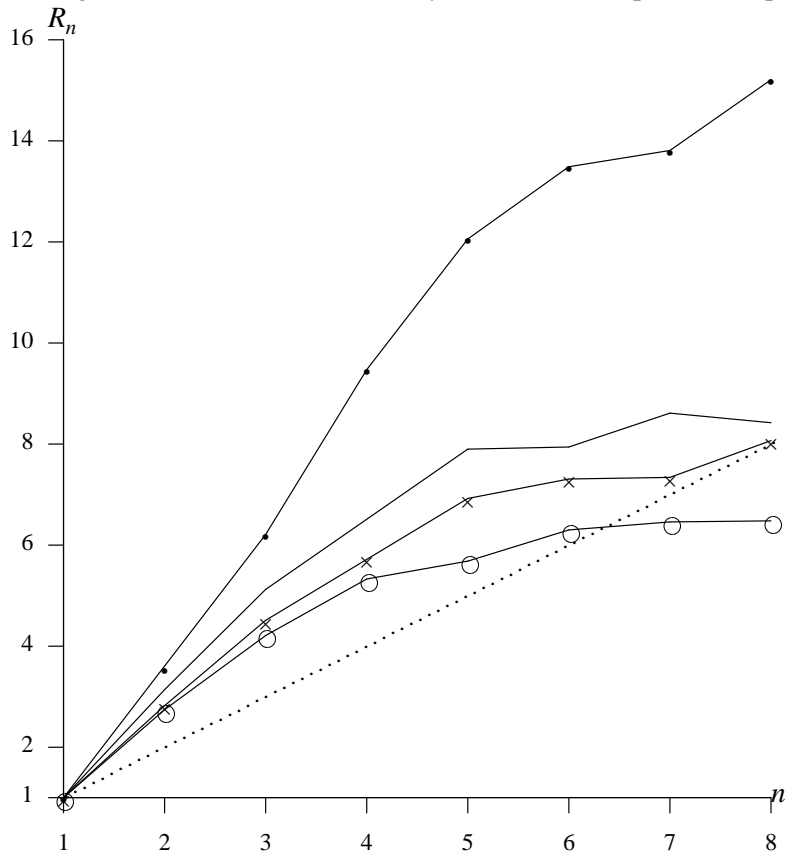
program	S	G	A
Matrix Multiplication	15	8	236
Fast Fourier Transform	63	17	587
Cholesky Decomposition	74	22	428
Congruence Transformation	66	18	434
S - number of Occam source lines G - number of dataflow graphs A - total number of actors			

### 7.4. Simulation Results

The principle performance metric used in our study is the system throughput ratio. The system throughput ratio,  $R_n$ , is defined as the ratio of the throughput on a system with  $n$  processors to the throughput on a system with 1 processor. Let  $X$  be the total work to be done in some particular workload class. The total workload consists of the sum of the user workload  $X^U$  and the kernel workload  $X^K$ . Consequently, the total execution time,  $T_n$ , is the sum of the user execution time,  $T_n^U$ , and the kernel execution time,  $T_n^K$ . The throughput of the system is the ratio of the user workload to the total execution time,  $X^U/T_n$ . (The contribution to the workload due to the kernel is overhead and is excluded from the

throughput calculation.) Thus,  $R_n = \frac{(X^U/T_n)}{(X^U/T_1)} = \frac{T_1}{T_n}$ .

The system throughput ratios obtained by simulation of the four benchmark programs described above are shown in Fig. 6. A number of Queue Machine Multiprocessor systems have been simulated having one to eight processing elements. Note that for small values of  $n$  (the number of processors), the system throughput ratio is greater than  $n$ . In effect, the system exhibits super-linear speedup.



Legend:

- × — × Matrix Multiplication
- — ○ Fast Fourier Transform
- Cholesky Decomposition
- — • Congruence Transformation

**Fig. 6. System throughput ratios for the benchmark programs vs. number of processors.**

These simulation results can be explained by using a modified derivation of Amdahl's law [14] that takes into consideration the effect of the kernel workload as well as the user workload. We assume that the user workload consists of two parts: (1) a sequential part that requires a fixed amount of execution time,  $S^U$ , and (2) a parallel part, the execution time of which is inversely proportional to the number of processors,  $P^U/n$ . Thus,  $T_n^U = \frac{P^U}{n} + S^U$ . The kernel workload consists of a large number of calls to kernel primitives. We assume that these calls can execute independently. Therefore,  $T_n^K = Y/n$ , where  $Y$  is proportional to the total work done by all the kernel calls. It is important to observe that some of the kernel workload is proportional to the number of processes running on a given processor. Thus, it is argued that  $Y = \frac{P^K}{n} + S^K$ . Hence,  $R_n = \frac{1}{1 + (1/n - 1)f + (1/n^2 - 1)g}$ , where  $f = \frac{S^K + P^U}{P^K + S^K + P^U + S^U}$ , and  $g = \frac{P^K}{P^K + S^K + P^U + S^U}$ .

Note that  $\lim_{n \rightarrow \infty} R_n = \frac{1}{1-(f+g)}$ . That is, the system throughput ratio has a finite upper bound. Also  $\left. \frac{dR_n}{dn} \right|_{n=1} = f+2g$ . That is the slope of the graph of  $R_n$  vs.  $n$  is  $f+2g$  at  $n=1$ . Note that when  $S^U < P^K$ ,  $f+2g > 1$ . That is, for small values of  $n$ , the slope of the graph of  $R_n$  vs.  $n$  can be greater than 1. This situation is called super-linear speedup.

Although it may seem unlikely, this kind of super-linearity is in fact a real phenomenon. Super-linear results have been reported in [15]. One way to view super-linearity is that is the manifestation of a kernel implementation that penalizes systems with a small number of processing elements and that becomes more efficient only when the number of processing elements is increased.

Note that if we neglect kernel overhead,  $f = \frac{P^U}{P^U + S^U}$  and  $g=0$ . In this case, the system throughput ratio can be simplified to  $R_n = \frac{1}{1+(1/n-1)f}$  (Amdahl's law)[14].

## 8. Conclusions

In this paper we have presented a new dataflow execution model called semi-static dataflow. Programs are partitioned into a collection of dataflow graphs. Each of these graphs is evaluated by a low level process (called a context) using the static dataflow execution model. Contexts are dynamically created during execution to implement iteration, conditional execution, and function invocation. This execution model provides the benefits of both static and dynamic dataflow architectures without the associated costs of code copying or tag manipulation.

We have also presented a number of program structure paradigms for function invocation, sequential iteration, and parallel iteration. These paradigms are used by a prototype compiler for the Occam programming language to generate semi-static dataflow program graphs. This compiler has been used to generate code for a semi-static dataflow multiprocessor. We have presented the simulation results for several benchmark programs. These results show that the semi-static execution paradigm is capable of automatically exploiting the increased parallelism available as the number of processors in the multiprocessor system is increased. We have used a modified derivation of Amdahl's law to justify the simulation results.

## 9. References

1. V. P. Srimi, "An Architectural Comparison of Dataflow Systems," *Computer*, Vol. 19, No. 3, Mar. 1986. pp. 68-88,
2. K. W. Todd, "Function Sharing in a Static Data Flow Machine," *Proc. 1982 Int. Conf. Parallel Processing*, IEEE, Aug. 1982. pp. 137-139,
3. J. B. Dennis, "Data Flow Supercomputers," *Computer*, Vol. 13, No. 11, Nov. 1980. pp. 48-56,
4. A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *Computer*, Vol. 15, No. 2, Feb. 1982. pp. 26-41,
5. G. S. Miranker, "Implementation of Procedures on a Class of Data Flow Processors," *Proc. 1977 Int. Conf. Parallel Processing*, IEEE, Aug. 1976. pp. 77-86,
6. Arvind and K. P. Gostelow, "The U-Interpreter," *Computer*, Vol. 15, No. 2, Feb. 1982. pp. 42-49,

7. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labelling," *1979 Nat. Comp. Conf.*, Vol. 48, June 1979. pp. 623-628,
8. INMOS Limited, *OCCAM Programming Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
9. B. R. Preiss, "Data Flow on a Queue Machine," Ph.D. Thesis, Univ. of Toronto, Dept. of Elec. Engin., Toronto, Ontario, Canada, 1984.
10. B. R. Preiss and V. C. Hamacher, "Data Flow on a Queue Machine," *Conf. Proc. 12<sup>th</sup> Ann. Symp. Comp. Arch.*, IEEE Computer Society Press, June 1985. pp. 342-351,
11. B. R. Preiss, "Design and Simulation of a Data-Flow Multiprocessor System," M.A.Sc. Thesis, Univ. of Toronto, Dept. of Elec. Engin., Toronto, Ontario, Canada, 1984.
12. J. D. Lipson, *Elements of Algebra and Algebraic Computing*, Reading, MA: Addison-Wesley, 1981.
13. D. P. O'Leary and G. W. Stewart, "Data-Flow Algorithms for Parallel Matrix Computations," *Communications of the ACM*, Vol. 28, No. 8, Aug. 1985. pp. 840-853,
14. J. P. Riganati and P. B. Schneck, "Supercomputing," *Computer*, Vol. 17, No. 10, Oct. 1984. pp. 97-113,
15. J. Sanguinetti, "Performance of a Message-Based Multiprocessor," *Computer*, Vol. 19, No. 9, Sept. 1986. pp. 47-55,