LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Can dataflow subsume von Neumann computing?

Computation Structures Group Memo 292
November 15, 1988

Rishiyur S. Nikhil
Arvind

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 N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Can dataflow subsume von Neumann computing?

Rishiyur S. Nikhil
Arvind

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA
nikhil@xx.lcs.mit.edu arvind@xx.lcs.mit.edu
(617)-253-0237 (617)-253-6090

Abstract

The aim of this paper is to explore the question: "What can a von Neumann processor borrow from dataflow in order to make it more suitable for use in a multiprocessor?" We start with a simple, "RISC-like" instruction set, and show how to change the underlying processor organization to make it multi-threaded. We then 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 multiprocessors. We compare our approach to existing MIMD machines, and to other dataflow machines.

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

1 Introduction

The aim of this paper is to explore the question: "What can a von Neumann processor borrow from dataflow in order to make it more suitable for use in a multiprocessor?" For many years, it has been argued that dataflow architectures are better building blocks for scalable multiprocessors because they have mechanisms that can (a) tolerate increased latencies and (b) handle greater synchronization requirements [2]. Unfortunately, dataflow architectures have generally been so different from von Neumann architectures that it has been difficult to compare them objectively in order to substantiate or refute this claim. In [18], Papadopoulos designed the Monsoon dataflow processor. By using directly-addressed instead of associative wait-match memory, and frames for local storage, similarities with von Neumann machines began to emerge. In [13, 14], Iannucci explored a dataflow/von Neumann hybrid architecture in which the similarities were more striking.

In this paper, we take this evolution a step further. First, 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 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 [21, 19, 10]. Further, the instructions are simple and regular, suitable for pipelining. Nevertheless, our storage model is an unusual one.

[0]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 N00014-84-K-0099.

In Section 3, we change the underlying processor organization to make it multi-threaded (*a la* HEP), which, in itself, is an improvement for multiprocessors. In Section 4, we extend it with three instructions that give it a fine-grained, dataflow capability, making it an even better building block for multiprocessors. The resulting processor architecture, which we call P-RISC (for "Parallel RISC"), can exploit compiling technology both from conventional multiprocessors as well as from dataflow processors. In Section 5, we discuss memory support for multiprocessors.

This paper should not be read as an engineering proposal for a specific instruction set and/or processor organization— many serious engineering issues remain, which we discuss in the conclusion (Section 6). Rather, it is a preliminary exploration of a synthesis of dataflow and von Neumann architectures.

To set things in context: we are concerned here only with asynchronous, MIMD models. This covers most current and proposed parallel machines, such as the HEP [26], BBN Butterfly [22], Intel Hypercube [24, 15], IBM RP3 [5, 21, 20], Sequent [25], Encore [6], etc., and excludes machines like the Connection Machine [11], Warp [1], and VLIW machines [7].

2 The runtime model

2.1 Storage: trees of frames, and heaps

Consider a typical sequential implementation of the following program:

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

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

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

There is a *stack* of *frames* (activation records) which may go through the configurations shown in Figure 1 (our stack grows upward). At any given time, only the code for the topmost frame is active—lower frames are dormant.

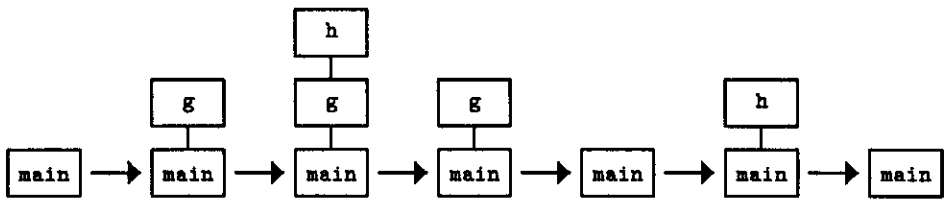


Figure 1: Stacks of frames in sequential implementations

In a parallel implementation, however, it is possible for `main` to call `g` and `h` concurrently, and `g` can call `h` concurrently as well. Thus, all frames can exist concurrently, so that we have to generalize our runtime structure to a *tree* of frames, as shown in Figure 2. Further, at any given time, the code for any of the frames can be active, not just at the leaves.

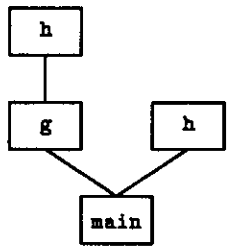


Figure 2: Tree of frames in parallel implementations

Another difference arises in loops. In a sequential implementation, a loop is typically implemented with a single frame. In a parallel implementation, however, we need many frames if we are to allow multiple iterations to run concurrently. Still, we have 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 most modern programming languages (e.g., Lisp, CLU, Id), it is possible for the lifetimes of data structures to differ from the the lifetimes of frames. Thus, data structures must be allocated on global *heap*.

A pair of frames can *share* values in two ways—either they can both refer to a common ancestor frame (by lexical scoping), or they can 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 a technique called "lambda-lifting" [16]. (We have a few more comments about this in the conclusion.)

To summarize: our runtime model of storage is this: 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 machine? We assume that a multiprocessor is 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.

At each instant, a Processing Element runs a *thread* of computation. A thread is completely described by an instruction pointer `IP`, and a frame pointer `FP`. The former points into the code in the PE, and the latter 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>` for continuations. Continuations correspond exactly to the "tag" part of a token in the terminology of tagged-token dataflow. It is usually convenient to have the

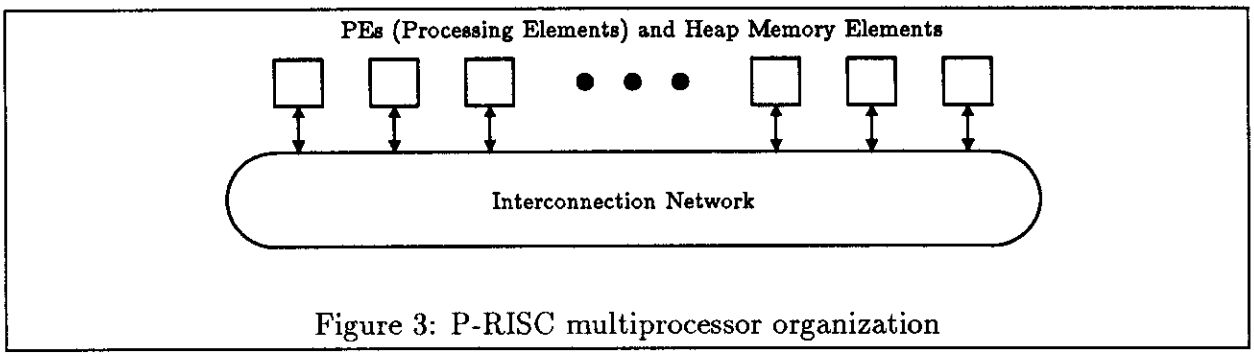


Figure 3: P-RISC multiprocessor organization

size of a continuation the same as the size of other values (such as integers and floating-point numbers), so that continuations themselves can be manipulated as values[1]

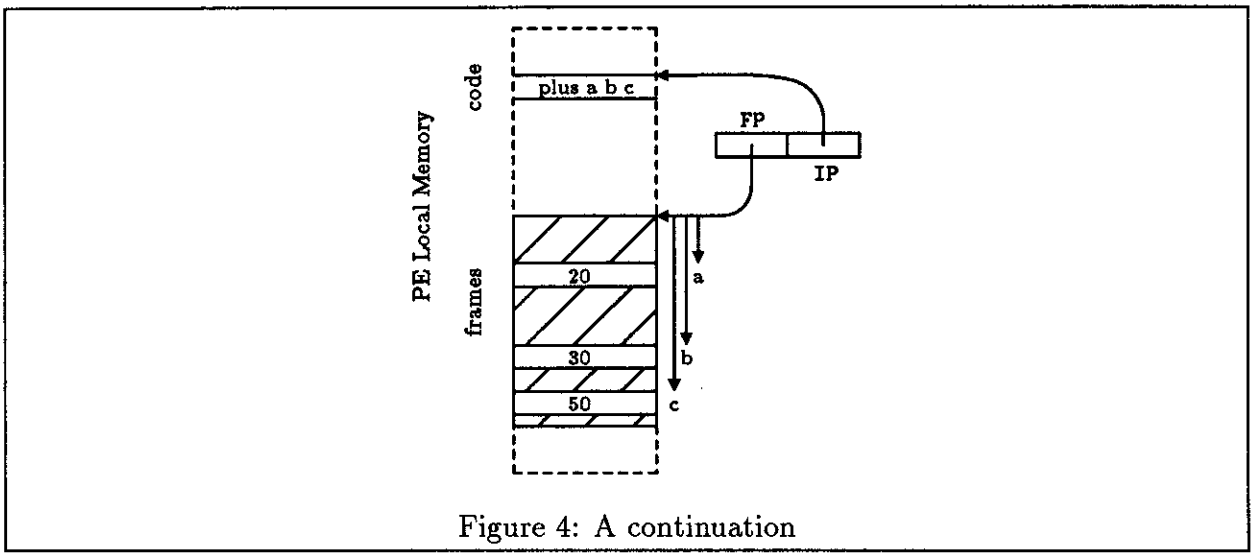


Figure 4: A continuation

As a running example, we will 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}}
```

The value of `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 of `s` in the final iteration. The example happens to be written in the language Id [17], but other parallel languages are equally acceptable.

Every instruction is executed with respect to a current `FP` and `IP`. All arithmetic/logic operations are 3-address frame-to-frame operations. For example, the instruction:

```
plus src1 src2 dest
```

[1]For example, in procedure calls, a "return continuation" is passed as an argument.

reads the contents of frame locations `FP+src1` and `FP+src2`, computes the sum, and stores the result in frame location `FP+dest`. The `compare s1 s2 d` instruction compares the contents of frame locations `FP+s1` and `FP+s2` and stores a condition code in the frame location `FP+d`. A *jcond* instruction:

```
jcond src newIP
```

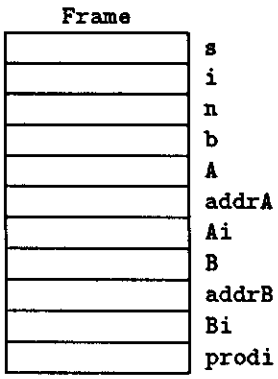
(for various conditions *cond*) treats the contents of frame location `FP+src` as a condition code, and changes `IP` to `IP+1` or `newIP` accordingly.

The `load a x` and `store x a` instructions move data between the 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. The `load` and `store` instructions are the only ones for moving data in and out of the PE, and do not involve any arithmetic. Thus, instruction-fetches and frame accesses involve local PE memory only, and no network traffic. Further, the arithmetic instructions should be simple and regular, so that they can be pipelined.

Here is a straightforward compilation of the body of `vip` into sequential code, with the frame shown on the right (for expository reasons, we have used more frame slots than are really necessary):

```
    load-immediate  0       s
LOOP:
    compare  i      n       b
    jgt      b      DONE
    plus     A      i       addrA
    load     addrA  Ai
    plus     B      i       addrB
    load     addrB  Bi
    mult     Ai     Bi      prodi
    plus     s      prodi   s
    incr     i      i
    jump     LOOP
DONE:   ...
```



2.3 Frames as register sets

In most RISC machines [21, 19, 10], the PE contains high-speed registers that allow two reads and a write on every cycle. Frames and the heap are not distinguished— both are on the other side of the interconnection network. Thus, 3-address instructions work on registers, and load/stores move data between registers and frames/heap. One of the registers is often interpreted as a frame pointer. To reduce network traffic, the size of the register set is increased, and the compiler tries its best to fit the entire frame (all local variables), into the register set.

Our model tries to formalize the view that a frame *is* the register set for a continuation. It is natural to ask whether this is realistic— can the register set be large enough to be regarded as local frame memory?

We assume that current technology allows thousands of registers in the PE (the HEP had over two thousand). Thus, even if all frames cannot be accommodated within the PE, it is certainly possible to compile code that provides that abstraction. Frames can be allocated in the heap, and the compiler and hardware can arrange to keep some current subset of them in the register set (a similar strategy is used in the Berkeley RISC [19]).

Another possibility is this. Let us assume that frames are of a fixed size, say 32 words (which, of course, can complicate life a little for the compiler). The PE contains an adequately large frame memory fronted by a cache for as many frames as is feasible. When an attempt is made to execute a continuation with an `FP` that names a missing frame, it's entire frame is loaded automatically, and the instruction is retried. Thus, loading a frame will involve a loss of many cycles. To avoid thrashing, the compiler/hardware would have to arrange it so that preference is given to continuations whose frames are currently in the cache.

With this much said, we ask the reader to bear with us, and we will return to this issue in Section 6.

2.4 Problems: memory latency and distributed memory

A major problem with our sequential code for `vip` is the latency of the two `loads`. Going out and back across the network to heap memory takes significant time relative to processor speeds, even on a uniprocessor; the problem is worse in a multiprocessor. It is alleviated somewhat by using a cache, but even this solution is less effective in multiprocessors. Thus, a processor may have to idle during a `load`, thereby reducing overall performance. In bus- or circuit-switched networks, long-latency `loads` can also interfere with each other, further degrading performance.

Ideally, we would like the following behavior for `loads`:

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

Many previous processor designs address this issue, to varying degrees. The Encore Multimax has split-phase bus transactions so that the bus is not occupied during a load, but a particular PE can have only one outstanding load— loads cannot be pipelined. The CDC 6600 [27], the Cray [23], and some RISC processors, can pipeline memory requests, but requests must come back in the same order. The 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. In general, all these solutions add significant complexity to the processor circuits.

A more detailed discussion of this issue may be found in [2].

An alternative is to go to a multi-threaded processor organization, similar to the HEP [26], and we follow this course in the next section.

3 Multi-threaded RISC: a dataflow implementation of the instruction set

In this section, we maintain the instruction set and its semantics, but change the underlying processor organization to support fine-grained interleaving of multiple threads.

In most high-performance machines (including RISC machines), the instruction pipeline is single-threaded, i.e., consecutive entities travelling through the pipe are consecutive instructions from the same thread, from addresses IP, IP+1, IP+2, and so on. Such an implementation introduces some extra complexity in the detection and resolution of inter-instruction hazards. Further, long latency instructions (such as loads) disrupt the pipeline and are an additional source of complexity.

The HEP [26] had a different pipeline implementation—it was time-multiplexed among several threads. On each clock, a different thread descriptor was inserted into the pipe. As thread descriptors 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 instruction, the thread 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.

In our multi-threaded RISC, we generalize the HEP's approach to allow an *arbitrary* number of threads to be interleaved. Recall that our thread descriptors, or *continuations*, are `<FP,IP>` pairs. The organization of the PE is shown in Figure 5. Since continuations are circulated in the processor, we also refer to them as *tokens*.

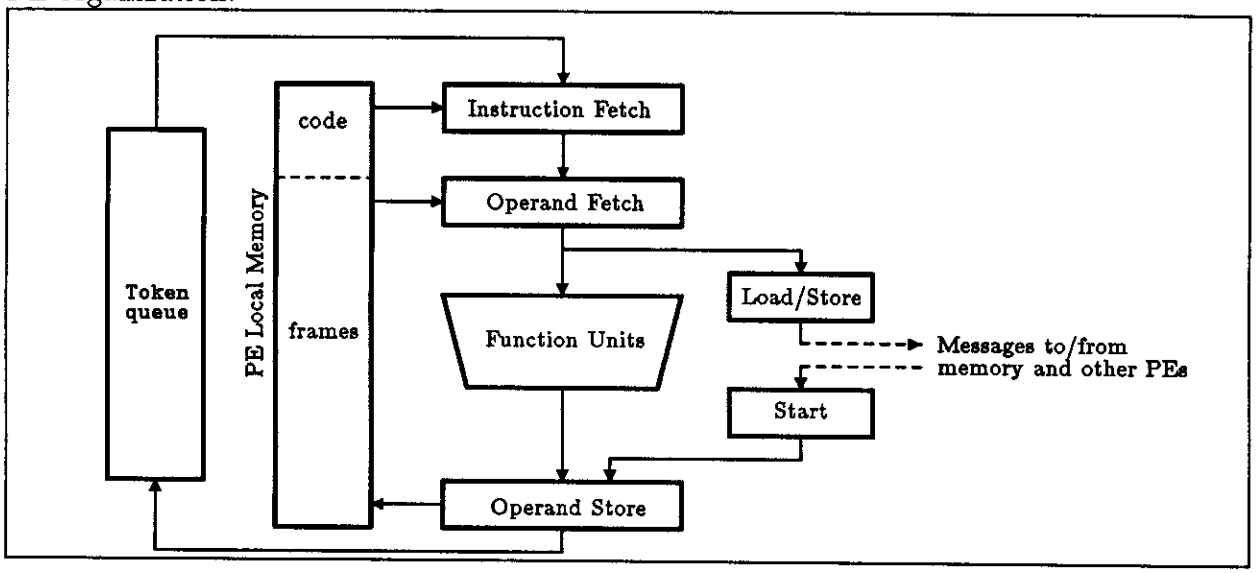
Tokens exist in the token queue (analogous to the HEP's PSW queue). On each clock, a token `<FP,IP>` is extracted from the token queue and sent through the pipeline, fetching and executing the instruction at `IP`, relative to the frame at `FP`. The pipeline consists of the traditional instruction fetch, 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 simply inserted into the token queue.

For Arithmetic/Logic instructions, the continuation is simply `<FP.IP+1>`. For the `jump` instruction, the continuation is simply `<FP.IP'>`. For *jcond* instructions, the continuation is either `<FP,IP+1>` or `<FP,IP'>`, 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 out into the network:

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

PE organization:



Instruction set summary:

Format	Frame operations	Continuations	Outgoing messages
Ordinary RISC-like instructions:			
`op a b c`	`[FP+a] op [FP+b] → FP+c`	`<FP.IP+1>`	—
`jump IP'`	—	`<FP.IP'>`	—
`jcond x IP'`	`[FP+x] →`	`<FP.IP'>` 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 IP'`	—	`<FP.IP+1>, <FP.IP'>`	—
`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]>`

Figure 5: P-RISC Processing Element (PE) organization and instruction set summary

Note: *no continuation* is inserted into the token queue! Meanwhile, the pipeline is free to process other tokens extracted from the token queue. Some of those, in turn, may be `load` instructions, thus 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,x>`

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

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

`<WRITE,`a`,`x`>`

into the network. A Heap Memory Element receives this, and writes 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
- 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.

Contrast our PE organization with that of the HEP [26]. In the HEP, too, each thread could issue a load. The main difference is that the HEP had a limited number of threads, and that was also a limit on the number of loads that could be outstanding.

The HEP had another limitation which is shared by our multi-threaded PE: even though there can be many outstanding loads from multiple threads, a *particular* thread can have no more than one outstanding load. We correct this situation in the next section.

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

In any multi-threaded system, there must be some way of initiating new threads, and to synchronize two threads. In most conventional architectures, these involve operating system calls, or traps, or pseudo-instructions or some such mechanisms. It is difficult to make such mechanisms very cheap, and so, one tends to avoid fine-grained parallelism, which, in turn, reduces the exploitable parallelism in programs.

We extend the multi-threaded RISC to P-RISC by adding two instructions to perform thread initiation and synchronization. It is important to realize that these are simple *instructions*— not operating system calls— that are executed entirely within the normal processor pipeline (again, please refer to Figure 5.):

- `Fork IP'` is just like a `jump` instruction, except that it produces *both* `<FP,IP'>` *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 the continuation `<FP,IP+1>`.

4.1 Inner-product revisited

Figure 6 shows, in outline, the change we need in the control flow of the inner-product program in order to do the two loads concurrently.

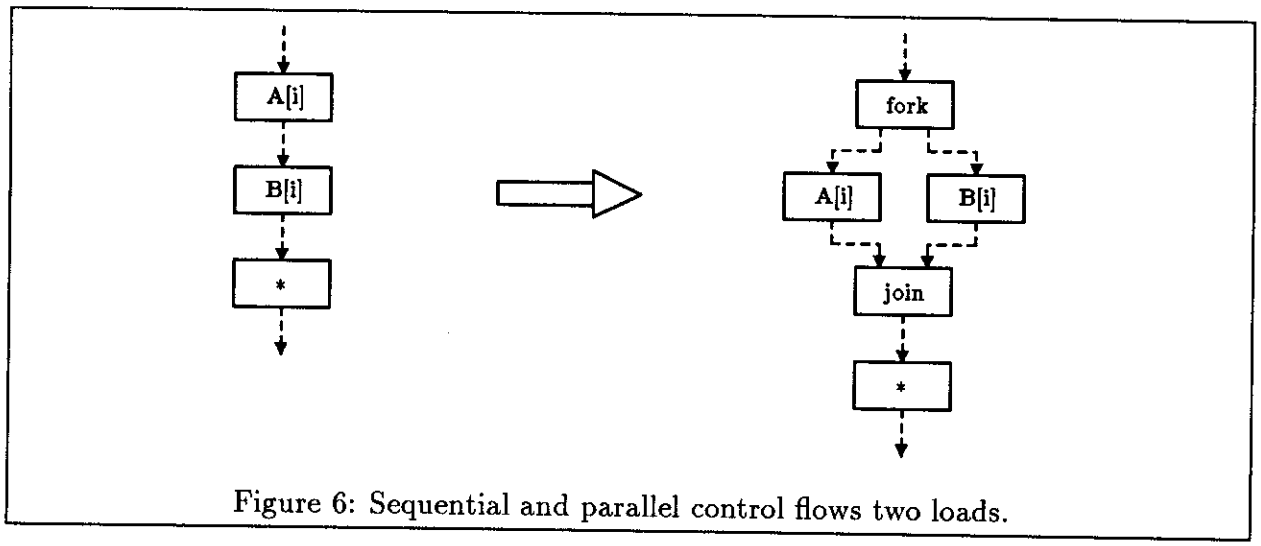
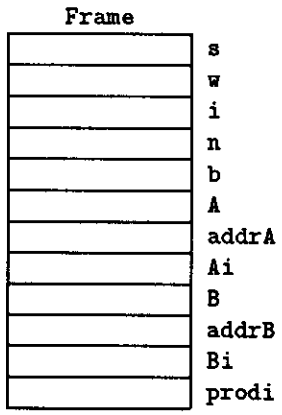


Figure 6: Sequential and parallel control flows two loads.

Here is our revised code for the inner-product procedure, where we have marked the new instructions with "*" labels:

```
    load-immediate  0  s
    load-immediate  0  w  (*)
LOOP:
    compare i  n  b
    jgt     b  DONE
    fork    LOADBi        (*)

LOADAi:                        |LOADBi:
    plus  A  i  addrA          |    plus  B  i  addrB
    load  addrA  Ai            |    load  addrB  Bi
                               |    jump  SYNCH        (*)
SYNCH:
    join  w               (*)
    mult  Ai  Bi  prodi
    plus  s   prod  s
    incr  i   i
    jump  LOOP
DONE:   ...
```



The frame, shown on the right, is identical to the previous one except that it has one additional location `w` used by the `join` instruction for synchronization. It is initialized to zero ("empty") in the second statement.

The `fork` instruction inserts tokens `<FP.LOADAi>` and `<FP.LOADBi>` into the token queue. The two resulting threads are shown above in two columns, for convenience. Both address calculations and loads can be executed concurrently.

When the `load` in the `LOADAi` sequence is executed, it sends the continuation `<FP.LOADAi+2>` (i.e., `<FP.SYNCH>`) in its message to heap memory. Thus, when the response arrives, the value is written into frame location `Ai` and `join w` is executed.

When the `LOADBi` sequence is executed, it sends the continuation `<FP.LOADBi+2>` in its message to heap memory. Thus, when the response arrives, the value is written into frame-location `Bi` and `jump SYNCH` is executed, causing `join w` to be 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 next instruction (the `mult`).

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 the "empty" state.

4.2 Simulating fine-grained dataflow

With such lightweight `fork` and `join` instructions, it is possible to think about simulating the fine-grained asynchronous parallelism of pure dataflow. For example, the left-hand side in Figure 7 shows a classical dataflow dyadic "+" instruction (at address IP0). It receives tokens t_l and t_r carrying values on its input arcs. When both tokens have arrived, it "fires", i.e., it consumes the tokens, adds the values, and produces tokens carrying the sum on each of its output arcs, destined for instructions at IP1 and IP2.

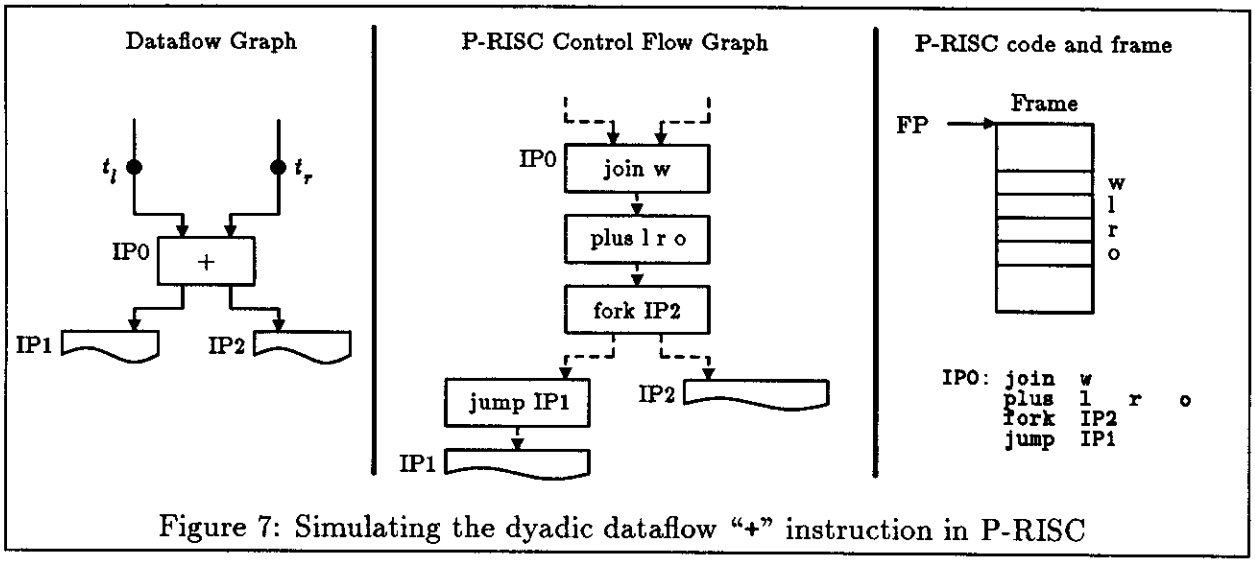


Figure 7: Simulating the dyadic dataflow "+" instruction in P-RISC

The P-RISC control graph that simulates the dataflow "+" instruction is shown in the middle of the figure, and the corresponding P-RISC code and frame are shown on the right. 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 that produced token t_l, there would be P-RISC code that stores the left input value in `l` and inserts a token `<FP.IP0>` in the token queue. Similarly, corresponding to the dataflow graph that produced token t_r, there would be P-RISC code that stores the right input value in `r` and inserts an identical token `<FP.IP0>` in the token queue. Thus, getting through the `join` instruction is a guarantee that both inputs are available. The effect of the last two instructions is simply to place the two tokens `<FP.IP1>` and `<FP.IP2>` in the token queue.

4.3 More parallelism

Here is a version of the inner-product program written using recursion instead of a loop:

```
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)) ;
```

In the `else` clause, not only the two loads, but also the recursive call can be done concurrently. Further, the multiplication can be done as soon as both loads have completed, even if the recursive call has not. This is summarized in the control flow graph in Figure 8.

Here is the code for `vip_aux`:

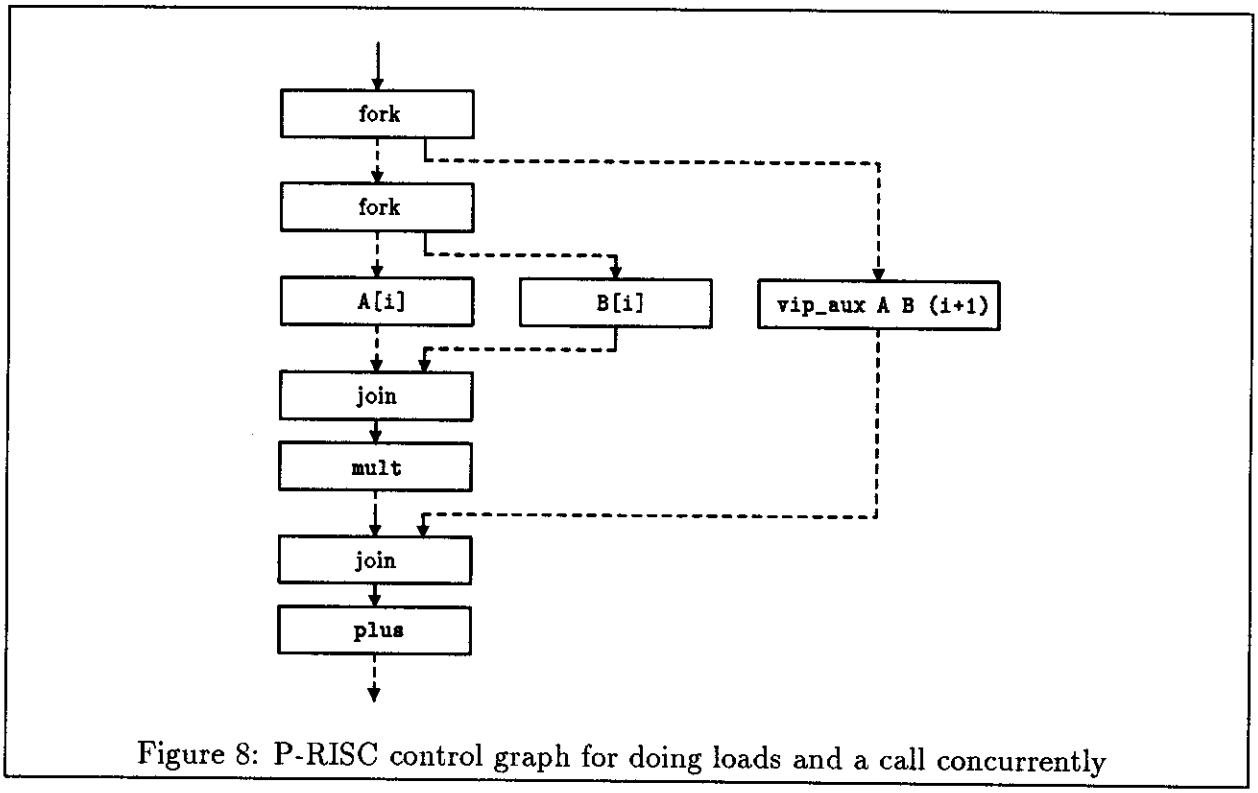
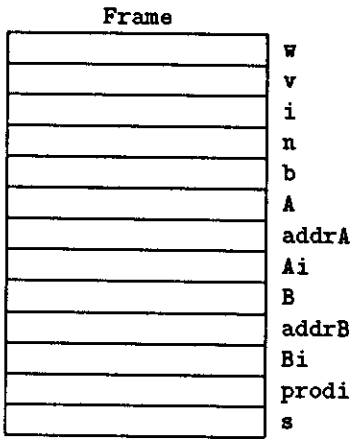


Figure 8: P-RISC control graph for doing loads and a call concurrently

```
VIP_AUX:
    load-immediate  0  w  ; reset to empty
    load-immediate  0  v  ; reset to empty
    compare  i  n  b
    jgt  b  TRIVIAL

    fork  REC-CALL
    fork  LOADBI
```




```
LOADAI:                 |LOADBI:                |REC_CALL:
    plus  A  i  addrA   |    plus B i addrB     |    ... call VIP_AUX A B (i+1) ...
    load  addrA  Ai     |    load  addrB  Bi    |    ... value is returned in s ...
                        |    jump SYNCH1        |    jump SYNCH2
SYNCH1:                                         |
    join w                                      |
    mult Ai Bi prodi                            |

SYNCH2:
    join v
    plus s prodi s
    jump DONE
```

```
TRIVIAL:
    load-immediate  0  s

DONE:   ...
```

In this version, the two `forks` result in three separate threads (shown, for convenience, in three columns)— one to load `A[i]`, one to load `B[i]`, and one to compute the value of the recursive call. The first two threads join at `SYNCH1`, and the multiplication is performed to compute the product term. This thread then joins with the third thread after it has returned from the recursive call, at `SYNCH2`. Finally, the addition is performed.

In this version of the inner-product computation, we can see that *all* the `loads` for the entire inner-product computation can be issued in parallel, the responses can come back in any order, and the multiplies are performed in arbitrary order, automatically scheduling themselves in the order in which the load responses arrive.

Of course, the additions in our inner-product program must still proceed sequentially. It would be just as easy to write a divide-and-conquer version where this was not so, but this program dramatically illustrates the following point: even in a program that appears sequential at the algorithmic level, there is much parallelism that can be exploited by our architecture— all the index calculations and the loads can be done in parallel.

4.4 Procedure-call/return linkage

When a procedure `g` calls a procedure `h`, we need to:

- allocate a frame `FPh` for `h`;
- communicate argument values from `g`'s frame `FPg` to `FPh`;
- initiate a thread of computation in `h`, using `FPh`;
- communicate result(s) from `FPh` to `FPg`;
- when `h` has completed (all its threads have stopped), deallocate the frame `FPh`.

In a sequential implementation, allocating and deallocating `FPh` just involves incrementing and decrementing a stack pointer. In a parallel implementation, there is a tree of frames, and so allocation and deallocation needs a more general storage management mechanism.

To communicate arguments and results, and to initiate a thread for the callee, we distinguish two cases: (1) both frames are on the same PE, and (2) the frames are on different PEs. The former could be handled by instructions that move data between local frames. However, we will discuss only the latter case, as it is more general.

When the two frames are on different PEs, there is a synchronization issue. Transferring arguments and initiating a thread both involve sending messages to the remote PE; however, the remote thread should not start until the argument has been written there. Thus, we tie these two together. We have already seen the `START` message which, when it arrives at a processor, writes a value and initiates a thread there (memory responses were `START` messages). Now, we allow such messages to be generated explicitly by an instruction:

```
start dv dfpip dd
```

This 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>

out of the PE, and no continuation is placed in the token queue. The caller executes a start instruction for each argument, writing a value into the callee's frame and initiating a thread there. Since each argument initiates a thread, the callee can begin processing each argument as it arrives, independent of the order of arrival. When synchronization/sequentialization is necessary, the callee executes join instructions. Similarly, to return a value, the callee executes a start instruction that writes a value into the caller's frame and initiates a thread in the caller.

4.5 Discussion

Note that every join instruction is fetched and executed twice. On the first attempt, 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 for a dyadic operator) [3, 8, 12].

Similar bubbles also occur in other pipelined machines. For example, in RISC machines, even though compilers try very hard to avoid it, the pipeline may have to stall when a required operand is not available (e.g., the register is busy). In the multi-threaded HEP, an instruction was automatically converted into a no-op and recirculated if it tried to access a busy register.

Our "continuations/threads" are really "flyweight" processes, and we interleave processes at the granularity of individual instructions.

5 Memory support for fine-grained multiprocessor parallelism

So far, we have seen that a Heap Memory Element in our model has the following behavior. 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 is not provide any synchronization— what happens if a READ message for a location arrives from PE0 before the corresponding WRITE message from PE1? To solve this problem, we extend the behavior of Heap Memory Elements in the

direction of the "I-Structures" of dataflow architectures [3]. 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 in the "full" state, it behaves like an ordinary `READ`. If it is in the "empty" state, the location contains a "deferred-list" (initially empty). Each element of the list 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", then, for each `(FP.IP,d)` in the deferred list, the memory sends out a message:

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

and, finally, the value v is written into the location `a`. Thus, `I-READ`s can safely arrive before the corresponding `I-WRITE`s. Of course, this assumes that the location is written only once; it may be possible to guarantee this from the language semantics and/or compiler analysis (for example, this 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 Future directions

To develop P-RISC, we went through the following steps:

- 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. We have invented an instruction set for this paper, but many variations are possible.
- Make it multi-threaded, using a token queue and by circulating `<FP.IP>` tokens (thread descriptors) through the processor pipeline and token queue. `Loads` are split into two phases— 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.
- Introduce synchronization in the Heap Memory Elements using "I-structure" semantics.

One of the attractive aspects of P-RISC is that it can leverage existing compiling technology, both conventional and dataflow. Being a superset of a regular RISC instruction set, existing compiling techniques for single- or multiprocessor FORTRAN should carry over directly. As alluded to in the section on simulating fine-grained dataflow, we also know how to compile a high-level language such as Id [17] for this machine [28, 29].

Thus, P-RISC appears to be an attractive synthesis of dataflow and von Neumann architectures. However, much work remains to be done to evaluate its potential.

6.1 Some implementation issues

Frames as register sets:

Perhaps the most important question about P-RISC that requires investigation is whether it is realistic to view frames as register sets. The HEP had over two thousand registers in the PE in order to support multiple, concurrent threads, and had a base pointer in each circulating PSW (Process Status Word) to offset into the space of registers; this corresponds exactly to our `FP`. Note that in P-RISC (and, to some extent, in the HEP), a small number of frames can support a much larger number of concurrent threads, because it can have fine-grained concurrency *within* a procedure activation or loop iteration.

In Section 2.3 we suggested a caching scheme if frame memory is too large to be treated as high-speed registers— frames are of fixed size; frame memory is fronted by cache of frames; a frame is automatically loaded into the cache when the PE attempts to execute a continuation that refers to a missing frame, and priority should be given to those tokens referring to currently cached frames. We need to study both the compiler and hardware engineering problems in implementing such a scheme.

Instruction scheduling:

As we described it, on each clock we pick up an `<FP.IP>` pair from the token queue and insert it into the processor pipeline. Similarly, on each clock, zero, one or two tokens are produced at the output of the pipeline and are inserted back into the token queue. An alternative would be the following: as long as a thread does not die (due to a `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 named high-speed registers. Also, it would improve locality for a cache-based frame memory, as discussed in Section 2.3.

Frame and heap synchronization:

In our model, synchronization occurs in two places— in frames during a `join` instruction, and in heap memory with I-structure operations. The former is more efficient because it needs no queueing— exactly two threads come together. In I-structure operations, an arbitrary number of threads can come together (a producer and any number of consumers). Thus, data structures that have many consumers are allocated in the heap. If it is known that a data structure has only one consumer, it may be allocated in a frame, and a variant of the `join` mechanism can be used to read it. The details are beyond the scope of this paper.

Lexically scoped variables may also have a variable number of consumers. There are various ways of dealing with this. One possibility is to transform the program using lambda-lifting [16], essentially converting a lexically scoped variable into an extra argument that is

passed down into nested procedures explicitly. Thus, we are left with a flat environment structure which is captured nicely by our FP.

An alternative is to keep a heirarchical environment structure, implying two things. First, accessing variables from superior frames must begin relative to the current frame's FP, and thus will involve more indirections. Second, since synchronization in frames does not have queuing, the compiler must ensure that such variables have their values before they are accessed, by inserting suitable control dependencies.

6.2 Comparison with other work

Buehrer and Ekanadham studied ways to incorporate split-phase memory transactions into von Neumann architectures [4]. Halstead and Fujita proposed an multi-threaded processor architecture [9].

Papadopoulos' Monsoon architecture [18] is a pure dataflow architecture in the sense that tokens are not only a means to schedule instructions, but also carry data. Inter-processor tokens are identical to intra-processor tokens. Since tokens carries 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.

The work that is closest to P-RISC is Iannucci's dataflow/von Neumann hybrid architecture [13, 14]. Every frame location had "presence bits" indicating whether it was full or empty. A `load` instruction simply sent a request to memory along with the address of the destination frame location, and execution continued at the next instruction. An operation that tried to read an empty frame location could trap, storing its process descriptor in that location and marking it "pending", and execution resumed at some other thread. When a response comes back 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 had split-phase loads, loads could be pipelined, responses could come in any order, and the namespace for waiting threads was the address space of memory.

The primary difference between Iannucci's machine and P-RISC is that Iannucci had presence bits on every location in frame memory, and synchronization could 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.

These comparisons with other work are by no means exhaustive— much more detailed design and experimentation is required to evaluate P-RISC. We hope this paper will stimulate research in this direction.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, and A. Webb. The warp computer: Architecture, implementation and performance. Technical Report CMU-RI-TR-87-18, Carnegie Mellon University, The Robotics Insititute, July 1987.

[2] 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, Springer-Verlag LNCS 295*, June 25-29 1987.

[3] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. Technical Report CSG Memo 271, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, June 1988. An earlier version appeared in *Proceedings of the PARLE Conference, Eindhoven, The Netherlands*, Springer-Verlag LNCS Volume 259, June 15-19, 1987.

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

[5] J. Edler, A. Gottlieb, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson. Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach. In *Proceedings of the 12th Annual International Symposium on Computer Architecture, Boston, June 1985*, pages 126–135, 1985.

[6] Encoré Multimax. Technical report, Encore Computer Corporation, 257 Cedar Hill Street, Marlborough, MA 01752.

[7] J. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockolm, Sweden*, June 1983.

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

[9] R. H. Halstead, Jr. and T. Fujita. Masa: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th. Annual International Symposium on Computer Architecture, Honolulu, Hawaii.* IEEE/ACM, June 1988.

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

[11] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

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

[13] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988. Ph.D. thesis.

[14] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the IEEE 15th. Annual International Symposium on Computer Architecture, Honolulu, Hawaii.* IEEE/ACM, June 1988.

[15] Intel iPSC. Technical report, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, Oregon 97006.

[16] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Springer-Verlag LNCS 201 (Proc. Functional Programming Languages and Computer Architecture, Nancy, France)*, September 1985.

[17] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[18] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, August 1988.

[19] D. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):9–21, January 1985.

[20] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 IEEE International Conference on Parallel Processing*, pages 764–771, August 1985.

[21] G. Radin. The 801 Minicomputer. In *Proceedings of the ACM Symposium on Architectural Support of Programming Languages and Operating Systems*, pages 39–47, March 1982.

[22] R. Rettberg, C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, W. Clark, and M. Kraley. Development of a Voice Funnel System. Technical Report Design Report 4098, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, August 1979.

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

[24] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[25] Sequent computers. Technical report, Sequent Computer Systems, Inc., 67 S. Bedford Street, Burlington, MA 01803.

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

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

[28] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[29] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988. Ph.D. thesis.

Contents

November 15, 1988