# EXPERIENCES IMPLEMENTING DATAFLOW ON A GENERAL-PURPOSE PARALLEL COMPUTER[1]

Ellen Spertus and William J. Dally
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract — The MIT J-Machine [3], a massively-parallel computer, is an experiment in providing general-purpose mechanisms for communication, synchronization, and naming that will support a wide variety of parallel models of computation. We have developed two experimental dataflow programming systems for the J-Machine. For the first system, we adapted Papadopoulos' explicit token store [10] to implement static and then dynamic dataflow. Our second system made use of Iannucci's hybrid execution model [8] to combine several dataflow graph nodes into a single sequence, decreasing scheduling overhead. The conclusion describes how the strengths of the two systems can be combined. We have demonstrated the feasibility of efficiently executing dataflow programs on a general-purpose parallel computer.

## Introduction

The dataflow programming model is attractive because it exposes parallelism in computer programs, a crucial step in compiling for multiprocessing computers. Once a program has been converted into a dataflow graph, only necessary constraints among the operations remain, and the program can, ideally, be spread across many processors. In the past, this has been done on special-purpose dataflow machines such as DDM1 [5] and the Manchester Dataflow Machine [6] that directly execute dataflow graphs. In addition to research in pure dataflow architectures, there is a growing interest in developing hybrid architectures , such as the EM-4 [12], that take advantage of the parallelism found by dataflow methods without sacrificing the straight-line efficiency of von Neumann machines. Most recently, people have begun developing compilation techniques for executing dataflow programs on general-purpose parallel machines [1]. Our work is on these techniques.

## The J-Machine

The J-Machine is a massively-parallel MIMD computer based on the Message-Driven Processor (MDP), a custom chip [3] [4]. For this research, we used a simulator of a 32-node J-Machine. Each processor has 260K (4K on chip) of 32-bit words augmented with 4-bit tags. Types specified by tags include booleans, integers, symbols, pointers, and cfutures. A cfuture, used for synchronization, represents

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and by a National Science Foundation Graduate Fellowship.

an empty location and typically is written into a memory location before the actual data value is ready. If an attempt is made to operate on the cfuture, a fault occurs.

The MDPs communicate with each other by sending messages through a low-latency network. When a message arrives at its destination, it is placed on the message queue, and a new task is created when the message reaches the head of the queue.

## Overview

In the next section, we describe a straightforward method of implementing the dataflow language Id [9], based on Papadopoulos' explicit token store (ETS) [10]. In the following section, we describe the system we built to simulate Iannucci's hybrid architecture [8] on the J-Machine. In the conclusion, we discuss the strengths and weaknesses of the two systems and describe our plans to combine them into an efficient implementation of Id.

# ETS

## Static Implementation

Our first experiments with dataflow on the J-Machine involved *static* dataflow, in which dataflow graphs must be acyclic and nonreentrant. A dataflow graph is a representation of a program where nodes represent operators and arcs represent dependencies. *Tokens* are the mechanism for carrying data values on these arcs. Because there is no way to distinguish different invocations of a procedure under static dataflow, only one instantiation of a dataflow graph can be active at a time.

For a J-Machine implementation of static dataflow, a token on its way to a commutative dyadic (two-input) dataflow node, such as *plus*, can be represented by two words:

1. A header with the address of the code that will operate on the token.

2. The data value.

When a token is sent to a dataflow node, this two-word message is sent to the processor on which it should run. The code for a *plus* node, written symbolically, is:

```
[Initialization code]
R0 <- CFUT:0
R1 <- MSG.VALUE + R0
[Code to send result to destination]
```

The first time this code fragment runs, the first line loads a cfuture into register *R0*, signifying that the needed data is not present. When the second line tries adding the cfuture in *R0* to the new argument, a *cfuture fault* occurs because no arithmetic operations can be performed on cfutures. The cfuture fault handler, which is not built into the hardware, encodes the arriving token's value into the location of the first instruction and then suspends.[2] After this happens, the code has been changed to:

```
[Initialization code]
R0 <- [value of first token]
R1 <- MSG.VALUE + R0
[Code to send result and clean up]
```

When this is executed, the value of the first argument to arrive will be loaded into *R0*, and the new argument will be added to it by the second instruction. The sum is then sent elsewhere to be processed by one or more other nodes.

While the above example only works for commutative operations, it is simple to extend the strategy to non-commutative operations, such as subtraction. This can be done implicitly by having different entry points for the left and right arguments, reflected in their destination addresses, or explicitly by having code at the beginning of each node to check which token has arrived and to branch accordingly. Dally [2] goes into more detail about implementing static dataflow on the J-Machine.

### Dynamic Implementation

Because static dataflow is too restrictive for most purposes, not allowing code reentrance, we extended the system to support dynamic dataflow, where multiple invocations of the code corresponding to a dataflow graph can be active at once. To implement dynamic dataflow, we added an additional word to each token to hold the *context* pointer. Every instantiation of a dataflow graph, corresponding to a procedure application, for example, has its own context. In order to allow the code corresponding to a node to be reentrant (by keeping tokens from different invocations separate), unmatched values are not stored within the code but in a separate area of memory where the locations are determined as a function not only of the destination address but also of the context. This way, two left tokens waiting for their partners at the same node will be stored in different locations. This format is modeled after Papadopoulos' explicit token store (ETS) [10].

Sample code for a *plus* node in the dynamic system is:

```
A1 <- MSG.CONTEXT
R1 <- MSG.VALUE
R1 <- [A1+3] + R1    ; This line may fault
[Code to send result and clean up]
```

[2] This approach would be less efficient if the encoding of the instruction to load a constant into *R0* on the J-Machine were not so simple. To load the constant $X$ into *R0*, one simply places $X$ in the instruction stream. When it is encountered, because it is not tagged as an instruction, the decoder knows to interpret it as a constant to load into *R0*. Thus, creating the instruction to load $X$ into *R0* is trivial.

First, the context pointer is loaded into address register *A1*. Next, the value of the just-arrived token is loaded into general-purpose register *R1*. The addition will cause a cfuture fault if the previous token has not been stored into the location pointed to by *A1* with a hard-coded constant offset. If the other token has already arrived, it can be found in this location, and the addition can take place. The purpose of the constant offset is to keep tokens from different nodes of a graph in different locations when their contexts are the same.

The cfuture fault handler is simply:

```
[A1] <- R1
suspend
```

This stores the value of the newly-arrived token into the location pointed to by the context pointer and then suspends. There is no unnecessary overhead in the code for the node or for the fault handler. Each only contains instructions to do essential loads, stores, and ALU operations.

Performance. A recursive factorial program takes 431 ticks to compute 4!, compared to the 315 ticks the same algorithm takes on the J-Machine under Concurrent Smalltalk [7, p. 110], the most optimized compiler for the J-Machine. A tick is the time unit used by the simulator: one tick equals one instruction, even though not all instructions on the J-Machine will take the same time. Spertus [13] describes more details of the dynamic ETS system on the J-Machine, such as the calling convention used and the mapping of instruction nodes to storage slots.

## A Hybrid Approach

An obvious shortcoming of the ETS systems is that every dataflow instruction is scheduled separately. After even the most trivial operation, such as a single addition, tokens must be built, sent, and matched. In contrast, on a hybrid system, the compiler groups *threads* of instructions into *scheduling quanta* (SQs), reducing overhead at the cost of reducing available parallelism. In practice, the reduction of parallelism is minimal, as factors such as network delay prevent procedure calls from being divided among processors on a dataflow machine. Instead, the parallelism comes from pipelining and from running iterations of a loop concurrently on separate processors [11]. A major benefit of a hybrid architecture over a pure von Neumann architecture is that the hybrid shares the latency toleration found in dataflow machines: When one branch of execution is waiting for a result from elsewhere, other instructions whose data dependencies are satisfied will be executed. A fuller justification of hybrid architectures can be found in [8].

We did not devise our own methods to partition instructions into threads but relied on Traub's methods [16]. We built a new back end for Iannucci's hybrid Id compiler and, in many cases, imitated mechanisms from his hybrid architecture [8].

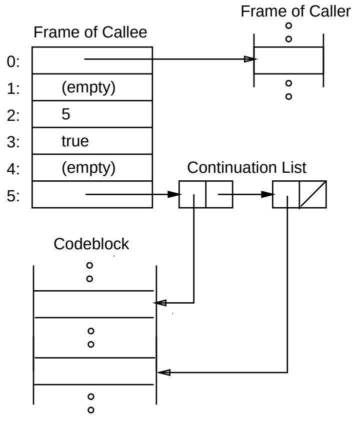


Figure 1: Run-Time Data Structures

Run-Time Structures

A procedure is divided into many SQs that together constitute a *codeblock*. When a codeblock is invoked, a contiguous region of memory called a *frame* is allocated for its arguments and scratch variables. The frame is given a unique global address, built by combining its processor number with its local address. Its first slot is initialized to point to the caller's frame. Because frames provide each invocation of a procedure with its own data area, the same procedure can be executed multiple times on one processor, with execution of the invocations interleaved. After a codeblock starts running, it will probably fault on a slot in its frame; that is, it will look for a value in a specific slot of the frame in which no data is present, and execution of the codeblock will thus be unable to continue. In this case, a *continuation* is created encoding the address where execution should restart when the data arrives, and this continuation is stored into the empty slot. When the data arrives, it will be written into the slot and the continuation will be re-enabled, i.e. placed in the message queue for subsequent execution. When all of the SQs in a codeblock have successfully completed and any return values have been sent to the caller, the frame can be freed. These structures are shown in Figure 1 and are described in greater detail in [14]. Our compiler took hybrid code produced by Iannucci's compiler [8] as input, outputting MDP code.

As encouraged by the semantics of Id, up to $K$ different iterations of a given loop can be active at a time, where $K$ is the dynamic loop-unfolding constant. Because iterations of an outer loop nominally run on the same processor, individual instructions are not executed concurrently. Instead, the parallelism comes from two sources: First, when a calculation within one iteration is waiting for data, such as the result of a procedure call to another processor, instructions from other active iterations may be executed, subject to data dependencies. Second, if there are subroutine calls or inner loops within the loop, these may be spawned onto other processors and run in parallel, as will be described in more detail below. Because up to $K$ iterations of a loop may be active at once, there must be $K$ iteration areas allocated to store the intermediate values. Thus, frames for codeblocks with loops are larger than frames for non-loop codeblocks. The complete loop parallelization strategy is described in [14].

Performance

Because we used a simulator of the J-Machine, we only were able to run small programs and to simulate a small number of processors (up to 32). Larger-scale experiments await the construction of the J-Machine. The two programs we analyzed in greatest depth were a recursive factorial program and a simple program that used the loop constructs.

Factorial. One test program was the same recursive factorial program used for the ETS system. In our preliminary system, no load-balancing is performed by the compiler. Instead, we modified the MDP code by hand to spawn the factorial call with argument $n$ to processor $n$ to distribute processor usage. The total code length is 180 MDP instructions, not counting code in library routines [14].

When 4! is computed on the MDP simulator, it takes 1263 ticks for the result to be written to the original calling frame. With four processors enabled, utilization is 37% — i.e., on average, a processor does useful work a little over a third of the time. Fault and library usage are shown in Table 1. The functions of these routines are:

- Lookup — Check if a frame slot holds a disabled continuation before writing to it. If a continuation is present, enable it in order to signify that the data has arrived.
- Cfut — Store the current continuation in the slot on which the cfuture fault occurred, then suspend.
- Move-Remote — Send a value into a frame slot of a parent or child procedure, usually on another processor.
- Continue-Test — Test whether a continuation will be able to run, before enabling it.

| *Handler Name* | *# Calls* | *Ticks/Call* | *Total Ticks* |
|---|---|---|---|
| Lookup | 25 | 5 or $6+6w$ | 212 |
| Cfut | 21 | 18 | 378 |
| Move-Remote | 16 | $13+7w$ | 264 |
| Continue-Test | 14 | 7 or 20 | 189 |
| Get-Context | 3 | 24 | 72 |
| Allocate | 3 | 12 | 36 |
| Total | 136 | n/a | 1061 |

Table 1: System Calls for `(fact 4)`: $w$ indicates the number of waiting continuations.

| $p$ | *Ticks/Call* | *Ticks/Skewed Calls* | | *Ticks/Nonskewed Calls* | |
|---|---|---|---|---|---|
| | | $1^{st}$ Result | $2^{nd}$ Result | $1^{st}$ Result | $2^{nd}$ Result |
| 4 | 1263 | 1864 | 2163 | 1992 | 2271 |
| 8 | 2691 | 4204 | 4590 | 4332 | 4611 |
| 12 | 4119 | 6544 | 6846 | 6672 | 6951 |

Table 2: Latencies for Factorial. This table compares the number of ticks required to compute one and two calls of factorial. For each case, the number of processors used is the same as the argument. The first data column shows how long it takes for one call executing alone to complete. The second set of columns shows how long it takes to complete two factorial calls initiated at the same time, skewed among the enabled processors. The last set of columns shows the completion times when the two calls are not skewed among the processors.

- Get-Context — Allocate and initialize a frame locally, send the address to the calling procedure, and enable the first SQ in the codeblock.

- Allocate — Allocate a block of memory to be used by get-context or cfut.

As the totals show, 56% of the time is spent in the operating system. Note that this value does not include any overhead operations, described above, that were inlined to the code for a node. The routine that consumed the most time is the cfuture fault handler. It is called 21 times, and each time takes 18 ticks. It takes so long because it must allocate space to store a continuation and fill in the necessary data.

As described above, the ETS implementation takes 431 ticks, compared to the hybrid system's 1263. The cost of the hybrid system's using library routines to simulate Iannucci's architecture was very high. A more efficient approach would involve less run-time interpretation.

A reason why computations take a relatively long time to complete on both systems is that at many design decisions, throughput was favored over latency, such as the decision to break apart any transaction of unbounded latency in the hybrid system, which increased the latency of tasks but improved throughput. Table 2 shows that computing two invocations of factorial concurrently on the J-Machine takes significantly less than twice as long as computing a single call. This is true for two reasons:

1. Each task suspends itself when it is waiting for a result from another processor, allowing that processor to be used by another task.

2. The factorial calls can be skewed among the processors, e.g., by mapping the calls to processors in such a way that interference between the two calls is minimized.

Table 2 isolates these factors by including results for both skewed and non-skewed procedure calls. Even when two calls are executed on the same processors, in the same order, throughput is increased over the single call case because subtasks of the second factorial call can run when no work can be done on a given processor for the first factorial call.

Loop Parallelization. For testing the loop parallelization strategy, we wrote a trivial program to sum the first $n$ integers. This micro-benchmark allowed us to determine the overhead to set up iteration areas and to launch iterations of a loop in parallel. The number of ticks, as a function of $K$ (the number of iterations to unroll) and $n$ (the argument) was $50 + 5 * K + 135 * n$. The three addends of the formula can be interpreted as follows: The constant term, 50, indicates that the additional cost for a procedure to use loop parallelization is low. The term $5 * K$ indicates that once the base cost for loop parallelization has been paid, it only costs 5 ticks to add and support each iteration area. This makes it reasonable to unroll many iterations of a loop. The term $135 * n$ shows the overhead involved in managing the flags and loop variables in each iteration of the loop. If each iteration of the loop spawns a long subroutine, the only additional code that will run on the home processor is that to spawn a procedure call. This means that each iteration of the loop will use fewer than 200 ticks on its home processor, regardless of how big a computation it performs.

## Conclusion

Our implementation of ETS and hybrid dataflow systems demonstrates dataflow programs can be executed in a reasonable manner on a general-purpose parallel computer. However, certain aspects of each system can be improved. The hybrid system did far too much run-time simulation of Iannucci's architecture, which was especially costly because his compiler was optimized for operations cheap on his architecture but expensive on ours. The ETS system gave better results, but a large percentage of its execution time went to overhead which could be avoided if nodes of the dataflow graphs were combined.

We now plan to build a system that retains the scheduling quanta and loop parallelization of the hybrid system and the low overhead of ETS. We believe this can be done by basing our new system directly on the Id compiler used by the MIT Computation Structures Group instead of using Iannucci's machine language as an intermediate step, only making use of his routines to parallelize loops and Traub's routines to partition a dataflow graph into scheduling quanta [16]. With Traub's SQs, we expect the new system to be several times faster than our ETS system.

The system we plan to build is based on the observation that each SQ has a fixed number of tokens used as inputs, where the number of input tokens is known at compile-time. Instead of performing a single operation, as the ETS system does, when the argument tokens arrive, this system would execute an entire SQ after all the input tokens arrive.

The first slot of the new frame structure would hold a count of the number of arguments to the SQ that have arrived, and subsequent slots would hold these arguments. When an argument arrives, it is stored in the appropriate slot, and the count is incremented. If the count indicates that all of the arguments have arrived, the SQ will be executed. This system would be more efficient than the ETS system because of the less fine-grained scheduling: Instead of scheduling one machine instruction at a time, it schedules several. Its primary advantage over the hybrid system is that the expensive lookup system call and cfuture fault handler used to imitate Iannucci's architecture would be entirely unnecessary.

## Acknowledgments

We would like to acknowledge the members of the MIT Concurrent VLSI Architecture and Computer Architecture groups for their help throughout the project, including Beng-Hong Lim and Lisa Sardegna, who helped us prepare this paper. Additionally, Arvind, Jamey Hicks, Robert Iannucci, and Jonathan Young of the MIT Computation Structures Group were very generous with their expertise and equipment. We are grateful to Cecelia Buchanan and John Zahorjan of the University of Washington for providing one of the authors with a computer account to work on this paper when away from MIT. We thank André Dehon, Kathy Knobe, and Nate Osgood of MIT, Jeri Steele of Texas Instruments, and Helen Takacs of Mississippi State University for their many helpful comments on earlier versions of this paper.

## References

[1] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek, "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pages 164 – 175.

[2] William J. Dally, *Dataflow on the J-Machine*, an unnummbered MIT Concurrent VLSI Architecture memo, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[3] William J. Dally, et al., "The J-Machine: A Fine-Grain Concurrent Computer," *Information Processing 89, Proceedings of the IFIP Congress*, 1989.

[4] William J. Dally, et al., *Message-Driven Processor Architecture*, MIT Artificial Intelligence Laboratory Technical Memo 1069, Cambridge, MA, 1988.

[5] A. L. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," *Proceedings of the National Computer Conference*, 1979, pages 1079–1086.

[6] John Gurd and Ian Watson, "Data Driven System for High Speed Parallel Computing — Part 2: Hardware Design," *Computer Design*, July, 1980.

[7] Waldemar Horwat, *Concurrent Smalltalk on the Message-Driven Processor*, MIT Artificial Intelligence Laboratory Technical Report 1080, Cambridge, MA, 1990. (Master's Thesis, Department of EECS, MIT.)

[8] Robert Alan Iannucci, *A Dataflow / von Neumann Hybrid Architecture* MIT Laboratory for Computer Science Technical Report 418, Cambridge, MA, 1988. (PhD Thesis, Department of EECS, MIT.)

[9] Rishiyur S. Nikhil, *ID Version 88.1 Reference Manual*, Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[10] Gregory Michael Papadopoulos, *Implementation of a General Purpose Dataflow Multiprocessor*, MIT Laboratory for Computer Science Technical Report 432, Cambridge, MA, 1989. (PhD Thesis, Department of EECS, MIT.)

[11] Gregory M. Papadopoulos and David E. Culler, "Monsoon: An Explicit Token Store Architecture", *Proceedings of the* $17^{th}$ *International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[12] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba, "An Architecture of a Dataflow Single Chip Processor," *Proceedings of the* $16^{th}$ *Annual International Symposium on Computer Architecture*, Jerusalem, Israel, 1989, pages 46–53.

[13] Ellen Spertus, *Preliminary Dataflow on the MDP*, MIT Concurrent VLSI Architecture Memo 21, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1989.

[14] Ellen Spertus, *Dataflow Computation for the J-Machine*, MIT Artificial Intelligence Laboratory Technical Report 1233, Cambridge, MA, 1990. (Bachelor's Thesis, Department of EECS, MIT.)

[15] Ellen Spertus and William J. Dally, *Experiments with Dataflow on a General-Purpose Parallel Computer*, MIT Artificial Intelligence Laboratory Technical Memo 1272, Cambridge, MA, 1991.

[16] Kenneth R. Traub, *Sequential Implementation of Lenient Programming Languages*, MIT Laboratory for Computer Science Technical Report 417, Cambridge, MA, 1988. (PhD Thesis, Department of EECS, MIT.)