Multithreading: A Revisionist View of Dataflow Architectures

G.M. Papadopoulos, K.R. Traub

In Proceedings of The 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991. Also published as Motorola Technical Report MCRC-TR 10

1991, May

Computation Structures Group
Memo 330

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Multithreading: A Revisionist View of Dataflow Architectures

Computation Structures Group Memo 330
March, 1991

Gregory M. Papadopoulos
Kenneth R. Traub

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Multithreading: A Revisionist View of Dataflow Architectures

Gregory M. Papadopoulos
Massachusetts Institute of Technology

Kenneth R. Traub
Motorola Cambridge Research Center

Abstract

Although they are powerful intermediate representations for compilers, pure dataflow graphs are incomplete, and perhaps even undesirable, machine languages. They are incomplete because it is hard to encode critical sections and imperative operations which are essential for the efficient execution of operating system functions, such as resource management. They may be undesirable because they imply a uniform dynamic scheduling policy for all instructions, preventing a compiler from expressing a static schedule which could result in greater run time efficiency, both by reducing redundant operand synchronization, and by using high speed registers to communicate state between instructions.

In this paper, we develop a new machine-level programming model which builds upon two previous improvements to the dataflow execution model: sequential scheduling of instructions, and multiported registers for expression temporaries. Surprisingly, these improvements have required almost no architectural changes to explicit token store (ETS) dataflow hardware, only a shift in mindset when reasoning about how that hardware works. Rather than viewing computational progress as the consumption of tokens and the firing of enabled instructions, we instead reason about the evolution of multiple, interacting sequential threads, where forking and joining are extremely efficient. Because this new paradigm has proven so valuable in coding resource management operations and in improving code efficiency, it is now the cornerstone of the Monsoon instruction set architecture and macro assembly language. In retrospect, this suggests that there is a continuum of multithreaded architectures, with pure ETS dataflow and single threaded von Neumann at the extrema. We use this new perspective to better understand the relative strengths and weaknesses of the Monsoon implementation.

1 Introduction

The ability of dataflow machines to expose ample amounts of all sorts of parallelism—instruction, loop, procedure, producer/consumer, unstructured—is well documented [3], and continues to be an attractive feature of the approach. This ability derives from fundamental properties of dynamic dataflow graphs, which are directly executed as a dataflow processor's machine language. Dataflow machine graphs represent the program as a partial order of essential dependences, and instructions are dynamically scheduled based on the availability of data. That is, a dataflow machine program does not *overspecify* the actual order of instruction execution, and instead relies on low-level runtime mechanisms, *e.g.* operand matching, to dynamically pick a particular execution order, including ones where many instructions are executed simultaneously within and across processors. Moreover, given sufficient parallelism, dynamic instruction scheduling has the added pragmatic benefit of being resilient to long and unpredictable communication latency.

The most recent generation of dataflow machines (*e.g.*, MIT's Monsoon [11, 12], ETL's EM-4 [13], and Sandia's Epsilon-2 [6]) have shown how operand matching can be accomplished with simple hardware structures in two machine cycles. There does seem to be an unavoidable price of purely dynamic instruction scheduling, however. Each dyadic (two-input) instruction requires the dynamic matching of its operands and must amortize the associated expense of making copies of data when a particular value is required by several instructions. In careful comparative studies of scientific codes, we have found that, for a given algorithm, a dataflow machine tends to execute two or three times as many instructions as a von Neumann uniprocessor [2]. Unfortunately, the overhead is incurred even when a good static schedule is known at compile-time. Given that operands for an *expression* are available, we would want a compiler to be able to specify a particular execution order for instructions within that expression in order to eliminate any intra-expression synchronization and associated data copying. In this case, intermediate values could be communicated among instructions in the sequence via temporary registers, rather than asynchronously propagating values on tokens [9, 13].

Another objection to pure dataflow graphs concerns the coding of low-level operating system and resource management functions, such as trap handlers and storage allocation routines. These operations require frequent, guaranteed exclusive access to processor state and need critical sections around the updating of shared data structures. When confined to pure dataflow graphs, these sections are extremely hard to code and reason about.

It appears that both the dynamic instruction inefficiency and critical section problems share a common root cause—the inability of the compiler to directly control when a *successor* instruction will execute. That is, we would like to be able to create a sequence of instructions which get executed, in order, once the first instruction in the sequence has been dynamically scheduled. This value of this ability has been recognized by several researchers [9, 13, 6].

In the case of synchronous circular pipelines, such as Monsoon, this capability is surprisingly simple to implement: an instruction merely demands that one of its successor tokens re-enter the pipeline immediately following its execution. The instruction may then communicate with its successor through temporary registers, as well as the value carried on the token. The relationship between successors resembles not so much a chain of dependents in a dataflow graph as it does a von Neumann sequential thread.

This leads to an entirely different view of computation in the dataflow machine. Rather than viewing computational progress as the consumption of tokens and the firing of enabled instructions, we instead reason about the evolution of multiple, interacting sequential threads, where forking and joining are extremely efficient. Because this new paradigm has proven so valuable in coding resource management operations and in improving code efficiency, it is now the cornerstone of the Monsoon instruction set architecture and macro assembly language. In the following sections, we present the Monsoon macroarchitecture as a multithreaded multiprocessor, giving coding examples and experimental performance results of the approach. We also show that, without loss of generality, this new macroarchitectural model captures traditional dataflow execution, as well. We emphasize, presented here is a new way to *denote* multi-threading that harmonizes the apparently disparate dataflow and sequential execution styles.

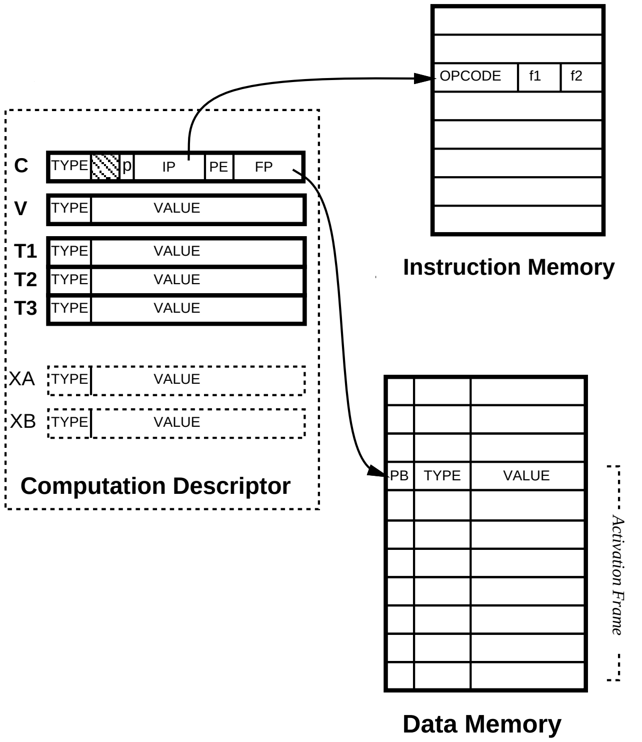


Figure 1: A Computation Descriptor

2 Computation Model

Under this new model we view Monsoon as a multi-processor, multi-threaded computer: within each of many processing elements (PE's) are many sequential threads of control. Each thread can be thought of as an independent instruction stream, and each thread has an independent set of registers (albeit a very small set). Naturally, each processor has a fixed limit as to the number of threads it can actually process simultaneously, but the limit on the total number of active threads within a processor is much, much larger. In the current version of Monsoon, for example, up to eight threads are processed simultaneously in the pipeline of one PE, but up to 32 thousand threads may be active in that PE, awaiting execution. Both of these numbers scale up as more processors are added.

The state of a thread on Monsoon is contained in a *computation descriptor*, or CD. As shown in Figure 1, the CD has five registers, the continuation register C, the value register V, and three temporaries *T1*, *T2*, and *T3*. There are also two other registers XA and XB, but as they only play a role during exceptions, the programmer usually is not concerned with them (exceptions are not discussed further in this paper).

The continuation register C of a CD defines the context in which the CD's thread executes. It contains a pair of pointers: the *instruction pointer* that indicates the next instruction to be executed, and a pointer to data memory called the *frame pointer*. The instruction set is designed to support a programming paradigm wherein the frame pointer is the base address of an activation frame for a procedure invocation; by using frame-relative addressing, the same code block can have multiple active invocations. The instruction pointer and frame pointer are constrained to have the same node number, and furthermore that node is the PE that will actually execute the thread. The C register also has a field called PORT that is used only during the Join operation (Section 3.2), and in that circumstance is akin to an extension of the instruction pointer.

The remaining registers in a CD, V, *T1*, *T2*, and *T3*, simply hold user data associated with a thread. Only V is used in "pure dataflow" instructions. The use of T-registers is discussed in Section 4.

As previously noted, on a given processor only eight CD's will have actively executing threads, whereas each processor will maintain a hardware-managed queue of up to 32 thousand CD's corresponding to threads that are ready, but not actively executing. A CD is automatically popped from the ready queue, with zero overhead, whenever a processor requires work. Note that the ready queue only maintains the C and V registers, so the temporary registers are defined only for actively executing threads. Our notation provides ways to guarantee that an actively executing thread will continue execution, and thus preserve its temporary registers.

In addition to the CD's of its resident threads, each processing unit of a Monsoon multiprocessor has under its control a portion of the global memory address space. There are actually two address spaces: the *instruction memory* address space and the *data memory* address space.[1] Both instruction memory addresses and data memory addresses have two parts: a node number and an offset into the portion of the address space under that node's control. For many

[1] Having separate instruction memory and data memory address spaces is not an essential feature of this type of architecture, but was done to simplify the design of the current prototype.

types of operations, the division of an address into node number and offset can be ignored, with the node number viewed simply as the high-order bits of a global address. On the other hand, many instructions have addressing modes that access data memory, but with the constraint that the data memory address have the same node number as that of the instruction being executed; an address fulfilling this constraint will be called a *local address*. As noted above, the PE field of a thread's C register indicates where it executes, and so references to the activation frame as well as instruction fetch are all local references.

Instruction memory words are 32 bits, and words of data memory are 72 bits (64 data bits plus eight type bits available for implementing tagged data). Registers that hold data are also 64 bits wide with eight type bits. In addition to data and type bits, each word of data memory has three *presence bits*. Presence bits are used to implement a variety of synchronization protocols, discussed at length in the sequel. Though associated with each location, presence bits are not considered part of the contents of a data memory word, and are not found in the registers of a CD.

3 Basic Instruction Set

In this section we describe the basic instructions for manipulating the C and V registers, for creating and synchronizing threads, and for making global memory references. We present these instructions from the multi-thread perspective, but at the end of this section we show that they are indeed familiar dataflow instructions. When seen from the multi-thread point of view, however, the natural style of coding is quite different from that derived from dataflow graphs.

3.1 Arithmetic and Local Data

The instruction set is similar to that of an accumulator (one-address) machine, with V acting as the accumulator. For example, a program fragment

```
e := a * b - 6.0 * c;
```

might be compiled into the following sequential thread, where each instruction is assigned a consecutive instruction memory address:

```
mov     [fp+A], v       ; v = A
fmul    v, [fp+B]       ; v = A * B
mov     v, [fp+TEMP]    ; TEMP = A * B
mov     [LIT_SIX], v    ; v = 6.0
fmul    v, [fp+C]       ; v = 6.0 * C
fsub    [fp+TEMP], v    ; v = A*B - 6.0*C
mov     v, [fp+E]       ; E = A*B - 6.0*C
```

In this example, it is assumed that a, b, c, e, and the temporary temp are local variables held in the activation frame. The example shows both the *FP-relative* addressing mode for accessing these variables, and the *absolute* mode for accessing the constant six. (In the example, the offsets from the beginning of the frame for a, b, *etc.*, as well as the address of the constant six, are shown with assembly-time symbols rather than with numbers.) Both addressing modes can only make *local* references, *i.e.*, references to locations on the same node as is executing the instruction. All of these instructions increment the IP field of C to go to the next instruction.

3.2 Multiple Threads: Fork and Join

The simplest primitive for introducing a new thread into the machine is fork. The fork instruction is like a combination of jump and falling through to the next instruction. For example, executing this instruction

```
fork    lab1
```

has two effects. First, the current thread continues to the next instruction; that is, the IP field of the C register is incremented. Second, a new thread is introduced into the system, whose C and V registers are the same as those of the current CD, except that the IP field of C is lab1.

Two threads executing on the same PE may synchronize with each other using the Join mechanism. Join is not an instruction in itself; rather, it is a modifier that may be applied to other instructions. For example, a join-subtract would be written like this:

```
label2 [fp+Q]:  fsub    vl, vr
```

The use of Join is indicated by the appearance of a memory operand before the colon that delimits a label. The idea behind a Join is that two threads will fetch a Join-modified instruction at different times, but only the second one (in time) actually performs the indicated operation and continues. The first thread to execute merely saves the contents of its V register, and then dies without finishing the join-modified instruction or proceeding to the next. When the second thread executes the join-modified instruction, it retrieves the saved value, so that both V's are available as operands.

Threads participating in a Join must have different PORT bits in their C registers, so that the left and right operands can be distinguished regardless of the order in which the two threads happen to execute. (Remember that the PE is free to choose any active thread for execution on each cycle.) Jump and fork instructions may arbitrarily set the port in addition to computing a new IP. By convention, when a thread executes a non-joining instruction, its C register specifies the left port.

Every Join operation involves a location in local data memory called a *rendezvous point*; in the earlier example, the rendezvous point was the location named by [fp+Q]. The rendezvous point plays two roles. One, it serves as temporary storage for the V register of the first thread to execute. Second, its presence bits record whether any thread has yet executed the join, and what its port was. Initially, the rendezvous point for a Join must have presence bits set to empty. When the first thread executes the join, the contents of its V register are stored in the data and type fields of the rendezvous point, and the presence bits are set to left-present or right-present, according to the PORT field in that thread's C register. When the second thread executes the join, the contents of the rendezvous point are retrieved, the presence bits are set back to empty, and the instruction continues processing with the retrived value and the second thread's V register as operands. The presence bits are set back to empty so that the rendezvous point may be reused without reinitialization.

Having two states left-present and right-present is not strictly necessary, because the PORT of the second thread to participate in a Join serves to indicate which operand is which. Two states are used so that an exception may be raised if two left or two right threads try to use the same rendezvous point.

Here is the simple arithmetic example from the previous section, using `fork` and Join to cause the multiplications to execute in separate threads.

```
        fork     lab1
        mov      [fp+A], v
        fmul     v, [fp+B]

subit [fp+TEMP]:
        fsub     vl, vr
        mov      v, [fp+E]
        jump     done

lab1:   mov      [LIT_SIX], v
        fmul     v, [fp+C]
        jump     subit.r
```

The "`.r`" suffix in the final `jump` statement says that the PORT field in C should be set to right (in addition to the IP field being changed to `subit`). On the other hand, the default fall-through from the first `fmul` results in PORT being left.

Note that this very simple example does *not* make a compelling case for multi-threading, but was chosen merely to illustrate the fork and join mechanisms.

3.3 Split-Phase Transactions

The addressing modes described earlier can only access local memory, and only when the address (or offset from the current frame pointer) are compile-time constants. Computed, global memory references are performed with *split-phase transactions* [4], in which a thread issues a memory request and continues while the request is processed concurrently. This is the means by which the Monsoon architecture tolerates long memory latency.

To illustrate, here is a statement from the inner loop of DAXPY:

```
y[i] = a * x[i] + y[i]
```

Assuming that pointers to `x[i]` and `y[i]` are in frame locations `XI` and `YI`, respectively, the code for the inner loop is as follows.

```
         fch     [fp+XI], mulit
         fch     [fp+YI], addit.R
         stop

mulit:   fmul    [fp+A], v

addit [fp+TEMP]:
         fadd    vl, vr
         str     [fp+YI], v
         jump    done
```

From the programmer's point of view, the Fetch instruction (`fch`) is like `fork`, except that the new thread will have the fetched value in its V register. This new thread may take a comparatively long time to enter the pool of active threads, depending on the latency of the fetch. After executing a `fch` instruction, the current thread continues execution; in the example, after initiating the fetch to `x[i]` the thread goes on to issue the fetch to `y[i]`. The example also illustrates the `stop` instruction, which simply removes the current thread from the pool of active threads. The Store instruction (`str`) has a more conventional appearance, as no result need be returned from the remote memory.

Both `fch` and `str` are implemented as split-phase transactions. This means they execute by issuing requests and continuing immediately to the next instruction in the instruction stream. While the issuing thread continues in this manner, the request travels through the communication network to the node that holds the global location being operated upon. A Fetch request has the following format:

$$\langle \mathit{reqop} = \texttt{fch}, n, o \rangle, \mathit{rc}$$

where n and o are the node and offset of the location to be fetched, and rc is the *return continuation.* When the Fetch request is processed by node n, it initiates in the requesting node a new thread whose V register contains the value fetched and whose C register contains the return continuation rc. A Store request has the following format:

$$\langle \mathit{reqop} = \texttt{str}, n, o \rangle, v$$

When the Store request is processed by node n, it simply stores v in location $\langle n, o \rangle$. In both cases, the node processing the request only makes access to *local* data memory.

The result of the fetch request was returned in the V register of a new thread. But since the store request has no return value, the thread issuing the request simply continues. With no store acknowledgment, there is some question of ordering: will fetches issued after the store instruction receive the new value, or the old value? A special hardware mechanism insures the following invariant: all requests to the same location issued subsequent to a store by a particular thread, as well as by threads subsequently forked by that thread, will be processed following the store request. In the example above, the code at the label `done` can count on the store to `y[i]` having taken place, and in particular any fetches to `y[i]` issued by that code will receive the new value (assuming some other thread does not store to the same location). More simply, the hardware supports strong sequential consistency.

Monsoon supports a number of synchronizing memory transactions in addition to the imperative Fetch and Store operations already discussed. The most common of these are the *I-Structure* operations I-Fetch and I-Store [5]. These use presence bits to implement a synchronizing write-once protocol: an I-Fetch to an empty location is deferred by saving the return continuation in the location, and a subsequent I-Store causes the fetch to be satisfied. A more detailed explanation, including what happens when more than one I-Fetch request arrives before the corresponding I-Store, may be found in [11]. Monsoon also supports a pair of mutual exclusion operations called Take and Put. The format of requests for I-Fetch/I-Store and for Take/Put is exactly like that described for Fetch/Store; only the processing of these requests at the receiving node differs.

3.4 Continuation Manipulation

Instructions that manipulate continuations are normally used for procedure linkage, as they create threads with different FP's. The `start` instruction is similar to `fork` in that it introduces a new thread into the system, but in the case of `start` both the C and V for the new thread are given as operands. Thus, `start` can introduce any arbitrary CD into the system. The Add Immediate to Continuation and Start (aics) instruction is similar, but has a small immediate operand that is added to the IP of the new continuation.

Both start and aics expect a continuation as an operand. Two instructions are provided for constructing continuations: Make Continuation (mc), and Make Continuation for Destination (mcd). The mc instruction composes a continuation from a pointer to a frame (a global data memory address) and a pointer to an instruction (a global instruction memory address, constrained to have the same PE as the frame pointer operand). Typically this instruction is used to create a continuation for calling a procedure. The mcd instruction has a label operand, and creates a continuation with the same FP as the current continuation, but with an IP pointing to the instruction named by the label. The mcd instruction is most often used to form a return continuation.

A typical calling convention is illustrated below, for the following function.

```
def f(a,b,c,d) = (a * b) + (c * d)
```

In the calling convention illustrated here, the return continuation and each argument is sent from caller to callee by starting a new thread with the argument or return continuation in V. Each thread is started with a different IP: the return continuation specifies the first instruction of the code for f, the a argument the second instruction, and so forth. Since these five threads will share the same activation frame, created by the caller, they may rendezvous with each other by joining in frame locations.

This is the code for f (the callee):

```
entry:  jump    returnit.R
a_arg:  jump    mul_ab.L
b_arg:  jump    mul_ab.R
c_arg:  jump    mul_cd.L
d_arg:  jump    mul_cd.R
mul_cd [fp+0]:
        mul     vl, vr
        jump    addit.R
mul_ab [fp+1]:
        mul     vl, vr
addit [fp+2]:
        add     vl, vr
returnit [fp+3]:
        start   vr, vl
        stop
```

As compiled, the multiplication of a and b will proceed even if the arguments c and d have not arrived, and vice versa. Alternatively, the code could have joined on all operands before doing any arithmetic (although there would be no advantage to doing so, in this example).

The caller for the function above would look like this, assuming frame location FADDR holds a pointer to the code for f, and assuming the actual parameters are in frame locations AA, AB, AC, and AD. This is the code for a caller of f:

```
[code to obtain frame]
mc      v, [fp+FADDR]
;; V is continuation with
;; IP = first instr of f

;; Send args: start a thread for each arg.
aics    v, [fp+AA], 1
aics    v, [fp+AB], 2
aics    v, [fp+AC], 3
aics    v, [fp+AD], 4

;; Compute and send
;; return address
mov     v, [fp+TEMP]
mcd     done
start   [fp+TEMP], v
stop

;; Result arrives here
done:   ...
```

The PE field of the new continuation for the call will determine where the called procedure actually executes. Thus, the "code to obtain frame" in the example is responsible for making a load distribution decision.

The calling convention illustrated starts a new thread in the callee for each argument, plus one with the return address. It is also possible to devise a convention that starts only one thread in the callee, by storing the arguments directly into the callee's frame using str instructions.

3.5 Compounds and "Traditional" Dataflow

In the preceding sections, some simple instructions were presented from the multi-thread point of view, but most of them can be recognized as pure ETS dataflow instructions. The correspondence is established by considering the CD (or CD's, in the case of Join) of the threads executing the instruction to be incoming tokens, and the CD (or CD's) of the threads that result to be outgoing tokens. The C register is easily seen to be the tag of the token, and V the value. Figure 2 shows a number of instructions in both multi-thread and dataflow graph notation.

A few instructions discussed earlier are not normally found in traditional dataflow. For example, the instruction

```
fch   v, label3
```

is a fetch generating two tokens: the token receiving the result of the fetch (with IP of label3) and an immediately generated acknowledgment (at the next consecutive IP). While this form of fch proved useful in Section 3.3 for launching several fetch requests into the memory pipeline, it rarely appears in code compiled for pure dataflow machines. One suspects that this is because there is no adequate way to notate this form of fch in a dataflow graph.

There are other dataflow instructions which require a more elaborate multi-thread notation. A two-input, two-output, floating point add, for example, is notated as

```
[fp+5]: fadd    vl, vr   || fork label3
```

i.e., as the *combination* of an fadd and a fork. In the Monsoon architecture, this is really a single instruction, because presence bit manipulation, operand fetch, arithmetic, and token forming are handled in separate stages of the pipeline. Such compounds are distinct opcodes when assembled; the instruction format is not VLIW. Instructions may not be combined arbitrarily, but must fit within the constraints of the pipeline: an optional join, up to one frame operand, one ALU operation, and one token-forming operation. Other examples of compound instructions may be seen in Figure 2.

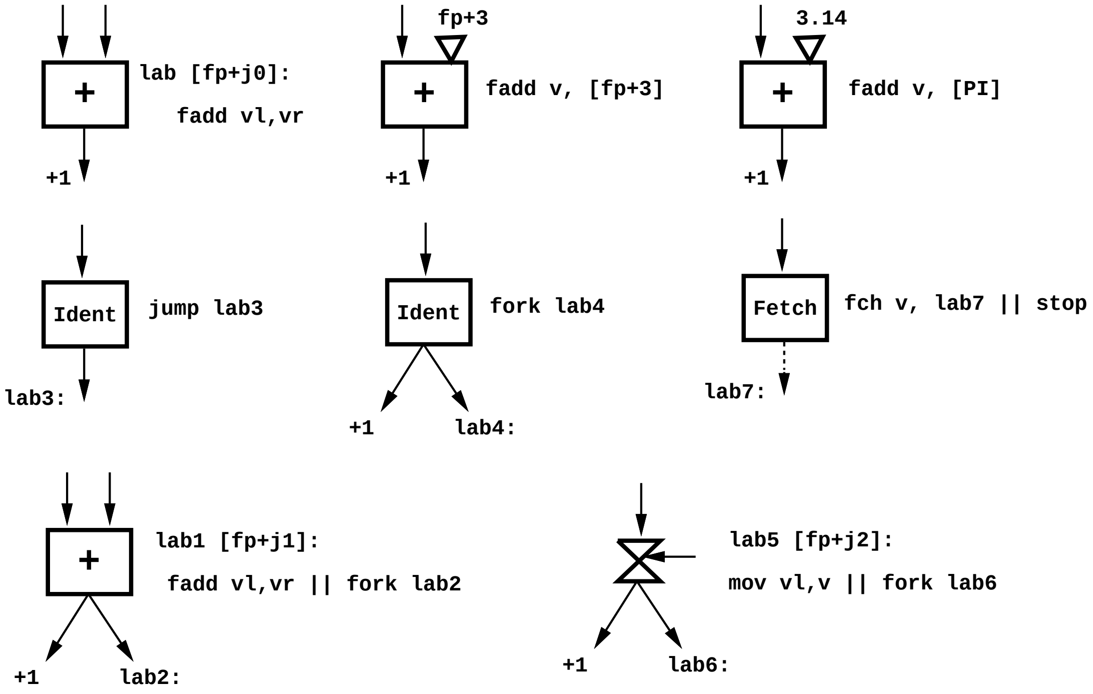


Figure 2: Sample Dataflow Operators and Corresponding Multithreaded Instruction

One noteworthy case is a `fch` that does *not* produce an acknowledgment:

```
fch v, label1 || stop
```

This can be read as "fork a new thread to `label1` containing the value at the location named by V, and stop the current thread." The distinctive notation for `fch` emphasizes its split-phase nature, whereas in dataflow graph notation `fch` would be notated exactly as a unary arithmetic operator.

4 Registers and Sequential Scheduling

There is a fundamental asymmetry in the multi-thread view of dataflow that is lacking in dataflow graph notation. Specifically, a distinction is drawn between tokens produced by *modifying the current C register* (falling through and jumping) and by *creating new threads* (the explicit destinations of `fork` and `fch`, and the computed destinations of `start` and `aics`). This asymmetry may be exploited by using it to control scheduling: if an instruction modifies the current C (*i.e.*, does not `stop`), then that thread will immediately re-enter the pipeline. All other threads are subject to a hardware queueing policy, as in conventional dataflow. Instructions that depend on the scheduling policy are annotated with a '>' mark.

This elementary "recirculate" scheduling directive opens up two new programming paradigms. One is the ability to write *critical sections*, exploiting both quick re-entry into the pipeline and that other threads are shut out of the pipeline as long as the critical section continues. This is essential to writing resource managers such as memory allocators for frame storage and heap storage—a topic that has received fuzzy treatment in the dataflow literature. We will not discuss critical sections further in this paper. One interesting issue in the Monsoon setting is that there may actually be eight simultaneous critical sections occupying different positions in the pipeline.

The other programming paradigm is the use of registers to improve the efficiency of code. It is easy to imagine that a CD has additional data registers besides V. But it would be impractical to provide these registers for all the active CD's in a processor, since they potentially number in the tens of thousands. Instead, extra registers are only provided in the CD's of the eight threads occupying the pipeline. As long as a thread recirculates in the pipeline it may freely use its registers, but once it leaves the pipeline it effectively loses the contents of all but C and V. Thus, the recirculate directive may also be viewed as a "preserve registers" directive.

The CD of each thread occupying the pipeline has three extra registers, *T1*, *T2*, and *T3*. Instructions may specify these as operands, in addition to V and the data memory operand. The result of an ALU operation, however, is always stored into V. Also, an instruction may separately write the old contents of V and/or the memory operand into T-registers; these writes are notated using compound notation (Section 3.5). For example:

```
fadd t1, t2 || mov v, t3 || mov [fp+5], t2 >
```

is a single instruction that stores V into *T3*, stores the contents of frame location five into *T2*, and adds *T1* and *T2*. When a register is both read and written, as *T2* is in the example, the read precedes the write. Thus, the operand to the addition is the *old* value of *T2*. While this may seem to be an exceedingly complex instruction, it really is just taking full advantage of the pipeline. Registers are read and written in a stage between the data memory stage and the ALU stage.

4.1 Example of Register Code

We now illustrate how scheduling and registers can improve compiled code efficiency with a very simple program that sums, in double-precision floating point, the integers from one to n. We compare the inner loops of four different formulations on Monsoon: pure dataflow, sequential without registers, sequential with registers, and parallelized sequential with registers. The model for the pure dataflow version is the following Id loop:

```
{ while (i <= n) do
    next sum = sum + i;
    next i   = i + 1;
 finally sum }
```

The model for the other three versions is this loop in C:

```
do {
  sum := sum + i;
  i   := i + 1;
} while (i <= n);
```

The pure dataflow code is shown in Figure 3. This code is similar to what would be produced by the Id Compiler[2], using a one-bounded loop schema. The instruction counts would be the same or slightly greater if a more elaborate k-bounded loop schema were used instead.

The first sequential version does not use T-registers, and so is "accumulator" style:

```
loop:
  mov   [fp+SUM], v
  fadd  v, [fp+I]
  mov   v, [fp+SUM]       ; sum <- sum + i
  mov   [fp+I], v
  fadd  v, [ONE]
  mov   v, [fp+I]         ; i <- i + 1
  fjle  v, [fp+N], loop   ; i <= n ?
```

The second sequential version uses T-registers, more than halving the length of the loop. The register allocation is $T1 =$ sum, $T2 =$ i, and $T3 =$ n.

```
loop:   fadd    t1, t2                          >
        mov     v, t1   || fadd t2, [ONE]       >
        mov     v, t2   || fjle v, t3, loop     >
```

Sequential code such as this will only occupy one of eight interleaves of the pipeline. To fill the pipeline, eight sequential loops can be initiated, each summing every eighth integer, with staggered starting points. The general scheme for this program is shown in Figure 4. This illustrates that the dataflow and multi-thread styles of reasoning about code may be freely mixed. The code for each inner loop is the same as above, except that eight is added to `i` instead of one.

The performance of these four loops is summarized in the table below. The figure under "instructions per iteration" is the number of instructions completed. The difference between this and "tokens per iteration" is always the number of joins per iteration. "Cycles per iteration" assumes that there is no other parallel activity in the machine, and a pipeline depth of eight. Threading without registers reduces the number of instructions per iteration from 14 to 7, but only reduces the number of cycles from 71 to 56. In Monsoon, eight threads are interleaved in the pipeline, so when only a single thread is executing the total number of cycles required will be eight times the number of instructions in the thread. The use of registers further reduces this number from 7 to only 3 instructions per iteration, because the three loop induction variables fit in the registers. Finally, the multithreaded code with registers is able to keep the pipeline fully utilized by spawning eight concurrent threads.

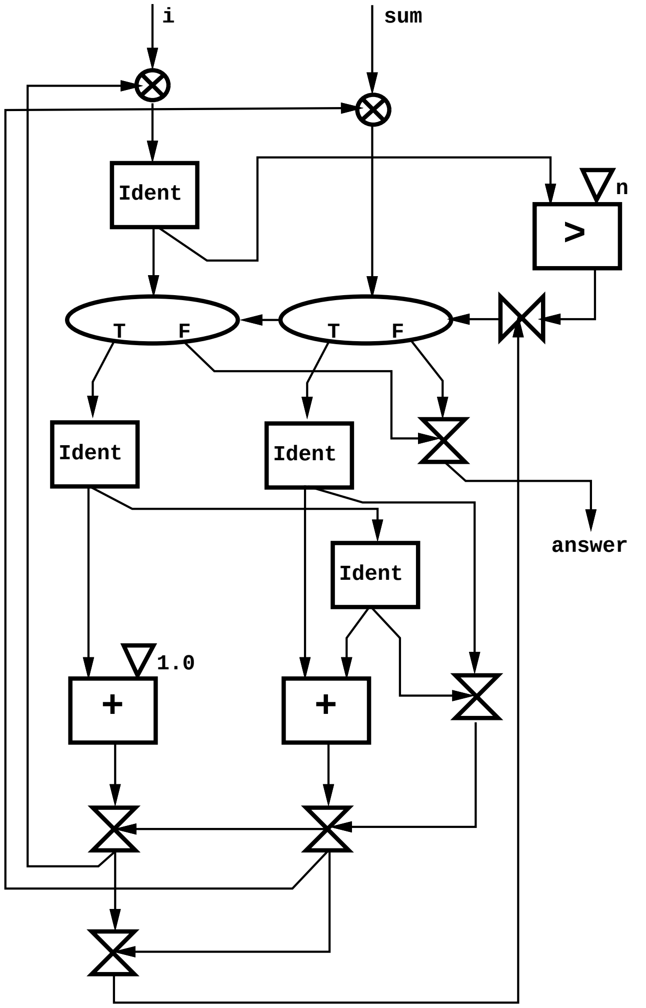
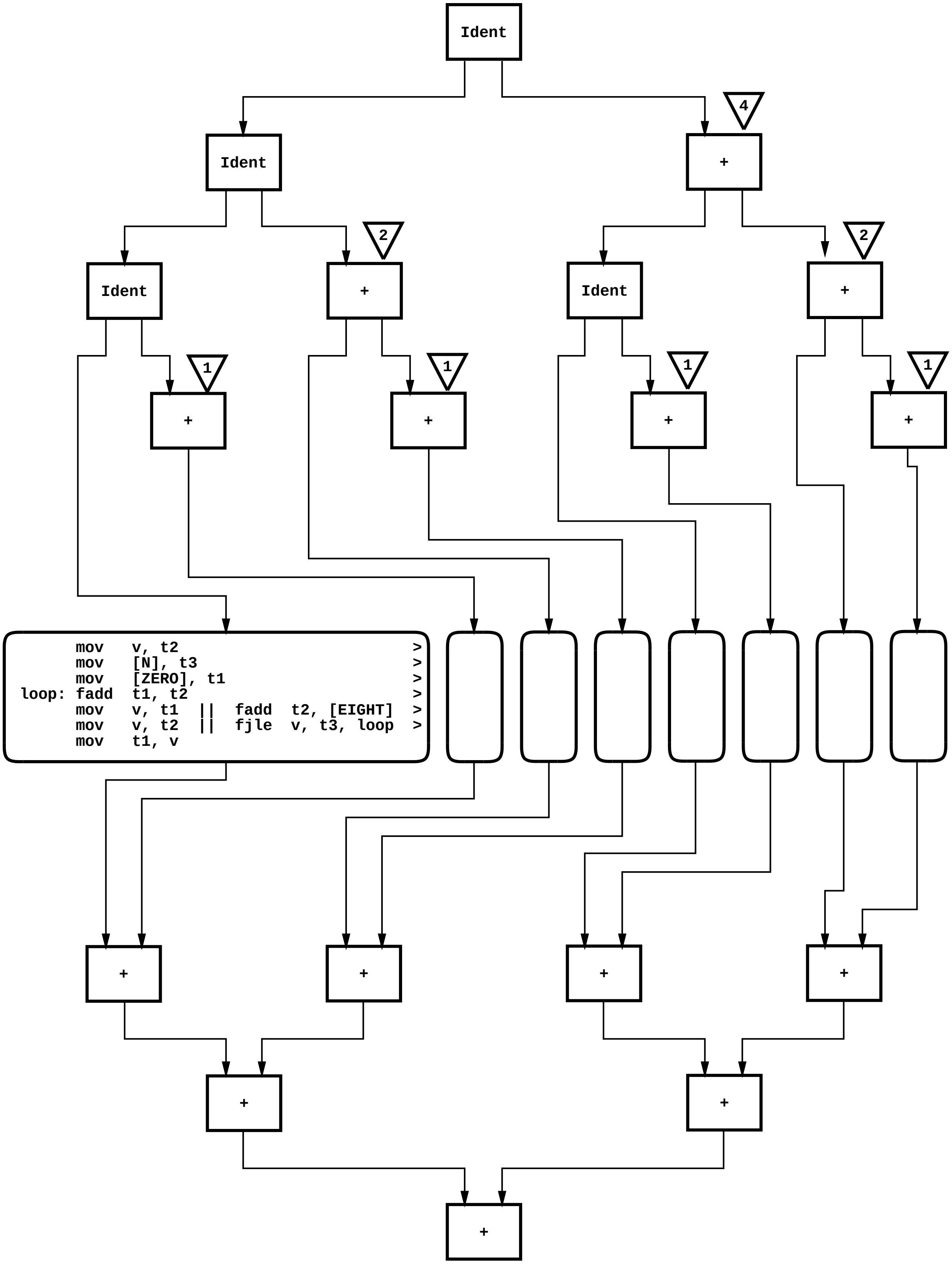


Figure 3: A Pure Dataflow Graph which Sums the First n Integers

Program	*Instructions / Iteration*	*Tokens / Iteration*	*Cycles / Iteration*
Pure Dataflow	14	22	71
Non-Register	7	7	56
Register	3	3	24
Parallel Register	3	3	3

While the performance improvements in this example are surprisingly large, we caution that this is a best-case scenario

[2] It was actually compiled by hand by one of the authors



```
      mov   v, t2                          >
      mov   [N], t3                        >
      mov   [ZERO], t1                     >
loop: fadd  t1, t2                         >
      mov   v, t1  ||  fadd  t2, [EIGHT]   >
      mov   v, t2  ||  fjle  v, t3, loop   >
      mov   t1, v
```

Figure 4: Using Parallelized Sequential Threads to Sum the First n Integers

because the loops only involve scalar values. If the loop required remote memory references, then the performance difference would be less dramatic because the responses would have to rejoin to the computation flow. Recall, temporary registers are not saved across join points.

5 Conclusion

Historically, dataflow and von Neumann computing have been viewed as radically different, and perhaps irreconcilable, models of computation. It is becoming increasingly clear, however, that the two models are simply the extrema of an architectural continuum. The von Neumann architecture can be extended into a multithreaded model by replicating the program counter and register set, and by providing primitives to synchronize among the several threads of control. The Denelcor HEP [14] interleaved up to 64 threads per processing element in order to hide the latency of remote memory references. More recently, researchers have suggested rapid context switching among fewer threads to mitigate the cost of occasional cache misses [15, 1]. While useful for dealing with unpredictable memory latency, these approaches do not fundamentally add to the basic von Neumann programming model because only a small number of threads can be managed efficiently by a processor. Thus, these machines may not be able to exploit the same degree of parallelism that is uncovered by a dataflow machine.

This has led some researchers to propose a kind of "virtualization" of the von Neumann model wherein a processor can efficiently manage a very large number of sequential threads which interact frequently. The VNDF hybrid machine maintains presence information on every memory location, and a thread can suspend or proceed as a function of the presence state [8]. P-RISC [10] attempts to reduce the implementation complexity of the hybrid approach by eliminating the presence bits in favor of explicit counting semaphores which are manipulated only at the beginning of a thread.

The model presented in this paper shares a number of the properties of the multithreaded von Neumann machines, but was derived by starting from a dataflow architecture and providing added imperative control over the usually asynchronous dataflow instruction scheduling rule. In the final analysis, the difference among the approaches is really one of emphasis. The multithreaded von Neumann models employ new techniques to reduce the cost of fine-grain synchronization while retaining the efficiencies of sequential scheduling. The multithreaded Monsoon model tries to improve the efficiency of computation while retaining the ability to perform extremely rapid synchronization.

From another perspective, multithreading is an acknowledgment of the empirical fact that *instruction-level* parallelism is probably better exploited in the context of a sequential thread executing in a well-engineered pipeline [7] (*e.g.*, superscalar RISC), rather than from the dynamic execution of the equivalent dataflow graph. However, we must caution against extending this line of reasoning too far. Clearly the greatest leveraging of the sequential structure is for scalar expressions where the timing of individual operations are known at compile time, and where it is easy for the hardware to relax the sequential order into parallel, pipelined execution. It is much more difficult for the same compiler/hardware structure to handle the cases when the execution time is long or unpredictable, *e.g.*, during a remote memory reference. Sequential processors rely upon reducing the probability of long latency operations with large caches, and have become even more vulnerable to cache misses as the processor cycle time in decreasing more rapidly than main memory access delays. This argues that a machine must, somehow, support multiple outstanding memory references and provide for inexpensive and rapid synchronization of the responses.

At the expression level, sequential threads will undoubtedly become preferable to pure dataflow graphs. A primary engineering objective will be to strike the right balance in the weight of the threads, both in terms of registers and cache requirements, and the ability to support fine-grain, rapid interactions among threads and between threads and global memory.

6 Acknowledgments

Mike Beckerle played an active role in defining, and then later testing, the machine programming model presented here. R. S. Nikhil provided much of the initial motivation for the effort through his seminal work on P-RISC. In fact, the `start` instruction is one of the key features of P-RISC, and our use of the same mnemonic suggests what we believe to be a deep relationship between his work and ours.

Research for this work was performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology and at the Motorola Cambridge Research Center. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

References

[1] A. Agarwal, Ben-Hong Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 104–114, Sealttle, Washington, May 1990.

[2] Arvind, D. E. Culler, and K. Ekanadham. The Price of Asynchronous Parallelism: an Analysis of Dataflow Architectures. In *Proceedings of CONPAR 88*, Univ. of Manchester, September 1988. British Computer Society — Parallel Processing Specialists. (also CSG Memo No. 278, MIT Laboratory for Computer Science).

[3] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *The International Journal of Supercomputer Applications*, 2(3), November 1988.

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

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, February 1987. (Also appears in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[6] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *Proceedings of Compcon90*, pages 88–93, March 1990.

[7] J. L. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[8] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, May 1988. (PhD Thesis, Dept. of EECS, MIT).

[9] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Int. Symp. on Computer Architecture*, pages 131–140, 1988.

[10] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989. To appear.

[11] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report TR432, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[12] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Archtecture*, 1990.

[13] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, Jerusalem, Israel, June 1989.

[14] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[15] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 1989 International Symposium on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 1989.