Can dataflow subsume von Neumann computing?

Rishiyur S. Nikhil
Arvind

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA

Abstract: We explore the question: "What can a von Neumann processor borrow from dataflow to make it more suitable for a multiprocessor?" Starting with a simple, "RISC-like" instruction set, we show how to change the underlying processor organization to make it multithreaded. Then, we extend it with three instructions that give it a fine-grained, dataflow capability. We call the result P-RISC, for "Parallel RISC." Finally, we discuss memory support for such multiprocessors. We compare our approach to existing MIMD machines and to other dataflow machines.

Keywords and phrases: parallelism, MIMD, dataflow, multiprocessors, multithreaded architectures

1 Introduction

Dataflow architectures appear attractive for scalable multiprocessing because they have mechanisms (a) to tolerate increased latencies and (b) to handle greater synchronization requirements [1]. Unfortunately, these architectures have been sufficiently different that it has been difficult to substantiate or refute this claim objectively. In this paper, we sidestep this issue and continue a recent trend towards a synthesis of dataflow and von Neumann architectures. In [4, 3], Ekanadham and Buehrer explored the use of dataflow structures in a von Neumann architecture. Papadopoulos' Monsoon dataflow processor [13] uses directly-addressed frames instead of an associative wait-match memory, showing a similarity to von Neumann machines. Iannucci [9, 10] explored a dataflow/von Neumann hybrid architecture. In this paper, we propose an architecture called P-RISC (for Parallel RISC). Not only can it exploit both conventional and dataflow compiling technology but, more so than its predecessors, it can be viewed as a dataflow machine that can achieve complete software compatibility with conventional von Neumann machines.

In Section 2, we describe the runtime storage model and the simple, RISC-like instructions on which we base our work. By RISC-like, our primary implication is that there are two categories of instructions—three-address instructions that operate entirely locally, *i.e.*, within a processing element, and load/store instructions to move data in and out of the processing element, without arithmetic [7, 14, 15, 18]. Further, the instructions are simple and regular, suitable for pipelining. Nevertheless, our storage model is unusual.

In Section 3, we change the underlying processor organization to make it multithreaded (*a la* HEP), itself an improvement for multiprocessors. In Section 4, we extend it with three instructions, giving it a fine-grained, dataflow capability, making it an even better building block for multiprocessors. Section 5 discusses memory support for such multiprocessors; Section 6 discusses some of the serious unresolved engineering issues, and Section 7 concludes with a comparision with other work. Our concern here is with asynchronous, MIMD models. This covers most current and proposed parallel machines, such as the HEP, BBN Butterfly, Intel Hypercube, IBM RP3, Cray YMP, IBM 3090, Sequent, Encore, *etc.*, and excludes machines like the Connection Machine, Warp, and VLIW machines.

2 The runtime model

2.1 Trees of frames, and heaps

Consider the following program:

```
procedure h(x) ... ;

procedure g(y) ...  h(e) ... ;

procedure M() ... g(e1) ... h(e2) ... ;
```

A sequential implementation uses a *stack* of *frames* (activation records) which goes through the configurations of Figure 1 (our stack grows upward). At each instant, only the code for the topmost frame is active—lower frames are dormant. In a parallel

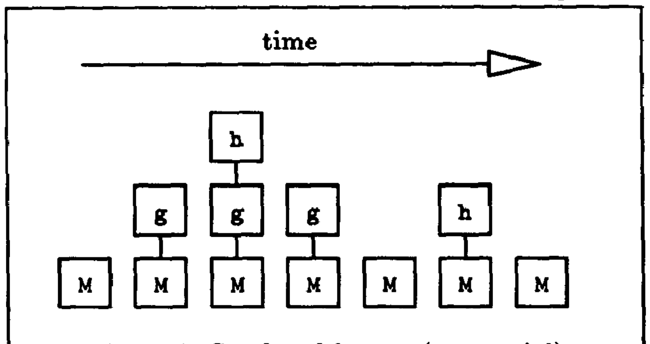


Figure 1: Stacks of frames (sequential)

implementation, however, M may call g and h concurrently, and g may call h concurrently as well, *i.e.*, all frames may exist concurrently. We must generalize the structure to a *tree* of frames (Figure 2). Further, at each instant, the code for any of the frames can be active, not just at the leaves.

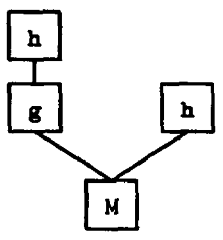


Figure 2: Tree of frames (parallel)

Another difference is in loops. Sequential loops normally use a single frame. In parallel loops, however, we need many frames in order that multiple iterations may run concurrently. Still, it is a tree structure: all frames for iterations of a loop have the same parent, and any procedure calls from the loop body are subtrees above the iteration frames. The set of paths to the root frame, in the parallel implementation's tree of frames, corresponds to the states of the sequential implementation's stack.

Of course, frames are not enough. In many modern programming languages, *e.g.*, Lisp, Smalltalk, ML, Id, *etc.*, it is possible for the lifetimes of data structures to differ from the the lifetimes of frames. Thus, data structures must be allocated on a global *heap*.

A pair of frames can share values in two ways. They may both refer to a common ancestor frame (by lexical scoping), or they may both refer to a data structure in the heap. In this paper, we will only consider the latter mechanism, as lexical scoping of scalars can always be eliminated by "lambda-lifting" [11].

To summarize: our runtime model of storage consists of a tree of frames and a global heap memory, with frames containing pointers into the heap.

2.2 Processing Elements, Continuations and sequential "RISC" code

How is this abstract storage model mapped to a multiprocessor? We assume an interconnection of Processing Elements (PEs) and Heap Memory Elements (see Figure 3). Each PE has *local memory for code and frames*. Even though the memories may be physically distributed, we assume a single, *global address space* for all memories.

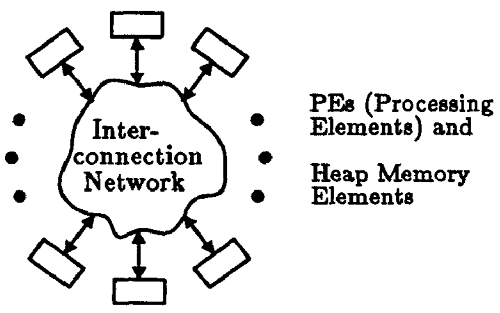


Figure 3: P-RISC multiprocessor organization

At each instant, a PE runs a *thread* of computation, which is completely described by an instruction pointer IP, and a frame pointer FP. The IP points into code in the PE, and the FP points at a frame in the PE (see Figure 4). We can regard this pair of pointers as a *continuation*, or "flyweight" process descriptor, and we use the notation <FP.IP>. They correspond exactly to the "tag" part of a token in the terminology of tagged-token dataflow. It is convenient for continuations to have the same size as other values, *e.g.*, integers and floats, so that continuations can be manipulated as values.

As a running example, we use the following procedure that computes the inner-product of two vectors A and B of size n:

```
def vip A B =
     { s = 0
     In {for i <- 1 to n do
            next s = s + A[i] * B[i]
        finally s}} ;
```

S is zero for the first iteration of the loop. For each subsequent iteration, s has the value from the previous iteration plus the product of two vector components. The value of the entire expression is the value

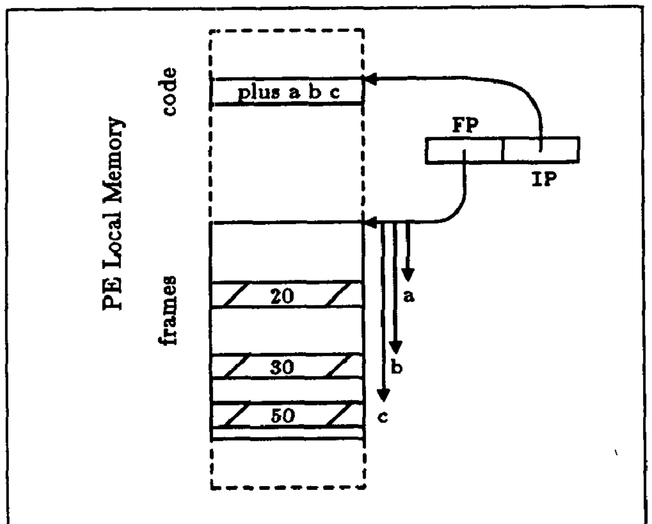


Figure 4: A continuation

```
  load-immed    0   s
  load-immed    1   i
LOOP:
  compare i     n      b
  jgt     b     DONE
  plus    A     i      aA
  load    aA    Ai
  plus    B     i      aB
  load    aB    Bi
  mult    Ai    Bi     AiBi
  plus    s     AiBi   s
  incr    i     i
  jump    LOOP
DONE:
  ...
```

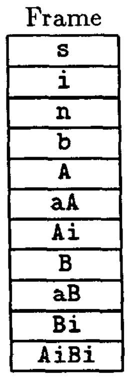


Figure 5: Sequential code for vip

of s in the final iteration. We use the language Id [12], but other parallel languages are also acceptable.

Every instruction is executed with respect to a current FP and IP. All arithmetic/logic operations are 3-address frame-to-frame operations, *e.g.*:

```
plus s1 s2 d
```

which reads the contents of frame locations FP+s1 and FP+s2 and stores the sum at FP+d. "Compare s1 s2 d" compares the values at FP+s1 and FP+s2 and stores a condition code at FP+d. A j*cond* instruction:

j*cond* s IPt

(for various *conds*) reads a condition code from FP+s and changes IP to IP+1 or IPt accordingly. For computed jumps, a variation would be to pick up IPt from the frame.

"Load a x" and "Store x a" move data between frame location FP+x and the heap location whose address is in the frame at FP+a.

The instruction set is RISC-like in the following sense. All arithmetic operations are local to the PE. Load and store are the only instructions for moving data in and out of the PE, and they do not involve any arithmetic. Thus, instruction fetches and frame accesses involve local PE memory only, and no network traffic. Further, the arithmetic instructions are simple and regular so that they can be pipelined.

Sequential code for vip is shown in Figure 5 with the frame shown on the right (for expository reasons, we use more frame slots than necessary).

2.3 Frames as register sets

In most RISCs there is a local register set, and both frames and heap are non-local [15, 7]. To reduce non-local traffic, one can have more registers, have multiple register sets (as in the HEP and Berkeley RISC [14]), or provide instruction and data caches.

In P-RISC, we think of a frame as being local and synonymous with a register set. The collection of frames on a PE is regarded as a collection of register sets, a particular register set being identified by an FP.

The total frame memory in a PE is likely to be large. This is *inconsistent* with the requirement for two reads and a write per cycle. We think that a high-speed cache that holds some subset of frames is necessary. An attempt to execute a continuation that refers to a missing frame will cause a fault, triggering a swap of frames between the cache and frame memory. Such a cache implementation would perhaps be simplified if frames were of fixed size. The compiler may have to split large code blocks to meet this requirement.

2.4 Problems: memory latency and distributed memory

A major problem in our vip code is the latency of the loads. The round trip to heap memory can take significant time, especially in a multiprocessor. A cache may help, but even caches are less effective in multiprocessors. Thus, a processor may idle during a load, thereby reducing performance. In bus or circuit-switched networks, long-latency loads can also interfere mutually, also degrading performance. Ideally,

1. Loads should be *split-phase* transactions (request and response) so that the path to memory is not occupied during the entire transaction.
2. The processor should be able to issue multiple loads into the network before receiving a response, *i.e.*, the network should be a pipeline for memory requests and responses.
3. The processor should be able to accept responses in a different order from that in which requests were issued. This is especially true in a multiprocessor, where distances to memories may vary.
4. The processor should be able to switch to another thread of computation rather than idle.

Many previous processor designs address this issue to varying degrees. The Encore Multimax uses split-phase bus transactions but a particular PE can have only one outstanding load. The CDC 6600 [18], the Cray [16], and some RISCs can pipeline memory requests, but requests must come back in the same order. The IBM 360/91 could pipeline memory requests and receive them out of order, but there was a small limit to the number of such outstanding requests. Further, in all these cases there is significant added complexity in the processor circuits. A more detailed discussion of this issue is in [1]. An alternative is to make the processor multithreaded (*à la* HEP [17]).

3 Multithreaded RISC

Maintaining the same instruction set, we change the underlying processor organization to support fine-grained interleaving of multiple threads.

In most high-performance machines (including RISCs), the instruction pipeline is single-threaded, *i.e.*, consecutive entities in the pipeline are instructions from the same thread, from addresses IP, IP+1, IP+2, and so on. This introduces some extra complexity in the detection and resolution of inter-instruction hazards. Further, long latency instructions such as loads disrupt the pipeline and add complexity. The HEP's pipeline, on the other hand, was time-multiplexed among several threads [17]. On each clock, a different thread descriptor was inserted into the pipe. As they emerged from the end of the pipe, they were recirculated via a queue to the start of the pipe. Thus, there was no hazard between consecutive instructions in a particular thread. Further, when a thread encountered a load, it was taken aside into a separate pool of threads waiting for memory responses; thus, threads did not block execution of other threads during loads. Unfortunately, the number of threads that could be interleaved in the pipe and the number of threads that could be waiting for loads was limited.

For multithreaded RISC, we generalize the HEP approach to an *arbitrary* number of interleaved threads. The organization of the PE is shown in Figure 6. Recall that our thread descriptors (continuations) are `<FP.IP>` pairs. Since they are circulated in the processor, we also refer to them as *tokens*. Tokens reside in the token queue (like the HEP's PSW queue). On each clock, a token `<FP.IP>` dequeued and inserted into the pipeline, fetching the instruction at `IP` and executing it relative to the frame at `FP`. The pipeline consists of the traditional instruction fetch, operand fetch, execution and operand store stages. At the end of the pipe, tokens are produced specifying the continuation of the thread; these tokens are enqueued.

For arithmetic/logic instructions, the continuation is simply `<FP.IP+1>`. For the `jump` instruction, the continuation is simply `<FP.IPt>`. For `j`*cond* instructions, the continuation is either `<FP.IP+1>` or `<FP.IPt>`, depending on the condition code in `FP+x`.

The first interesting difference arises in the `load` instruction. A heap address a is fetched from frame location `FP+a`, and the following message is sent into the network:

`<READ,`a`,FP.IP+1,x>`

There is *no continuation* inserted into the token queue! Meanwhile, the pipeline is free to process other tokens from the token queue. Some of them, in turn, may be **loads**, pumping more `READ` messages into the network.

The `READ` messages are processed by Heap Memory Elements, which respond with `START` messages:

`<START,`v`,FP.IP+1,x>`

When such a message enters the PE, the value v is written into the location `FP+x`, and the token `<FP.IP+1>` is inserted into the token queue. Thus, the thread descriptor travels to the heap and back.

A **store** fetches a heap address a from frame location `FP+a`, and a value v from `FP+x`, and sends the message:

`<WRITE,`a`,`v`>`

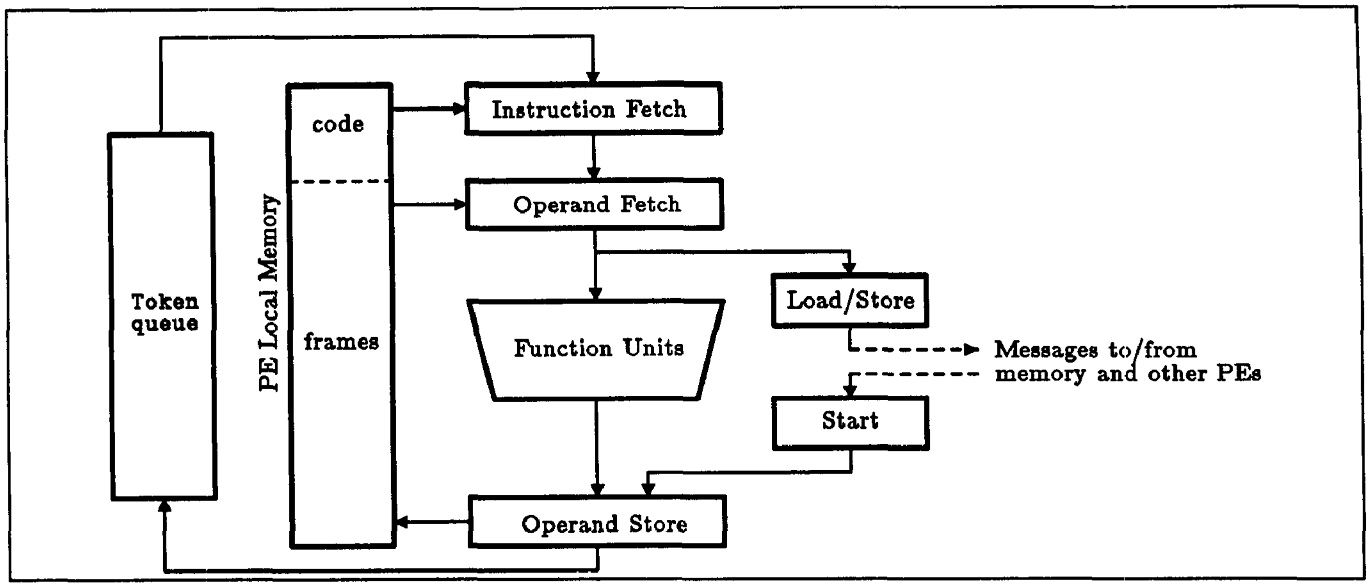
into the network. A Heap Memory Element receives this, and stores the value. Meanwhile, the token `<FP.IP+1>` emerges from the pipe and is inserted into the token queue.

3.1 Discussion

Notice that we have achieved our goals:

- loads are split-phase transactions,
- any number of loads can be pipelined into the communication network,
- responses can come back in any order

PE organization:



Instruction set summary:

Format	Frame operations	Continuations	Outgoing messages
Ordinary RISC-like instructions:			
`op s1 s2 d`	`[FP+s1] op [FP+s2]` → `FP+d`	`<FP.IP+1>`	—
`jump IPt`	—	`<FP.IPt>`	—
`j`*cond* `x IPt`	`[FP+x]` →	`<FP.IPt>` or `<FP.IP+1>` (depending on `[FP+x]`)	—
`load a x`	`[FP+a]` →	*none*	`<READ,[FP+a], FP.IP+1, x>`
`store x a`	`[FP+x],[FP+a]` →	`<FP.IP+1>`	`<WRITE,[FP+a], [FP+x]>`
Incoming messages (from memory, other PEs):			
`<START,v,FP.IP,y>`	`v` → `FP+y`	`<FP.IP>`	—
P-RISC instructions (extensions for fine-grained parallelism):			
`fork IPt`	—	`<FP.IP+1>, <FP.IPt>`	—

`join x`	toggle `[FP+x]`	if `[FP+x]`: *none*
		if ¬ `[FP+x]`: `<FP.IP+1>`
			—

`start v c d`	`[FP+v],[FP+c],[FP+d]` →	*none*	`<START,[FP+v], [FP+c], [FP+d]>`
`loadc a x IPr`	`[FP+a]` →	`<FP.IP+1>`	`<READ,[FP+a], FP.IPr x>`

Figure 6: P-RISC Processing Element (PE) organization and instruction set summary

- The processor interleaves threads on a per-instruction basis, and is not blocked during loads. The number of threads it can support is the size of the token queue. Assuming enough tokens in the token queue, the pipeline can be kept full during memory loads—the processor never has to idle.

In the HEP, too, each thread could issue a load. The main difference is that the HEP had a small limit on the number of threads and outstanding loads. The HEP had another limitation shared by our multithreaded PE: even though there can be many outstanding loads from multiple computations, a *particular* computation can have no more than one outstanding load. We correct this situation next.

4 P-RISC: An extension for fine-grained parallelism

In any multithreaded system, there must be a way to initiate new threads and to synchronize two threads. Often, these involve operating system calls, traps, pseudo-instructions, *etc.* It is difficult to make this cheap and so one avoids fine-grained parallelism which, in turn, reduces the exploitable parallelism in programs.

We extend the multithreaded RISC to P-RISC with two instructions for thread initiation and synchronization. It is important that these are simple *instructions*—not operating system calls—that are executed entirely within the normal processor pipeline (again, please refer to Figure 6):

- `Fork IPt` is just like a jump, except that it produces *both* `<FP,IPt>` *and* `<FP,IP+1>` tokens as continuations.
- `Join x` toggles the contents of frame location `FP+x`. If it was zero (empty) it produces no continuation. If it was one (full) it produces `<FP,IP+1>`.

4.1 Inner-product revisited

Figure 7 shows, in outline, a new control flow in order to do the two loads concurrently. The corresponding code is shown in Figure 8 along with its frame. The frame has an additional location `w` used by the `join`, initialized to zero (empty) in the third statement.

The `fork` produces two continuations: `<FP.LOADAi>`, *i.e.*, the next instruction, and `<FP.LOADBi>`. Then, both address calculations and loads are executed concurrently. When the `load` in the `LOADAi` sequence is executed, it sends the continuation `<FP.LOADAi+2>` in its message to heap memory (`LOADAi+2` points at the `jump SYNCH` instruction). When the response arrives, the value is written into frame location `Ai`, the

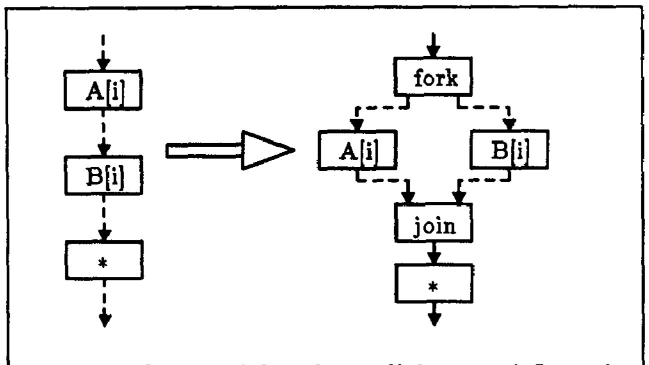


Figure 7: Sequential and parallel control flows for the two loads

```
   load-immed  0  s
   load-immed  1  i
   load-immed  0  w   % new
LOOP:
   compare i  n  b
   jgt     b  DONE
   fork    LOADBi     % new
LOADAi:
   plus  A i  aA
   load  aA  Ai
   jump  SYNCH        % new
LOADBi:
   plus  B i  aB
   load  aB  Bi
SYNCH:
   join  w            % new
   mult  Ai Bi  AiBi
   plus  s AiBi  s
   incr  i  i
   jump  LOOP
DONE:
   ...
```

```
 Frame
 s
 w
 i
 n
 b
 A
 aA
 Ai
 B
 aB
 Bi
 AiBi
```

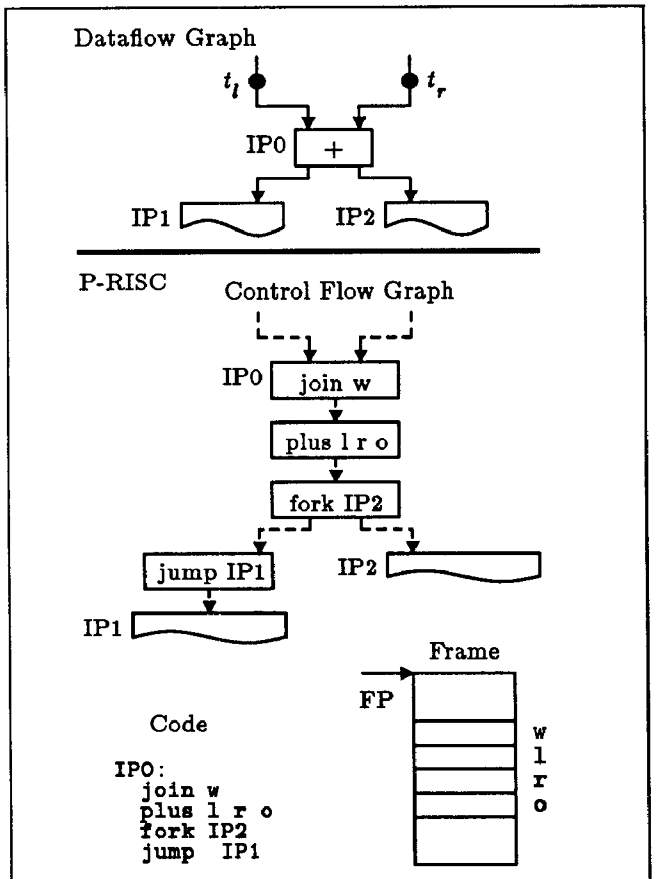
Figure 8: Code for concurrent loads

jump is taken, and `join w` is executed. When the `LOADBi` sequence is executed, it sends the continuation `<FP.SYNCH>` in its message to heap memory. When this response arrives, the value is written into frame location `Bi` and `join w` is executed.

Thus, `join w` is executed twice, once after the completion of each `load`. The first time, it toggles the location `FP+w` from empty to full and nothing further happens. The second time, it toggles it from full to empty and execution continues at the `mult` instruction. Note that the order in which the loads complete does not matter. Also, note that the location `FP+w` is ready for the next iteration as it has been reset to empty.

4.2 Fine-grained dataflow

With lightweight **fork** and **join** instructions, it is possible to simulate the fine-grained asynchronous parallelism of pure dataflow. For example, the top of Figure 9 shows a classical dataflow "+" instruction (at address IP0). When tokens t_l and t_r arrive on its input arcs carrying values, it "fires," *i.e.*, consumes the tokens and produces tokens carrying the sum on each of its output arcs, destined for instructions at IP1 and IP2.



```
IP0:
  join w
  plus l r o
  fork IP2
  jump  IP1
```

Figure 9: Dyadic dataflow "+" instruction in P-RISC

The P-RISC control graph is shown next, followed by the P-RISC code and frame. The slots `l` and `r` are used to hold the left and right input values, respectively; the slot `o` is used to hold the output value, and the slot `w` is used for synchronization.

Corresponding to the dataflow graph producing t_l, there would be some P-RISC code that stores the left input value in `l` and inserts `<FP.IP0>` in the token queue. Similarly, some P-RISC code would store the right input value in `r` and insert an identical token `<FP.IP0>` in the token queue. Thus, getting past the `join` is a guarantee that both inputs are available. The last two instructions place the two tokens `<FP.IP1>` and `<FP.IP2>` in the token queue. With a little analysis, it is fairly easy to do better, *i.e.*, to have fewer `joins` than might be required by such a direct translation.

4.3 Extracting more parallelism

So far, the iterations of our inner-product program went sequentially, with just the two loads of each iteration proceeding concurrently. In principle, however, all $2n$ loads are independent, and all n multiplications are independent of each other. A loop compilation scheme that achieves this concurrency is described in [2, 19], along with optimizations permitting reuse of frames. Here, instead, we outline a recursive reformulation that is easier to describe:

```
def vip A B = vip_aux A B 1 ;

def vip_aux A B i =
    if i > n then 0
    else
      A[i]*B[i] + (vip_aux A B (i+1)) ;
```

The recursive call can be done concurrently with the two loads Further, the multiplication can be done before the recursive call has completed, as soon as both loads have completed (Figure 10). All $2n$ `loads` can be issued in parallel, the responses can come back in any order, and the multiplications are performed in an arbitrary order, automatically scheduling themselves as the load-responses arrive.

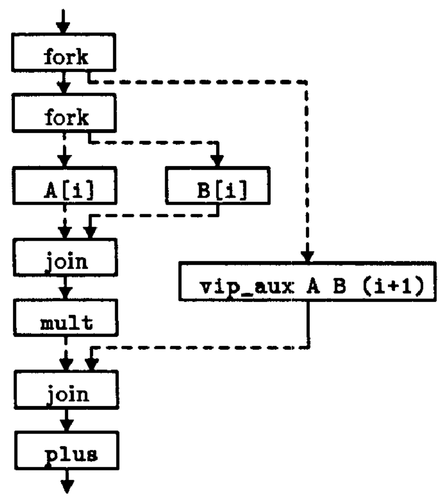


Figure 10: Control graph for concurrent loads and iterations

The additions in our program still proceed sequentially, and it would be easy to write a divide-and-conquer version where this is not so. However, the program as it stands dramatically illustrates that even in "apparently" sequential programs, there is much parallelism to be exploited by our architecture—all index calculations, loads and multiplications can be done in parallel.

4.4 Procedure call/return

There is no specific architectural support for procedure linkage— it is purely a compilation issue. In [2, 19], a mechanism is described that is capable of supporting the high degree of parallelism in nonstrict function calls. We summarize it here in P-RISC terms, keeping in mind that it is always possible to constrain it for less parallelism.

Suppose g calls h with n arguments. We can visualize this as storing n values into h's frame and initiating n corresponding threads in h. Similarly, to return m results, we store m values in g's frame and initiate m threads in g.[1] All this can be done asynchronously, *i.e.*, the n arguments and m results can be sent in any order. A fact often surprising to those unfamiliar with nonstrict functional languages is that it is possible to return results before all the arguments have been sent! For example, if h computes a vector-sum, it could allocate and return a pointer to the result vector even before it received the input vectors (assuming the size is known beforehand). This caller/callee overlap is a tremendous source of parallelism.

There is only one synchronization requirement—the j'th thread in h must not begin until the j'th argument has been stored in its frame (and similarly for returned results). For this we use a new instruction:

```
start dv dFPIP dd
```

which reads a value v from FP+dv, an <FP.IP> continuation from FP+dFPIP and an offset d from FP+dd, and sends the message:

<START,v,FP.IP,d>

to the destination PE. No continuation is placed in the token queue. We have already seen START messages (memory responses); when one arrives at a PE, it writes a value into a frame and initiates a thread.

A possible linkage convention is this. Let the first instruction in h be at IPh. Let FPh point at its frame. In g, we fork a thread for each argument. The j'th thread ends in a start instruction that sends:

<START,arg$_j$,FPh.IPh+j,j>

This deposits the argument into FPh+j and initiates a thread at IPh+j. Returned results are handled similarly. The only subtlety is that g needs to send extra arguments to h that describe its own slots and continuations that await the results.

[1] Of course, other threads may be active in g throughout.

The only remaining issue is frame allocation and deallocation. Since simple stack allocation will not do, g issues a call to a *frame manager* which returns a pointer to a frame allocated from a free list that it maintains in frame memory. The manager call is itself a split-phase transaction, so that other computations may proceed concurrently. The frame manager is not an ordinary procedure. It has a fixed, known context (frame), and is coded as a critical section, responding to requests in the order that they arrive. Requests that arrive while it is servicing a previous request are queued, possibly in I-structure memory. In a multiprocessor, there will typically be a frame manager on each PE, and these managers also perform load balancing to ensure that frames are evenly distributed across PEs.

After all results are received, g sends FPh back to the frame manager for deallocation. Many details are described in [2, 19], including automatic recirculation of frames and the generation of "self-cleaning" graphs to guarantee that a frame is not returned while there are still references to it. Frame allocation and deallocation does not need general garbage collection.

4.5 Discussion

Every join is fetched and executed twice. The first time, it introduces a bubble into the pipe because the thread dies. These bubbles correspond to the bubbles in the pipe of a Tagged-Token Dataflow Architecture when the wait-match operation fails (first token to arrive at a dyadic operator) [2, 5, 8].

Arithmetic instructions do two reads and one write into a frame. There is a potential hazard if two successive instructions in the pipe compete for the same frame location. To avoid this, many machines use reservation bits to stall the pipe. In a multithreaded machine, successive instructions can be from unrelated threads and are thus less likely to compete for the same location. This may mitigate, but not eliminate, the problem. In the HEP, an instruction was converted into a no-op and recirculated if it accessed an empty register.

If code is systematically and directly compiled from dataflow graphs (as in the "+" example in Section 4.2) we can, in fact, guarantee that, with one exception, such hazards will not arise—there will always be an adequate number of joins to prevent races between normal instructions. The exception is that there can still be a race between two join instructions. Each join reads a location, tests it, toggles it, and writes it back, and this must be atomic. If the next instruction in the pipe is a join for the same rendezvous location, the pipeline must be stalled.

We described a join as referring to an entire location, even though it needs only one bit. Several variations are possible. The bits for all `join` locations in a frame could be packed into a few frame locations. Or, we could generalize it to an n-way synchronization: the frame location `x` is initialized to $n-1$; a "`join x` n" instruction decrements it and dies if it is non-zero; if zero, it is reinitialized to n and the thread continues.

The `start` instruction and `START` message take three parameters: a value, a tag `<FP.IP>` and an offset `d`. An obvious variation is to combine the latter two: `<FPd.IP>`, where `FPd = FP+d`. The value is stored directly at `FPd` and the tag `<FPd.IP>` enqueued. The thread at IP can then adjust the frame pointer back to `FP` by subtracting `d`.

We can see that `loads` are frequently surrounded by `forks` and `jumps` to gain concurrency. It is thus useful to have the following "load and continue" instruction:

```
loadc a x IPr
```

It picks up a heap address a from `FP+a`, sends the message:

```
<READ,a,FP.IPr,x>
```

and continues at `IP+1`. Thus, it does an implicit fork. It simplifies the code of Figure 8:

```
LOOP:
   ...
   jgt    b   DONE
   plus   A   i  aA
   loadc  aA  Ai SYNCH
   plus   B   i  aB
   load   aB  Bi
SYNCH:
   ...
```

5 Memory support for fine-grained parallelism

We have seen the following behavior for a Heap Memory Element. It receives two kinds of messages and responds with one kind of message. On receiving:

```
<READ,a,FP.IP,d>
```

it reads value v from location `a` and responds with:

```
<START,v,FP.IP,d>
```

On receiving:

```
<WRITE,a,v>
```

it writes the value v into address `a`.

However, this is inadequate, because it does not provide synchronization—a **READ** for a location may arrive from PE_0 before the corresponding **WRITE** from PE_1. To solve this, we extend the behavior of Heap Memory Elements in the direction of "I-Structures" [2]. Every location has additional presence bits that encode a state for that location.

For producer-consumer situations, we introduce two new types of messages. On receiving:

```
<I-READ,a,FP.IP,d>
```

if the location `a` is full, it behaves like an ordinary **READ**. If it is empty, the location contains a "deferred-list" (initially nil). Each list element contains the `(FP.IP,d)` information of a pending read. The information in the current `I-READ` message is added to the list. On receiving:

```
<I-WRITE,a,v>
```

if the location `a` is empty, for each `(FP.IP,d)` in the deferred list, the memory sends out a message:

```
<START,v,FP.IP,d>
```

and, finally, v is written into `a`. Thus, `I-READs` can safely arrive before the corresponding `I-WRITEs`. Of course, this assumes that the location is written only once; it may be possible to guarantee this at the language level and/or by compiler analysis (*e.g.*, it is easy in functional and logic languages).

There are, of course, other useful messages that can be processed by Heap Memory Elements, such as exchanges, test-and-sets, *etc.*

6 Implementation issues

Our development of P-RISC went as follows:

- Start with a RISC-like instruction set, *i.e.*, a `load/store` instruction set in which most instructions are simple, regular, 3-address frame-to-frame operations. Many variations on our instruction set are possible. In particular, it is possible to take the instruction set of an existing, commercial RISC and to generalize it in the direction of a P-RISC. This would facilitate a smooth transition for software development. One of the attractions of P-RISC is that it can use both conventional and dataflow compiling technology.
- Make it multithreaded, using a token queue and by circulating `<FP.IP>` tokens (thread descriptors) through the processor pipeline and token queue. Loads are split-phase—request to memory and response, so that the processor pipeline and the interconnection network are not blocked in the interim. Request and response messages are identified by the full continuation, so that the synchronization namespace is the full address space, network traffic can be pipelined, and responses may arrive in any order.

- Introduce `fork` and `join` instructions that are executed in the processor pipe, and a `start` instruction to communicate between frames on different PEs. The `loadc` instruction is a useful optimization.
- Introduce synchronization in the Heap Memory Elements using I-structure semantics.

Frames as register sets

This is perhaps the most serious implementation issue in P-RISC. Our dataflow experience indicates that total frame memory requirements are likely to be large. As mentioned earlier, the necessity for two reads and a write on each cycle will probably require an implementation that uses a cache for a subset of frames. With current technology, it should be possible to build a cache that can hold hundreds of frames. When a token referring to a missing frame is dequeued, it will trigger a fault that causes a frame to be swapped between the cache and the frame memory. It may be possible to continue executing other tokens during the swap. To avoid thrashing, the compiler/hardware would have to give preference to tokens in the token queue that refer to frames currently in the cache. Note that in P-RISC, a small number of frames can support a large number of concurrent threads, because of the fine-grained concurrency *within* a procedure activation or loop iteration.

Storage classes

We described four kinds of stores—code, token queues, frames and the global heap. For completeness, *e.g.*, for loading programs, debugging, garbage collection, *etc.*, additional instructions are necessary to read and write all stores. An interesting possibility would be to consider all local memory as a temporary copy taken from the global store.

Instruction scheduling

In our description, the processor pipe and the token queue formed a ring around which tokens were circulated. An alternative is this: as long as a thread does not die (due to `load` or `join`), continue executing the same thread, using the normal "`IP+1`" scheduling of a von Neumann processor; extract a token from the token queue only when a thread dies. This solution reintroduces the complexity of inter-instruction hazard detection and resolution, but it does allow adjacent instructions in a thread to communicate via a small set of named high-speed registers. Also, it could improve locality for a cache-based frame memory.

Compilation issues

Synchronization occurs in two places—in frames during a `join`, and in heap memory with I-structure operations. The former is more efficient because it needs no queueing—there is exactly one continuation. An I-structure write, on the other hand, can trigger an arbitrary number of continuations since it may have any number of consumers.

Compiler analysis of lifetimes and accessibility of variables can reveal additional information that can make use of the cheaper synchronization and better locality of frame storage by allocating data structures in frames instead of on the heap.

7 Comparison with other work

The work of Ekanadham and Buehrer was an important step in exploring the use of dataflow structures in a von Neumann architecture [4, 3]. Halstead and Fujita proposed a multithreaded processor architecture for Lisp [6].

Papadopoulos' Monsoon architecture [13] is a pure dataflow architecture in the sense that tokens not only schedule instructions but also carry data. Interprocessor tokens are identical to intra-processor tokens. Since tokens carry data, only one frame operation is required in each pass through the pipe, unlike P-RISC's three frame operations. While it is clear how to use dataflow compiling techniques for Monsoon, it is not clear how to use compiling techniques from conventional processors. Further, unlike P-RISC, it is not easy to imagine an implementation that uses the conventional "`FP+1`" instruction scheduling.

Closest to P-RISC is Iannucci's dataflow/von Neumann hybrid architecture [9, 10]. Every frame location has full/empty presence bits. A `load` instruction sends a request to memory along with the address of the destination frame location, and execution continues at the next instruction. An operation that tries to read an empty frame location traps, storing its process descriptor in that location and marking it "pending"; execution resumes at some other thread. When a response returns from memory to a pending frame location, the value is exchanged with the process descriptor residing there, and the process is re-enabled. Thus, Iannucci's architecture has split-phase loads, loads can be pipelined, responses can come in any order, and the namespace for waiting threads is the address space of local memory.

The primary difference between Iannucci's machine and P-RISC is that he has presence bits on every location in frame memory, and synchronization can occur in *any* instruction by using a synchronizing frame access operation. In P-RISC, frame memory does not have presence bits; instead, some locations are interpreted as full/empty synchronization locations. Synchronization occurs only at `join` instructions. Thus, P-RISC is a simpler architecture, but it may execute more instructions since synchronization is separated from arithmetic instructions.

These comparisons are by no means exhaustive, and much detailed design and experimentation remains to evaluate P-RISC. We hope this paper will stimulate research in this direction. The focus of current work at MIT is to take an existing RISC implementation and modify/extend it in the direction of P-RISC, while maintaining software compatibility with the original processor.

Acknowledgements: We thank current and past members of the Computation Structures Group at MIT, especially G.P. Papadopoulos and R.A. Iannucci, for an exciting and stimulating environment. We appreciate greatly the ideas and comments of K. Ekanadham of IBM, and the comments of S.L. Peyton Jones and K.J. Ottenstein. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

References

[1] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. DFVLR Conf. 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 1987.

[2] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, 1989 (to appear). An earlier version appeared in *Proc. PARLE, Eindhoven, The Netherlands*, Springer-Verlag LNCS 259, June, 1987.

[3] R. Buehrer and K. Ekanadham. Incorporating Dataflow Ideas into von Neumann Processors for Parallel Execution. *IEEE Trans. on Computers*, C-36(12):1515–1522, Dec. 1987.

[4] K. Ekanadham. Multi-tasking on a dataflow-like architecture. Technical Report RC 12307 (55198), IBM T.J.Watson Res. Ctr., Yorktown Heights, NY, Nov. 1986.

[5] J. R. Gurd, C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Comm. of the ACM*, 28(1):34–52, Jan. 1985.

[6] R. H. Halstead, Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. 15th. Annual Intl. Symp. on Comp. Arch., Honolulu, Hawaii*, June 1988.

[7] J. Hennessey. VLSI Processor Architecture. *IEEE Trans. on Computers*, C-33(12):1221–1246, Dec. 1984.

[8] K. Hiraki, S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Technical report, Computer Systems Division, Electrotechnical Lab., 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[9] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Lab. for Computer Science, 545 Tech. Sq., Cambridge, MA 02139, May 1988.

[10] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th. Annual Intl. Symp. on Comp. Arch., Honolulu, Hawaii*, June 1988.

[11] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. Func. Prog. Langs. and Comp. Arch., Nancy, France, Springer-Verlag LNCS 201*, Sept. 1985.

[12] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Lab. for Computer Science, 545 Tech. Sq., Cambridge, MA 02139, Aug. 1988.

[13] G. M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. Technical Report TR-432, MIT Lab. for Computer Science, 545 Tech. Sq., Cambridge, MA 02139, Aug. 1988.

[14] D. Patterson. Reduced Instruction Set Computers. *Comm. of the ACM*, 28(1):9–21, Jan. 1985.

[15] G. Radin. The 801 Minicomputer. In *Proc. ACM Symp. on Arch. Support of Prog. Langs. and Op. Sys.*, pages 39–47, Mar. 1982.

[16] R. Russell. The CRAY-1 Computer System. *Comm. of the ACM*, 21(1):63–72, Jan. 1978.

[17] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proc. 1978 Int'l Conf. on Parallel Processing*, pages 6–8, 1978.

[18] J. Thornton. Parallel Operations in the Control Data 6600. In *Proc. SJCC*, pages 33–39, 1964.

[19] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Lab. for Computer Science, 545 Tech. Sq., Cambridge, MA 02139, Aug. 1986.