LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

*T: A Multithreaded Massively Parallel Architecture

Computation Structures Group Memo 325-1
November 15, 1991
(Also to be released as a DEC Cambridge Research Laboratory Technical Report)

Rishiyur S. Nikhil†
Gregory M. Papadopoulos‡
Arvind‡

† Digital Equipment Corporation, Cambridge Research Laboratory
‡ MIT Laboratory for Computer Science

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

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

*T: A Multithreaded Massively Parallel Architecture

Rishiyur S. Nikhil[1]
Gregory M. Papadopoulos[2]
Arvind[3]

November 14, 1991

Abstract

In this paper we address the following question: What should the architecture of each node be in general purpose, massively parallel architectures (MPAs)? We frame the question in concrete terms by describing two fundamental problems that must be performed well in any general purpose MPA. From this, we systematically develop the required general organization of an MPA node, and present some details of *T (pronounced *Start*), a concrete architecture designed to these requirements. We discuss a hand-compiled example, and we discuss compilation issues. We examine several other multithreaded architectures to see how they meet our requirements.

Keywords: Multithreaded architectures, dataflow, fine grain parallelism, latency and synchronization

---

[1] Digital Equipment Corporation, Cambridge Research Laboratory
One Kendall Square, Bldg 700, Cambridge, MA 02139, USA
Phone: 1 (617) 621 6639 Fax: 1 (617) 621 6650 Email: nikhil@crl.dec.com

[2] MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA
Phone: 1 (617) 253 2623 Fax: 1 (617) 253 6652 Email: greg@au-bon-pain.lcs.mit.edu

[3] MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA
Phone: 1 (617) 253 6090 Fax: 1 (617) 253 6652 Email: arvind@au-bon-pain.lcs.mit.edu

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

Contents

1 Introduction 1

2 Fundamental problems in an MPA 2

3 Solutions 4

3.1 Caches 4

3.2 Multithreading 4

3.3 Fine Grain Multithreading: forks and joins 5

3.4 Messages and message-handling 6

3.5 Processing incoming messages efficiently 7

3.6 Handling synchronized loads 10

4 The *T architecture 11

4.1 The Data Processor 12

4.2 The Synchronization Coprocessor 13

4.3 Address Hashing 16

5 An example: DAXPY 17

6 Other multithreaded architectures 22

6.1 The Denelcor HEP and the Tera Computer System 23

6.2 The Dataflow/von Neumann Hybrid Architecture 24

6.3 The MIT Monsoon Dataflow Machine 25

6.4 The J-Machine, with Message Driven Processors 25

6.5 ETL's EM-4 dataflow system with EMC-R processors 26

7 Compiling for *T 26

1 Introduction

There appears to be widespread consensus that general purpose supercomputers of the future will be Massively Parallel Architectures (MPAs) consisting of a number of nodes in a high speed, regular interconnection network. This view has been eloquently summarized in the slogan "attack of the killer micros" by Eugene Brooks of Lawrence Livermore Laboratories [8]. In this paper, we focus on the organization of each node in such an MPA.

In an MPA, inter-node communication latency is expected to be high and caching global locations is difficult. In Section 2 we present this issue in concrete terms by describing two specific latency-related problems that must be performed well by MPA nodes. In Section 3 we systematically develop a node organization that can address these problems. In Section 4 we present some details of *T, a new node architecture based on these principles. The name *T is pronounced *Start*, after the principal communication instruction in the machine. The name can also be read as an acronym for "multi- (*) Threaded", a principal feature of the machine. In Section 5 we discuss the coding of DAXPY on this machine. In Section 6, we examine several other multithreaded processors to see how well they meet the requirements of an MPA node. We conclude in Section 7 with some discussion of compilation issues.

*T is an evolutionary step bringing together many years of research and development in two previously separate areas: dataflow and von Neumann processors for parallel computers. From the dataflow side, general purpose dataflow architectures such as the TTDA [5], the Manchester machine [17], the SIGMA-1 [19], Monsoon [31], Epsilon [16] and the EM-4 [36] have been strong influences. From the von Neumann side, one of the earliest influences is the seminal Denelcor HEP [38]. In 1986, Buehrer and Ekanadham [9] incorporated some dataflow features in a von Neumann architecture, and the important Hybrid Dataflow/von Neumann architecture by Iannucci [22, 21] was a complete and detailed design, including a flexible compilation model for threads. Nikhil and Arvind's P-RISC model [29] further simplified the mechanism to synchronize two threads.

Other researchers have arrived at similar mechanisms starting from other vantage points (*e.g.*, [18, 25]). Closely related to our work is Dally's Message Driven Processor (MDP) [12]. While developed through different paths for different language models, *T shares a number of basic mechanisms with the MDP, notably the dispatch of message handlers directly from instruction pointers in messages packets. It is likely that implementations of *T will draw directly on lessons learned from the MDP.

Software considerations, *i.e.*, compiling, have played an equally important role in the design of *T. From the very beginning, the compilation of the high-level parallel programming language Id [27] has been central to our thinking [40]. Nikhil, in [28], and Culler *et al* in [11] proposed P-RISC-like compilation models extended to allow additional instruction sequencing constraints, making explicit the notion of a thread. Culler's model, the Threaded Abstract Machine (TAM) has been implemented by his group and has been in use for almost two years, providing a wealth of data on the nature of threads, locality, synchronization, *etc.* [37]. This experience has directly influenced our thinking on *T.

The last two sections of this paper (Sections 6 and 7) will make this heritage of *T technically more explicit.

2 Fundamental problems in an MPA

In this section, we systematically develop our basic requirements for general purpose Massively Parallel Architectures (MPAs). We start by presenting, in Figure 1, a high-level structure for MPAs that is widely accepted today. An MPA consists of a collection of nodes

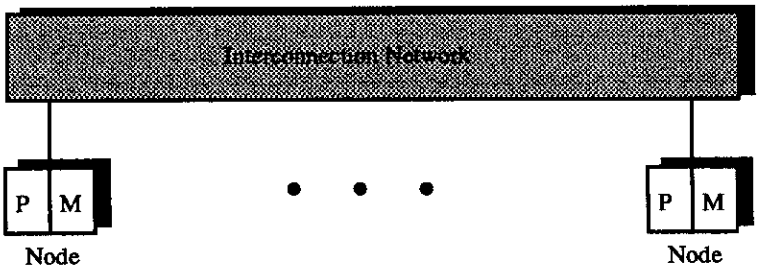


Figure 1: Structure of a massively parallel architecture.

that communicate using some form of interconnection network. Each node contains a processor and some local memory (the "dance-hall" configuration where all processors are on one side and all memories are on the other side is not considered feasible because of the volume of memory traffic and the latency penalty on every memory access). Given a node N, we refer to its own memory as *local memory* and to other memories as *remote memories*. It is also natural to place a cache in front of each memory to improve its apparent access time for requests from the associated processor or network connection (note that this is not global caching, where a cache in node N1 may contain a copy of a location from another node N2— that raises a cache-coherence problem, which we discuss briefly in Section 3.1).

Memory addressing: although we believe it is important, we do not consider here the issue of whether the memories in the different nodes belong to a single, global address space or whether the memory in each node has its own local address space. We also do not consider here the issue of virtual memory.

Locality and memory latencies: In an MPA, the latencies of local *vs.* remote memory accesses typically vary by orders of magnitude. Further, remote memory latencies grow with machine size— the best-case scenario is that the longest remote accesses will cost $\log N$ time, where N is the number of nodes. For some network topologies, it can be much worse.

Packet switching, or message based communication: It is also widely accepted that in order to pipeline the network in an MPA, communications will be packet switched, or message based. We do not address the issue here whether messages have fixed or variable length.

A consequence of message based communication is that remote accesses must be structured as *split transactions*. If node N1 wants to read the contents of location A in node N2, it involves two messages: a request message from N1 to N2 carrying the address A and a return address identifying N1, followed by a response message from N2 to N1 carrying the contents of location A.

We now present two problems which, we hope the reader will agree, a general purpose MPA must solve efficiently in order to have good performance over a wide variety of applications [6].

1. The Latency of Remote Loads

The remote load situation is illustrated in Figure 2. Variables A and B are located on nodes N2

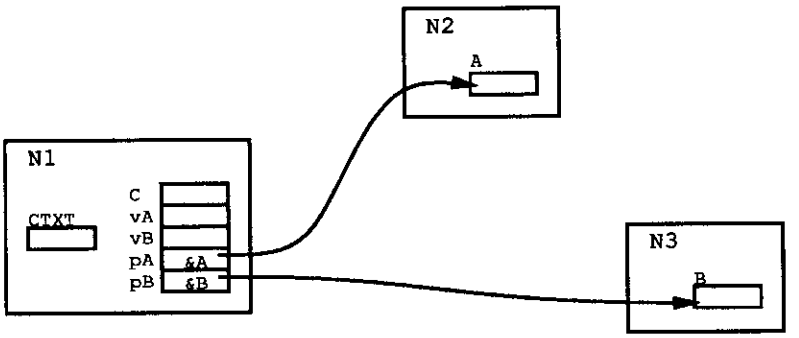


Figure 2: The remote load problem: Node N1 to compute difference of variables in nodes N2 and N3

and N3, respectively, and need to be brought to node N1 in order to compute the difference A-B in variable C. The basic intent of the computation is expressed by the following statement sequence which executes on N1:

```
vA = rload pA
vB = rload pB
C = vA - vB
```

where pA and pB are pointers to locations A and B, respectively. The first two statements perform remote loads, copying values from A and B into vA and vB, respectively. The last statement computes the difference and stores it in C.

In the figure we also show a variable CTXT, which is the *context* of the computation on N1. It could be, for example, a stack pointer, a frame pointer, a "current-object" pointer in an object-oriented system, or a process identifier, *etc.* In general, variable names like vA, vB and C are interpreted relative to CTXT.

The key issue in remote loads is how to deal with the long latencies in the communication system, *i.e.*, how to avoid idling the processor in N1 during a remote load operation. Note that in most architectures, these latencies are predictable because they depend mainly on the distance between the nodes.

2. Idling due to Synchronizing Loads

In the remote load problem, we now assume that the variables A and B are being computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to

read. We need some form of synchronization to ensure that the remote loads from N1 read the values only after they have been produced.

Unlike the remote load problem, the latency here is not just an architectural property— it is also application-specific because it depends on how long it takes to compute A and B. The time for A and B to become ready may be much longer than the actual transit time of messages. In particular, the unpredictable latencies of synchronizing loads are a problem *even if the loads are local.*

3 Solutions

3.1 Caches

The classic approach to masking memory latencies is to use caches. We could augment our MPA model of Figure 1 by adding a cache to each node to hold copies of remote locations.

A significant problem in an MPA is *cache coherence*, whereby the multiple copies of a location residing in different caches must be kept consistent. Even with caches, there is the question of how to avoid processor idling when there is a cache miss. A possible solution for the distributed cache coherence problem is to use *directories*, and for the cache miss problem is to multiplex between a small number of contexts to cover cache loading. Implementing these solutions appears non-trivial, and there are several research projects exploring their feasibility (the MIT Alewife [1], the Stanford DASH [42, 24] and the MIT MASA [18], for example). In any case, *the proposals in this paper may be seen as either an alternative or orthogonal approach to distributed cacheing.*

Note that while distributed cacheing can help in the remote load problem which has bounded latencies, they do not offer anything for the synchronizing load problem.

3.2 Multithreading

One way to avoid idling the processor during a remote load is to multiplex it amongst several threads. When one thread issues a remote load request, the processor begins work on another thread, and so on. Clearly, the cost of switching to another thread should be much smaller than the latency of the remote load, or else the processor might as well wait for the remote load's response.

Continuations on messages

A consequence of multithreading is that *messages should carry continuations*. Suppose, after issuing a remote load from thread T1, we switch to thread T2 which also issues a remote load. The responses may not arrive in the same order— for example, the requests may travel different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, *etc.* Thus, each remote load and response should carry an identifier for the

appropriate thread so that the right thread can be re-enabled on the arrival of a response. We refer to these thread identifiers as *continuations.*

Adequate continuation namespace

It should be clear that the longer latency of remote loads, the more threads we need to avoid idling the processor. Thus, it is important that we have a *continuation namespace* that is large enough to name an adequate number of threads waiting for remote responses.

The size of this namespace can affect the programming model seriously. If the latency of a single remote load is l, we would expect that at any given time, we need no more than about l active threads in the processor to cover remote loads. However, if the programmer (or compiler) cannot predict the scheduling of these threads precisely, he may need to provide *many more than* l threads in the hope that, dynamically, l of them are ready to run at any given time. Thus, if the continuation namespace is too small, it requires more precise scheduling of threads, which limits the programming model.

As we shall see shortly, if we generalize our remote transaction model to include not just remote loads, but (a) synchronizing remote loads and (b) remote parallel procedure calls, then it becomes extremely difficult for the programmer/compiler to schedule threads precisely. Therefore, it is essential to have a large continuation namespace.

3.3 Fine Grain Multithreading: forks and joins

So far, we have not said anything about the cost of creating threads or synchronizing them, just that once we have them, we should be able to multiplex amongst them efficiently. However, if thread creation and synchronization are sufficiently cheap, the multithreading idea can be advantageously taken to an even finer granularity.

Instead of the rload-rload-subtract sequence earlier, suppose we could fork separate threads for the two rloads and synchronize them when both remote loads have completed. Then, instead of taking four message-transit times (doing the two rload serially), we could perhaps do it in two (do them in parallel). We can express the idea as follows:

```
     fork M1
L1:  vA = rload pA     ; (A)
     jump N

M1:  vB = rload pB     ; (B)
     jump N

N:   join 2 J          ; join synchronization of responses
     C = vA - vB
```

`Fork` initiates a new thread at label `M1`. The parent thread continues at `L1`, issuing the remote load (A), which suspends to await the response. This allows the just-forked thread at `M1` to run, which issues the remote load (B), which also suspends to await the response. When the first response arrives (say, (A)), the first thread resumes, completing the instruction at `L1` by

storing the arriving value into vA, and jumps to N. Similarly, when the second response arrives (B), the second thread resumes, completing the instruction at M1 by storing the arriving value in vB and also jumping to N.

The join instruction at N is executed twice, once by each thread. Each time, it increments J (assumed initialized to 0) and compares it to the terminal count (2). If J is less than the terminal count, the thread dies, otherwise it continues. In general, if the terminal count is c, then $c-1$ threads arriving at N will die and the only the last one continues to perform the sequel.

Again, it is important to keep in mind that the cost of fork and join should be small compared to the latencies of the remote loads that we are trying to cover. Note that delayed loads, cache prefetching, *etc.* are also modest, "peephole" examples of using fork-join parallelism to overlap long-latency loads with useful computation, but they do not generalize well to remote loads in MPAs.

We refer to the behavior at N as a *join synchronization*. This is a very important action. A significant number of messages entering a node may be destined for join synchronizations. In an inner-product, at each index we remote load $A[j]$ and $B[j]$, join the responses, and multiply. Note that for all but the last thread entering a join, the amount of work done is very small— the thread increments a counter, tests it and dies. For a two-way join, this may be as small as testing and setting one bit. It is important that the processor should not take a large hiccup to do this.

Exactly the same kind of join synchronization is also needed for *software* reasons. For example, a function that sums the nodes of a binary tree may have exactly the same structure as the above program with the two rloads simply replaced by remote function calls to sum the left and right subtrees, respectively, and the subtraction in the final line replaced by addition. Now, the latencies include not only communication time but also the time to perform the subcomputations.

This leads to the following observation: for messages that arrive at highly unpredictable times (as in the tree sum) or at widely separate times (even if predictable) the rendezvous point J for join synchronization must typically be *in memory, not registers*. The reason: processor utilization will be low if registers remain allocated to a process for unpredictable and possibly very long durations.

3.4 Messages and message-handling

Our first approximation to message formats looks like this:

```
NA: < msg_type, arg1, arg2, ...>
```

Node address NA is used for routing in the network, and it may be consumed in transit or on arrival. The msg_type specifies how the message should be handled. The remaining arguments depend on the message type. Figure 3 shows the messages for the first rload in our example (A), where &A and *A mean the address and contents of variable A, respectively.

There are many possible *encodings* of the message fields. Node addresses may be derivable from CTXT and/or L1 (*e.g.*, in machines with global address spaces). CTXT may be a

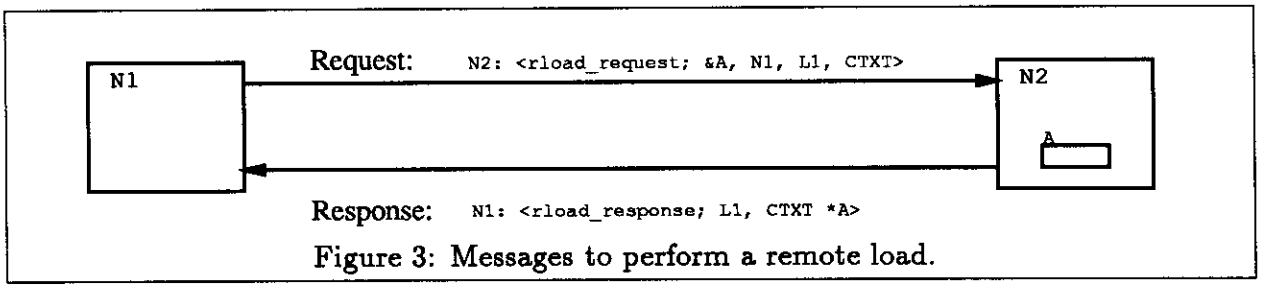


Figure 3: Messages to perform a remote load.

process id, an address of a process descriptor, a frame or stack pointer, the name of a register, a heap address, *etc.* L1 may be an actual program counter, an interrupt vector index, or may even be missing because it may be derivable from CTXT. Indeed, the machines surveyed in Section 6 use all these techniques.

When the rload request message arrives at node N2, it must handle it by reading location `A` and, using information that came in the request message, format and dispatch the response message to N1. An important issue is: when does N2 handle this message relative to the thread that may be executing when the message arrives, relative to other threads that may already be scheduled to execute on N2, and relative to messages that arrived earlier at N2 and have not yet been handled?

When the rload response message arrives at node N1, it must (a) complete the unfinished `rload` instruction at `L1` and (b) continue executing the thread. Again, there is the important issue of when this occurs relative to current activity in N1. It is possible to separate (a) and (b), *i.e.,* to complete the rload quickly, but to schedule the continuation just like other threads in N1. In this case, we need a separate *continuation queue* that will hold, for example, continuations like `<L1+1, CTXT>`.

3.5 Processing incoming messages efficiently

When a message such as a remote load request arrives at a node, it may be in the midst of an unrelated (and perhaps long) computation. For quick response, the arriving message could interrupt the processor, which responds and then resumes its normal computation. Unfortunately, interrupts are severely disruptive in modern processors, which achieve high speeds through deep pipelining and more processor state (registers, caches). Any change in context can incur a large penalty draining, saving, flushing, reloading, *etc.*In particular, frequent short threads, such as an rload handler or a message that goes into a failing join, are likely to be severely disruptive.

We observe that a very fast way to vector the processor to the right handler for an incoming message is for the the "message type" field (`rload_request`, `rload_response`) to directly *be* the program counter of the appropriate handler. Therefore, we refine our message format to be:

```
NA: < IP, arg1, arg2, ...>
```

i.e., our previous `msg_type` field is now interpreted as an *Instruction pointer* IP that points directly at the code that handles this message.

A well known technique for "lightweight" interrupts is to provide separate state in the processor for the interrupt handler (*e.g.*, a copy of the registers) so that the main thread's processor state does not have to be saved and restored during an interrupt. However, the performance of the main thread is still affected adversely, because each incoming message takes away cycles from the main thread and may require the processor pipeline to drain and refill.

In an MPA node, there is good reason to insist that we do not compromise on *excellent single-thread performance* (competitive with state-of-the art RISC uniprocessors). First, there are important SIMD/SPMD applications that do not need such efficient message handling, *i.e.*, they have long sequential threads interspersed with infrequent and/or regular communications. Second, even in more dynamic applications, each processor is likely to have certain sequential critical sections (such as storage allocators) whose latencies must be kept as short as possible. Third, users may wish to port their applications from uniprocessor versions of an MPA. Fourth, various libraries developed for uniprocessors should be portable to the processors of an MPA.

Hardware Support for Message Queues

In fact, given that a message may arrive on every clock in modern, pipelined MPA networks, it may not even be feasible to consume messages fast enough using interrupts. A solution is to provide a hardware *network interface* for accepting and queuing incoming messages, to be consumed by the processor at its convenience, as shown in Figure 4. The message queue

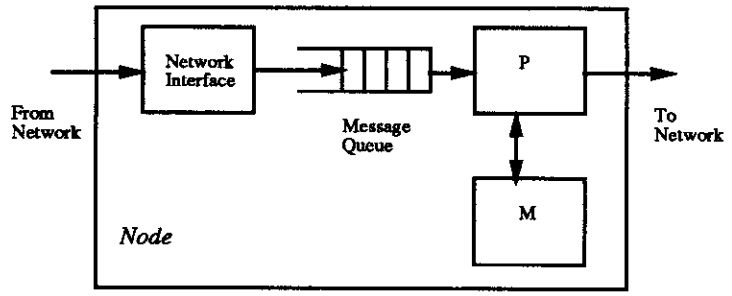


Figure 4: Network interface and incoming message queue.

may be implemented in separate memory or in the node's main memory. However, note that the time spent waiting in message queues must now be added to the overall latency of a remote load. If the processor is busy executing a long thread, this additional latency could be substantial.

A separate Synchronization Coprocessor

The key issue is efficient handling of the many messages that arrive for short, simple threads: rload request handlers, response handlers that just store a value and join, *etc.* We can offload this burden onto an entirely separate *synchronization coprocessor* sP that is optimized for

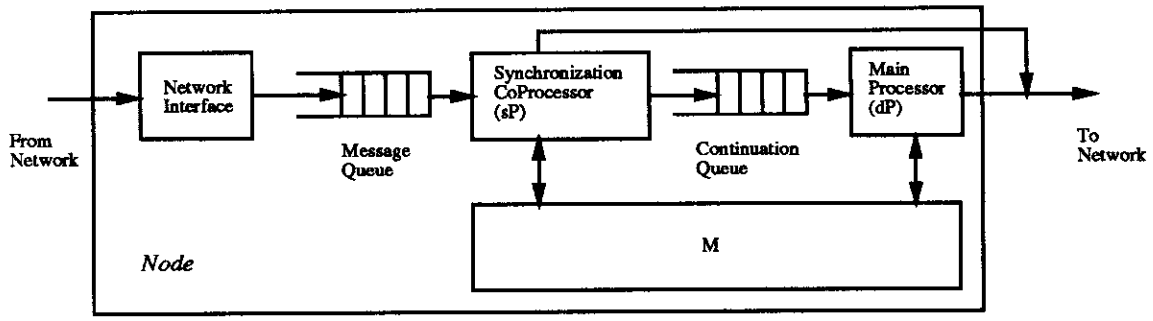


Figure 5: A separate synchronization coprocessor for short threads.

short threads, as shown in Figure 5. We will refer to the remaining, conventional main processor as the *data processor* (DP). For very fast message handling, one can imagine that the contents of a message are loaded directly into the registers of SP, with the first message field (IP) loaded into its PC, so that it is vectored directly to the appropriate handler. The basic action of the SP is to repeatedly load a message from the message queue and run it to completion; the run is always non-suspensive and is expected to be very short.

For an incoming rload request, it can be handled entirely within SP, without disturbing the main processor DP at all.

For an incoming rload response, the synchronization coprocessor can complete the unfinished rload, storing the value from the message into the destination location. For this, it needs access to the node's memory M, as shown in Figure 5. After this, it could place the <L1+1,CTXT> continuation into the *continuation queue*, to be picked up later by the main processor DP. Even better, if the immediate successor is a join instruction, the synchronization coprocessor can execute it, and place the continuation in the continuation queue only if the join succeeds. In this manner, the main processor DP does not even have to see join instructions.

Thus, the overall picture of a node is of *two asynchronous coprocessors*. The synchronization coprocessor executes short threads in response to messages, handling remote memory requests and join synchronizations. The main data processor executes long, sequential threads that are fed to it by the synchronization coprocessor. If the main processor is busy executing a long sequential thread, the continuation queue eventually fills up, blocking the synchronization coprocessor. However, messages may still be arriving in the message queue. For this reason, it is necessary to have a large message queue.

Clearly, with a larger continuation queue, more asynchrony is possible between the two coprocessors— the synchronization coprocessor can consume and dispose of more messages before blocking on a full continuation queue. This also results in faster turnaround (lower latency) for incoming rload requests.

There is an important special case where the continuation queue has size zero, where there is *no* asynchrony between the coprocessors. The hardware can be simpler because the two coprocessors can share a single datapath and cache to local memory, and local cache coherence is not an issue. This is very similar to the "lightweight interrupts" model discussed

earlier, but is actually a little better because the separate message and main processors can be optimized separately for short and long threads, respectively.

3.6 Handling synchronized loads

We have already observed we need some form of synchronization so that the remote loads read A and B only after their respective producers have completed. Towards this issue, some architectures add *presence bits* on each memory location indicating whether it is FULL or EMPTY. We will follow this idea, although our development does not depend on it—exactly the same arguments can be made if we simply interpret ordinary memory locations as semaphores (indeed, presence bits are a kind of semaphore). The variables A and B are initially marked EMPTY. The producer for each location, when writing a value there, also marks it FULL. Thus, the remote loads should succeed only after the locations become FULL.

The busy waiting solution

One solution is simply to busy wait, *i.e.*, if N1's remote load request for A arrives at N2 while it is still EMPTY, N2 responds immediately with a status indicating this. N1 simply retries the remote load later. Unfortunately, this can waste network bandwidth, and may also waste cycles on N1 in order to test and perform the retries. A minor variation: N2 could simply return the contents of A, presence bits and all, and leave it up to N1 to recognize that it is empty.

A data driven solution

A better solution is this. To execute a remote load that must synchronize at the remote location, node N1 executes, for example, an `iload` instruction instead of `rload`:

```
L1: vA = iload pA
```

The only difference in the request message is a different IP:

```
N2: <iload_request, &A, N1,L1,CTXT>
```

At N2, `iload_request` handler tests the presence bits of A. If FULL, it responds just like the `rload_request` handler. If EMPTY, it attaches the arguments (N1,L1,CTXT) to location A, marks it PENDING, and *produces no response.* When the producer later stores the value at A, N2 again checks the presence bits. If EMPTY, it simply stores the value there and marks it FULL. If PENDING, it not only stores the value there, but also constructs and sends the response to the waiting load. N2's response message for an `iload` looks identical to an `rload` response.

In general, there may be multiple loads pending at A, so a list or queue of pending requests needs to be constructed at A. Thus, one of the costs of handling iloads in this manner is that N2 must perform some storage management to construct these queues. However, a very simple free-list strategy may be used, since queue components are of fixed size and are not

shared. Thus, it does not matter if the remote load request arrives before the value is ready— it simply waits there until it is ready. There is no additional message traffic for the synchronizing remote load, and N1 does no busy waiting at all. As far as N1 is concerned, an iload is just an rload that may take a little longer— the interface is identical.

This data driven solution corresponds to *I-structures* in the dataflow literature, but we note that the suggested implementation is more general; a variety of other synchronization primitives (such as Fetch-and-ϕ) could be implemented using the same methodology. Finally, note that these message handlers are still simple enough to be handled entirely by the synchronization coprocessor, without disturbing the main processor DP.

4 The *T architecture

In this section we describe the *T architecture which is based on the general principles described so far. We provide just enough detail to establish its feasibility today. Figure 6 is a refinement of Figure 5 to show some of the registers in the two coprocessors, and a message formatter. Although not shown in the figure, it is likely that the coprocessors

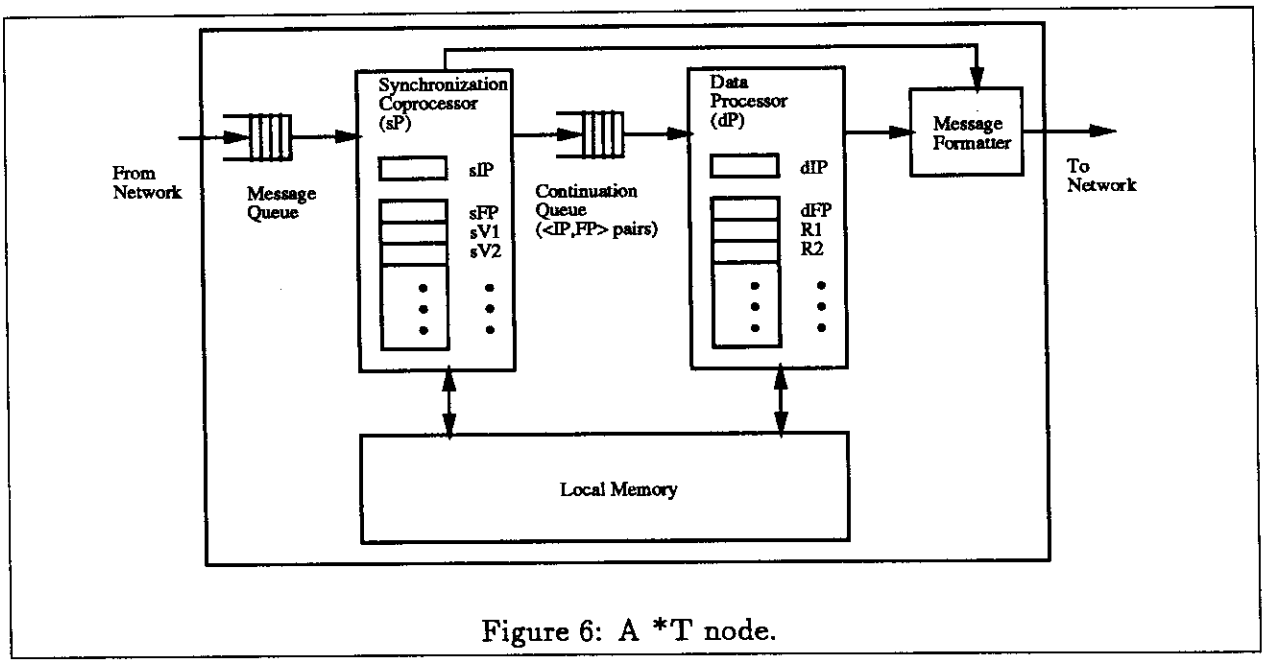


Figure 6: A *T node.

access local memory through on-chip and second level caches with some local cache coherence mechanism. In this section, we will describe the SP and DP coprocessors in terms of simple architectural extensions to otherwise conventional RISC processors.

Both the Synchronization Coprocessor and the Data Processor execute instructions, and each can refer to code labels for the other. For clarity, we will distinguish labels for the two coprocessors by the subscripts S and D, respectively (*e.g.*, $\texttt{L}_S$, $\texttt{L}_D$, $\texttt{M}_S$, $\texttt{M}_D$, ...).

We will look at messages in more detail shortly, but for the moment it is enough to know that each message has the form:

<IP, Address, Value1, Value2, ...>

Address is a global address that identifies a unique destination node in the MPA, and is typically a frame pointer FP or an address to a location in the heap. The message is automatically routed there. Of course, messages from a node to itself are short-circuited back directly.

We assume that a *context* for a computation is represented simply by an address in memory. In the spaghetti stack model, this may be a stack pointer or a pointer to a frame in the stack. In an object oriented model, this may be a pointer to an object. For uniformity, we will use the term *Frame Pointer* FP to refer to a context.

We assume that frames or contexts do not span nodes in the MPA, *i.e., each context resides entirely within one node* of the MPA. Thus, FP may be regarded as a pointer to the collection of local variables of that context. A frame pointer encodes a node number. Thus, a continuation is given by <IP, FP> where IP is a pointer to an instruction on the same node encoded by FP. This model of locality is not restrictive in any way— parallel procedure calls and loop iterations may be distributed to different nodes by giving them separate contexts.

It may be desirable to allow contexts to migrate across MPA nodes for load balancing and locality reasons. However, that capability is orthogonal to our discussion here and, for simplicity, we assume that frames, once allocated, do not migrate (a virtual-to-physical FP mapping, plus forwarding, may be added in the manner of the J-Machine [12] or Empire [22]).

4.1 The Data Processor

The Data Processor DP is a superset of a conventional RISC processor, with a conventional repertoire of register-to-register instructions and ability to manipulate local memory using conventional load and store instructions. Its program counter is called dIP ("Data processor Instruction Pointer"). One of its registers, called dFP, is assumed, by convention, to contain the current Frame Pointer for the thread being executed by the DP. Being a conventional RISC processor, the Data Processor is optimized to run sequential threads efficiently. It obtains the starting points of these threads from the continuation queue using a **next** instruction. Each such thread is run to completion, *i.e.,* there is no concept of a thread "suspending" in the Data Processor. On completion of a thread, if there is no new thread available from the continuation queue, the Data Processor simply waits until one is available.

In addition to its conventional RISC instructions, the Data Processor can execute a few additional instructions which are summarized in Figure 7. Most of them send messages into the network. These are non-blocking sends, *i.e.,* the Data Processor continues executing after sending a message. The message can cause threads to be scheduled on the other nodes or on the same node, and a later response may deposit values in the sender's frame. Thus, sending a message is also an implicit fork.

The new instructions are readily implemented as memory-mapped operations, so that it is possible that DP is an *unmodified* RISC microprocessor. To integrate the **start** instruction into a two-read/one-write style of RISC instruction, it is convenient to treat rIP and rFP

DP instruction	Semantics
`start rIP, rFP, rV`	Send message: <L_S, FP, V>
`next`	Load (L_D,FP') from continuation in continuation queue into registers (dIP,dFP) (so, execution continues at L_D)
`rload rIP, rA`	Send message: <rload_handler$_S$, A, L_S, FP>
`rstore rA, rV, rIP`	Send message: <rstore_handler$_S$, A, V, L_S, FP>

where: L_S = Register[rIP], FP = Register[rFP], V = Register[rV], A = Register[rA]

Figure 7: Data Processor instructions (beyond conventional RISC set)

as the double register rC which in essence holds the continuation for the remote thread. Another possibility is that messages with more than one value can be incrementally defined by a sequence of instructions similar to the technique employed in Dally's Message Driven Processor (MDP) [12].

The start instruction starts a new thread in a different context. For example, suppose function F calls another function G. The frames for these functions may be on different nodes (hence, different contexts). Thus, start instructions are used by F to send arguments and initiate threads in G, and start instructions are used by G to to send results and resume threads in F. However, since start instructions are non-blocking, it is possible to perform parallel function calls and coroutines. Note that instruction pointer L_S on the start message is a label for the target node's Synchronization Coprocessor, not Data Processor. (Readers familiar with dataflow literature will recognize that the contents of a start message correspond exactly to a dynamic dataflow "token"— FP is the context or color, L_S is the instruction pointer or statement number and, of course, V is the value on the token.)

The next instruction terminates the present DP thread by loading a new continuation, after which the DP is executing a new thread. Note that there is no implicit saving or restoring of the other registers. In general, at the beginning of a new thread, no assumptions can be made about the contents of any DP registers except dFP. We also say that registers are "volatile" or "ephemeral" across threads. This instruction is similar in effect to Monsoon's STOP instruction and the MDP's SUSPEND.

We will discuss the rload and rstore instructions shortly, after describing the Synchronization Coprocessor.

4.2 The Synchronization Coprocessor

The instruction set architecture of the Synchronization Coprocessor also looks very much like a conventional RISC microprocessor— it has a program counter sIP, general purpose

registers, and it can load and store to local memory. Unlike a conventional RISC microprocessor, it is tuned to very rapid disposition of incoming messages instead of the computation of arithmetic expressions. In particular, some of its registers (`sIP`, `sFP`, `sV1`, `sV2`, ...) can be loaded directly from messages, there is datapath support for `join` operations, it can post <FP,L_D> pairs into the continuation queue, and it can send messages.

Like a dataflow processor, the Synchronization Coprocessor is triggered by the arrival of a network message (it simply waits, if there is none available). When it picks up a message, its `sIP`, `sFP` and `sV1`, `sV2`, ... registers are loaded with the corresponding values from the message, after which it begins executing instructions from the address in `sIP`. The unconventional instructions of `sP` are summarized in Figure 8.

sP instruction	Semantics
`start rIP', rFP, rV`	Send message: <L_S, FP', V>
`next`	Load (L_S,A,V1,V2, ...) from message in message queue into into registers (sIP,sFP,sV1,sV2,...) (so, execution continues at L_S)
`post rIP, rFP`	Post <FP', L_D> into continuation queue

`join` *ctr, tc,* `rIP`	`Memory[FP +` *ctr*`] := Memory[FP +` *ctr*`] + 1`
	If `Memory[FP +` *ctr*`]` $\geq$ *tc* then
	Post <FP, L_D> into continuation queue
	`Memory[FP +` *ctr*`] := 0`

where: L_S = `Register[rIP']`, L_D = `Register[rIP]`, `FP'` = `Register[rFP]`, `FP` = `Register[sFP]`
`V` = `Register[rV]`, `A` = `Register[rA]`

Figure 8: Synchronization Coprocessor instructions (beyond conventional RISC set)

The `start` instruction is identical to the Data Processor's `start` instruction— it starts a new thread in a different context. This allows it to respond to incoming messages directly. For example, in an `rload` request message, the *IP* field points to `rload_handler`$_S$, the *address* field points at the address to be read, and the *V1* and *V2* fields contain the return continuation (return IP, return FP). The handler code looks like this:

```
rload_handler_S:
     load   rX, sFP[0]      ;  sFP  contains the address to read A
     start  sV1, sV2, rX    ;  create response message
     next                   ;  done; handle next message
```

Similarly, here is the handler for remote stores (the message's *V1*, *V2* and *V3* fields contain the value to be stored and the return continuation, respectively):

```
rstore_handler_S:
     store  sFP[0], sV1     ;  sFP  contains the address to write A
```

```
start  sV2, sV3, 0    ;  create acknowledgment message
next                  ;  done; handle next message
```

The acknowledgement may be used to ensure serial consistency— the destination thread executes under a guarantee that the store has completed— or to return miscellaneous status information. Omitting the return continuation from the message and the **start** instruction from the handler implements unacknowledged stores.

The **next** instruction ends the present SP thread by reloading its registers from the next message, after which it is executing that message's handler.

The **post** instruction allows the Synchronization Coprocessor to post a thread to DP by inserting a continuation into the continuation queue. Recall, this thread will be executed by the DP when it is popped off the continuation queue by the DP issuing a **next** instruction. Here is a typical SP code sequence that executes for an incoming message that loads label L_S into sIP (for example, this may be the response to a remote load):

```
L_S:
     store  sFP[T], sV1     ;  store incoming value into frame offset T
     post   L_D, sFP        ;  enable thread L_D  with this frame in Data Processor
     next                   ;  done; handle next message
```

The **join** instruction allows fast JOIN synchronization using synchronization counters in the frames. Because this is a very frequent operation for the SP, we provide special hardware support to accelerate JOIN operations. The **join** instruction implements an atomic test-and-increment semaphore on a location in the current activation frame called a *join counter*. The **join** instruction *conditionally posts* a thread to the continuation queue only if incrementing the join counter causes it to reach the terminal count (given by *tc*). It is assumed that a newly allocated activation frame will have all of its join counter locations initialized to zero. Observe, the **join** instruction implements a self-cleaning protocol by returning the counter to zero after the terminal count has been reached.

For example, suppose two messages arrive at node N1 (such as two responses to remote loads):

```
L_S, FPx, V1
R_S, FPx, Vr
```

On arrival of each message, the corresponding values are stored in the frame at offsets T1 and Tr, respectively. Then, a join counter at offset jC in the frame is incremented and compared with the constant 2. The two messages may be processed in any order; the first message will find the counter equal to 1, and SP will go to process the next message. The second message will find the counter equal to 2 and SP will post <FP, L_D> to the Data Processor. Here is the code for both the SP and the DP:

```
;;
;;  Synchronization Coprocessor message handlers
;;
L_S:
     store sFP[T1], sV1     ;  store incoming value V1  into FPx[T1]
     join  jC, 2, L_D       ;  attempt join, post thread if counter reaches 2
     next                   ;  handle next message
```

```
R_S:
    store sFP[Tr], sV1      ;  store incoming value Vr  into FPx[Tr]
    join  jC, 2, L_D        ;  attempt join, post thread if counter reaches 2
    next                    ;  handle next message


;;
;;  Data Processor code to handle thread
;;
L_D:
    load  rV1, dFP[T1]      ;  fetch first operand from frame
    load  rVr, dFP[Tr]      ;  fetch second operand from frame

     body of computation thread ...

    next                    ;  goto next thread
```

The Synchronization Coprocessor message handlers correspond almost exactly correspond to *inlets* in the TAM model [11]. This style of message handling is also easy to code for the MDP, although there is no dedicated logic for fast n-ary join synchronization. The salient difference is that in *T the message handlers (inlets) and the computation threads are processed by two logically distinct and asynchronous processors.

Turning back to the Data Processor, it can initiate remote loads and stores using `rload` and `rstore` instructions, each of which sends a message. The destination node is implicit in the global address `A`, which is used to route the message there. We have already shown the remote `rload` handler code. Observe, the response contains `FP` in its *address* field, so it is routed back to the node that issued the `rload`. The `rload` and `rstore` instructions are also forks— they send a message and continue at the next instruction. We can initiate many remote accesses before receiving any response, and the responses may return in any order.

An important point is that the `rload` instruction copies a remote value into a local *frame* location, and not a register. Although this means that an additional (local) load instruction is necessary to move it from the frame to a register, the choice is deliberate: it allows the thread that issued the `rload` to easily relinquish the Data Processor before the result comes back. Our choice recognizes the fact that storing the value in the frame would be necessary anyway if `rload` returned to a failing join or if it was a synchronizing load.

We note that an implementation might want to hardcode the `rload` and `rstore` message handlers in the synchronization coprocessor SP with specialized hardware in order to improve their performance and offload the SP. However, a nice aspect of their implementation as SP message handlers is that they are readily extended to perform a variety of synchronization operations such as synchronizing loads and stores (`iload` and `istore`), M-structures and Fetch-and-ϕ.

4.3 Address Hashing

In an MPA, it is useful simultaneously to have different kinds of address maps for different kinds of objects. For example, we may wish to interleave large data structures such as vectors

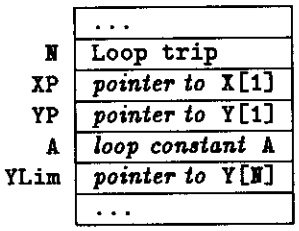
and matrices across the nodes of the MPA. On the other hand, we may wish code segments, stack frames and small objects to be addressed linearly within a node's local memory. Of course, these variations in address hashing could be performed in software, but this will incur a large overhead. In *T we intend to provide hardware support for address randomization in a manner similar to the IBM RP3 [34] or the Tera [2]. A logical place for this mapping to take place is in the Message Formatter.

5 An example: DAXPY

DAXPY is the inner loop of the Linpack benchmark:

```
for i = 1 to N do
    Y[i] = a * X[i] + Y[i]
```

We assume that the current frame (context) contains the following data, with symbolic names for the frame slots shown at left:



Uniprocessor code

Uniprocessor *T code for the DAXPY loop is shown below (in a uniprocessor, there is only "local" memory, which contains arrays X and Y). We use names beginning with "r" as symbolic names for general purpose registers. For clarity, the loop has not been unrolled. A typical compiler optimization would be to unroll the loop four or eight times.

```
        load  rXP, dFP[XP]          -- load ptr to X
        load  rYP, dFP[YP]          -- load ptr to Y
        load  rA,  dFP[A]           -- load loop constant: a
        load  rYlim, dFP[YLim]      -- load loop constant: Y pointer limit
        cmp   rB,rYP,rYLim          -- compare ptr to Y with limit
        jgt   rB, OUT               -- zero-trip loop if greater

LOOP:

        load  rXI, rXP              -- load X[i] into rXI          (L1)
        load  rYI, rYP              -- load Y[i] into rYI          (L2)
        add   rXP,rXP,8             -- increment ptr to X
        fmul  rTmp,rA,rXI           -- a*X[i]
        fadd  rTmp,rTmp,rYI         -- a*X[i] + Y[i]
        store rYP, rTmp             -- store into Y[i]                    (S1)
        add   rYP,rYP,8             -- increment ptr to Y
        cmp   rB,rYP,rYLim          -- compare ptr to Y with limit
        jle   rB, LOOP              -- fall out of loop if greater
```

```
OUT:
    ... loop sequel ...
```

The code runs entirely in the Data Processor (in a uniprocessor, the Synchronization Processor is ignored completely).

Multiprocessor code: using rload's to mask latency

Let us consider what happens when X[I] and Y[I] are on remote nodes of a multiprocessor. Each of the two loads **(L1)** and **(L2)** need to be changed to remote loads. We will issue rload's to initiate the movement of X[i] and Y[i] into the local frame, and we will free up the processor to do other work. Each response arrives at the Synchronization Processor, deposits the value into the frame, and tries to join with the other response at frame location c1. When the join succeeds, the Synchronization Processor enables the thread in the Data Processor that computes with these data, executes an rstore and continues to the next iteration.

When the loop completes, it gives up the Data Processor by executing a next instruction. Meanwhile, the rstore acknowledgments all arrive at the Synchronization Processor and join at frame location c2. The continuation of this join is the loop sequel; the sequel executes only after all rstores have completed. Here is the augmented frame layout and the new code (it is easier to start reading the code at the data processor section):

	...
N	Loop trip
XP	*pointer to* X[1]
YP	*pointer to* Y[1]
A	*loop constant* A
YLim	*pointer to* Y[N]
XI	*copy of* X[I]
YI	*copy of* Y[I]
c1	*join counter for* rloads
c2	*join counter for* rstores
	...

```
;;
;;  Synchronization Processor Message Handlers
;;

L1S:
    store sFP[XI],   rV1        --  store away incoming X[I]
    join  c1, 2, CONTINUE_D     --  attempt continuation of loop
    next                        --  next message

L2S:
    store sFP[YI],   rV1        --  store away incoming Y[I]
    join  c1, 2, CONTINUE_D     --  attempt continuation of loop
    next                        --  next message

S1S:
    load  rN, sFP[N]            --  total number of stores
```

```
    join  c2, rN, OUT_D          -- do sequel after all stores complete

;;
;;  Data Processor Threads
;;

    load  rXP, dFP[XP]           -- load ptr to X
    load  rYP, dFP[YP]           -- load ptr to Y
    load  rYlim, dFP[YLim]       -- load loop constant: Y pointer limit
    cmp   rB,rYP,rYLim           -- compare ptr to Y with limit
    jgt   rB, OUT_D              -- zero-trip loop if greater

LOOP_D:

    rload rXP, L1_S              -- initiate load X[i]          (L1)
    rload rYP, L2_S              -- initiate load Y[i]          (L2)
    next
CONTINUE_D:
    load  rXI, dFP[XI]           -- load copy of X[I]
    load  rYI, dFP[YI]           -- load copy of Y[I]
    load  rA,  dFP[A]            -- load loop constant: a
    load  rXP, dFP[XP]           -- load ptr to X
    load  rYP, dFP[YP]           -- load ptr to Y
    load  rYLim, dFP[YLim]       -- load loop constant: Y pointer limit

    fmul  rTmp,rA,rXI            -- a*X[i]
    fadd  rTmp,rTmp,rYI          -- a*X[i] + Y[i]
    rstore rYP, rTmp,S1_S        -- store into Y[i]             (S1)
    add   rXP,rXP,8              -- increment ptr to X
    add   rYP,rYP,8              -- increment ptr to Y
    store dFP[XP],   rXP         -- store ptr to X
    store dFP[YP],   rYP         -- store ptr to Y
    cmp   rB,rYP,rYLim           -- compare ptr to Y with limit
    jle   rB, LOOP_D             -- fall out of loop if greater
    next

OUT_D:
     ... loop sequel ...
```

Analysis of Multiprocessor Code

Here we see the clear and unpleasant consequence of our choice that rloads deposit into the frame and not into registers. Now, the Data Processor performs more loads and stores on the current frame than in the uniprocessor case. The reason is that since we relinquish the Data Processor at the `next` instruction after the rloads, the registers may have changed by the time we get the Data Processor back at label $\texttt{CONTINUE}_D$. Thus, we have to repeatedly reload the the `X` and `Y` pointers and the loop constants `A` and `YLim`, and repeatedly store the incremented `X` and `Y` pointers back.

Now, in a multiprocessor, all these extra instructions may actually pay off when compared with nodes built with conventional processors, because *none of them are long latency operations*. Suppose we have a multiprocessor built with conventional processors and the same

load/store instructions are used for remote locations (*i.e.*, the code looks like the uniprocessor code presented earlier). Here is a comparison of the dynamic instruction counts of the Data Processor for the body of the inner loop for the two codes, generalized for k-way unrolling:

	Arith	Branch	Local		Remote	
			`load`	`store`	`load`	`store`
Conventional processor	$1+4k$	1	0	0	$2k$	k
			`load`	`store`	`rload`	`rstore`
*T	$1+4k$	2	$4+2k$	2	$2k$	k

Of the five arithmetic operations in the loop body, only the `cmp` is not replicated on loop unrolling. For *T, even in an unrolled loop, the two loop constants and the two pointers to `X` and `Y` need to be loaded only once and the two incremented pointers need to be stored only once; also, we count the `next` instruction as a branch.

Assume that arithmetic, branch and local loads and stores cost one time unit. For the conventional processor, assume that the remote loads cost an average of l units (processor stalls, waiting for the reference to complete), and assume that each remote store costs only 1 unit because it is likely to find the just-loaded `Y[i]` in the cache. For *T, we charge one unit for `rloads` and `rstores`, since they are non-blocking message sends. We charge no additional cost for the thread switch at `next`, assuming sufficient parallelism in the continuation queue to keep the processor busy with other work. Therefore, *T will take less overall execution time when,

$$9k + 9 \leq 2kl + 5k + 2$$

If the loop is unrolled 8 times ($k = 8$) then the *T code is preferred whenever the average global penalty is $l \geq 2.4$. That is, if the expected value of the remote load/store penalty is about 3 units or more, then the extra local instructions executed by the multiprocessor code are worth it.

As a calibration point, consider a cache coherent multiprocessor with miss penalty of 100 units (see for example the Stanford DASH[24]). If the cache miss rate exceeds about 2–3% then the multithreaded code would be prefered to relying on the cache.

Other optimizations

Speculative Avoidance of the extra `load`*'s and* `store`*'s*: The multithreaded code and coherent caching are not mutually exclusive. If fact, as Culler has observed [11], the multithreaded code overhead can be mitigated by speculating that the `rloads` and `rstores` will hit the cache. We only pay the cost of a thread switch in case the cache misses. In many ways, this is the same tradeoff offered by the April processor used in Alewife [1]. Here is the modified code.

```
;;
;;  Synchronization Processor Message Handlers
;;

L1s:
    store sFP[XI],   rV1        -- store away incoming X[I]
```

```
    join  c1, 3, CONTINUE_D      -- attempt continuation of loop
    next                         -- next message

L2_S:
    store sFP[YI],   rV1         -- store away incoming Y[I]
    join  c1, 3, CONTINUE_D      -- attempt continuation of loop
    next                         -- next message

L3_S:
    join  c1, 3, CONTINUE_D      -- attempt continuation of loop
    next                         -- next message

S1_S:
    load  rN, sFP[N]             -- total number of stores
    join  c2, rN, OUT_D          -- do sequel after all stores complete
    next                         -- next message

;;
;;  Data Processor Threads
;;

    load  rXP, dFP[XP]           -- load ptr to X
    load  rYP, dFP[YP]           -- load ptr to Y
    load  rA,  dFP[A]            -- load loop constant: a
    load  rYlim, dFP[YLim]       -- load loop constant: Y pointer limit
    cmp   rB,rYP,rYLim           -- compare ptr to Y with limit
    jgt   rB, OUT_D              -- zero-trip loop if greater

LOOP_D:

    rload rXP, L1_S              -- initiate load X[i]              (L1)
    rload rYP, L2_S              -- initiate load Y[i]              (L2)

    load  rC1, dFP[c1]           -- load join counter               (L3)
    cmp   rB, 2, rC1             -- Two responses arrived?          (C1)
    jeq   rB, FAST_CONTINUE_D    -- yes, skip save/restore          (J1)
                                 -- gamble failed; save modified regs
    store dFP[XP],   rXP         -- store ptr to X                  (S2)
    store dFP[YP],   rYP         -- store ptr to Y                  (S3)
    start L3_S, dFP, 0           -- start 3rd synch thread          (S4)
    next                         -- do something else

CONTINUE_D:                      -- here if gamble failed; restore regs
    load  rA,  dFP[A]            -- load loop constant: a
    load  rXP, dFP[XP]           -- load ptr to X
    load  rYP, dFP[YP]           -- load ptr to Y
    load  rYlim, dFP[YLim]       -- load loop constant: Y pointer limit

FAST_CONTINUE_D:                 -- directly here if gamble succeeded
    store dFP[c1], 0             -- re-initialize join counter
    load  rXI, dFP[XI]           -- load copy of X[I]
    load  rYI, dFP[YI]           -- load copy of Y[I]

    fmul  rTmp,rA,rXI            -- a*X[i]
```

```
fadd   rTmp,rTmp,rYI          -- a*X[i] + Y[i]
rstore rYP, rTmp,S1_S         --  store into Y[i]                (S1)
add    rXP,rXP,8              --  increment ptr to X
add    rYP,rYP,8              --  increment ptr to Y
cmp    rB,rYP,rYLim           --  compare ptr to Y  with limit
jle    rB, LOOP_D             --  fall out of loop if greater
next
```

OUT_D:
... loop sequel ...

Unlike the previous version, the data processor now loads all relevant values into registers before the loop, including the loop constants `a` and `ylim`, gambling that it can keep them there. After issuing the two rloads in statements **L1** and **L2**, it peeks at the join counter `c1` (**L3**, **C1**). If the two rload responses have already arrived (say, because of a cache hit), `c1` will have been incremented from 0 to 2; the gamble succeeds and we jump directly to $\texttt{FAST\_CONTINUE}_D$. In this case, we only load the values that arrived on the message; the loop constants and `X` and `Y` pointers are already in registers. Note also that after the two pointer increments (`add` instructions), we no longer store the pointers back into the frame.

If the gamble fails (`c1` < 2 in statement **C1**), we save modified registers (**S2**, **S3**), start a third (trivial) message handler in the synchronization coprocessor which will synchronize with the returning rload responses (**S4**), and switch to another thread. In the synchronization coprocessor, when the three message handlers at $\texttt{L1}_S$ $\texttt{L2}_S$, and $\texttt{L3}_S$ have executed (two rload responses, one started locally at **S4**), the data processor thread at $\texttt{CONTINUE}_D$ is initiated, which reloads the two loop constants and the two pointers into registers before proceeding.

In the above code, a number of tricks could be used to improve the probability that the gamble succeeds. Other useful instructions could be executed after the rloads and before peeking at the join counter. For example, if the loop is unrolled n times, all $2n$ rloads could be issued before looking for the responses for the first pair. We could even busy-wait a little, polling the join counter some number of times.

Loop splitting: The loop can be split into several loops working on independent chunks of the arrays, so that the remote loads of one loop are overlapped with computation in other loops, and vice versa.

Clearly, there are a number of challenging compilation issues. We will discuss them briefly in Section 7.

6 Other multithreaded architectures

We now examine several existing architectures to see how they address the fundamental problems due to latency and synchronization. We examine both "von Neumann" processors as well as "dataflow" processors, since our proposal for *T may be seen as a complete synthesis of these two previously separate approaches. Please note that our analysis is only from the narrow perspective of the framework outlined thus far; we do not examine technology issues, and each of the machines has many other very interesting and important features that are outside the scope of this paper.

*T is an evolutionary step that builds on the ideas of all the machines that we are about to discuss. There are several additional machines that have also influenced *T but, for lack of space, we omit a detailed comparison. Buehrer and Ekanadham proposed incorporating dataflow features in a von Neumann architecture [9], but this is subsumed by Iannucci's Hybrid machine. Arvind *et al*'s Tagged-Token Dataflow Architecture [5], ETL's Sigma-1 [19] and Gurd *et al*'s Manchester dataflow machine [17] are classic dynamic dataflow architectures, but these are subsumed by Monsoon. Other important dataflow architectures include Sandia's Epsilon dataflow architecture [16], Dennis and Gao's Argument-Fetching Dataflow machine [15] and Dennis' multithreaded architecture [14]. Nikhil and Arvind's P-RISC architecture [29], which simplified the synchronization mechanisms of Iannucci's Hybrid machine, is almost a direct predecessor of *T. Halstead and Fujita's MASA [18] and Nuth's Named State Processor [30] have had a less direct influence, but are nevertheless extremely interesting.

6.1 The Denelcor HEP and the Tera Computer System

The Denelcor HEP [38] and the Tera Computer System [2, 23, 39] are machines designed by Burton Smith and use multithreaded processors with replicated processor state and split transactions to mask the latency of remote memory access. For synchronization, all registers and memory locations have presence bits, and accesses can choose to test them. The Tera provides address-hashing with a system similar to the IBM RP3, where individual memory segments may be arbitrarily hashed across memory nodes in a programmable manner. We have the following concerns:

- Small continuation namespace: only 64 "process tags" per processor in the HEP, and 128 "i-stream tags" in the Tera. We have remarked that a small continuation namespace can seriously complicate compiling.

- Inadequate support for join synchronization: An instruction can busy wait on an empty register. Thus, the remote load program can be expressed as follows (using our own notation):

```
        ; main thread                ; forked thread
        fork M1                M1:   !RvB = rload RpB
        RvA = rload RpA              stop
   N:   RC  = RvA - !RvB
```

  Assume that register RvB is initially EMPTY. Fork initiates a second thread at M1 allowing the two loads to be issued concurrently. The "!" on RvB at M1 indicates that when the load completes, the register should be marked FULL. The "!" on RvB at N indicates that instruction should busy wait until register RvB is FULL. Thus, this solution is adequate only if the expected busy waiting is small, *i.e.*, it depends on the temporal locality of the two rloads and the bounded latency of the network.

  If temporal locality is not predictable, joins must be done in main memory. In the HEP, where all memories are equally distant, this can add significantly to message traffic. The Tera has local memories so message traffice due to joins is less of a problem.

However, in both machines suspending and resuming a thread must be done using usual software mechanisms.

- Inadequate support for the synchronizing loads problem: it can be solved using busy waiting (the Tera has more sophisticated busy waiting with a retry backoff mechanism and a retry limit trap) , but implementing the full data driven solution requires software process suspension and resumptions (like joins in memory).

- Single thread performance is derated by thread interleaving. In the HEP, a particular thread must traverse the entire pipeline (8 clocks) before it can reenter the execution pipe for the next instruction (even longer if there are more than 8 active threads). In the Tera, an ingenious lookahead mechanism allows upto 8 instructions from a thread to be in the pipeline, but the pipeline depth has increased to about 70 clocks, so a thread still cannot utilize the pipeline at full speed.

We noted at the end of Section 4 that busy waiting on remote loads is also supported in *T because the Data Processor can poll the the frame for the arrival of responses from other nodes.

6.2 The Dataflow/von Neumann Hybrid Architecture

The Dataflow/von Neumann Hybrid Architecture was proposed by Iannucci at MIT [22, 21] and followed by an implementation effort led by Iannucci at IBM's T.J. Watson Research Center (the Empire project).

The Hybrid machine is multithreaded, but does not interleave threads. It runs each thread exclusively in a conventional pipeline with instruction lookahead; when a thread suspends, the next one is automatically popped off a continuation queue. Local memory locations have presence bits; a thread can suspend by attempting to read an EMPTY location— the thread descriptor (one word) is stored in the location, to be ejected and reenabled later by a write into the location. Data driven synchronizing loads are supported with *I-structure* memory units. Thus, there is no busy waiting either on joins or on synchronizing loads. We have the following concerns:

- Without a separate message processor, short threads because of failed joins are still likely to be highly disruptive of the processor pipeline. Apart from its own occupation of the pipeline, precluding other threads from running, a failed join may also introduce bubbles because until the register-fetch stage, it is not known whether this thread is to be suspended or not.

- Since local memory reads and writes may involve storing and ejecting a thread descriptor, respectively, this can add significant complexity to the data paths and control structures of the processor pipeline.

6.3 The MIT Monsoon Dataflow Machine

There is a very close correspondence between the organization in Figure 5 and classical dynamic dataflow processor organization [7]: the message queue is the *token queue*, the synchronization coprocessor is the *wait-match* section and the main processor is the *execute* section. Dataflow architectures have evolved substantially over the years (see [5, 4, 3, 13, 17, 19]), and Monsoon [31, 32, 10] and EM-4 (discussed in the next Section) are the most recent representatives (please see [33] for an interpretation of Monsoon as a multithreaded architecture). Like the Hybrid machine, it performs joins using presence bits in local memory, and synchronizing loads using I-structure memory units. We have the following concerns:

- Like the HEP/Tera, single thread performance is derated due to interleaving of threads in the pipeline.
- Single thread performance in Monsoon is also reduced due to a paucity of registers (only three per thread) and limited addressing modes, but these may be regarded as an implementation restriction and not fundamental to the model.
- Referring to Figure 5, Monsoon does not have any separate continuation queue. We remarked that this lack of asynchrony causes short threads (due to join synchronizations) to be very disruptive. In Monsoon, every failed join causes a bubble in the execution unit.

6.4 The J-Machine, with Message Driven Processors

The J-Machine's nodes, with Message Driven Processors (MDPs) and local memory, comes very close to the organization of Figure 5 [12]: hardware support for message queueing, direct vectoring to message handlers, and direct access to message contents. As in *T, split transactions between two nodes are implemented by suitable coding of both ends. Message handlers contain arbitrary code and so can perform joins, synchronizing loads, *etc.* We have the following concerns:

- Because there is no separate processor to handle messages, message handling can disrupt the current thread. This is alleviated somewhat by a kind of lightweight interrupt model, with three levels of priority and three corresponding copies of the processor state. An interrupting message can be handled without saving and restoring the current thread state. However, they still share a single pipeline, which disrupts the current thread. Since there is no separate optimization for short and long threads, each priority level has the same small number of general purpose registers (four) and this, too, can limit single thread performance.
- The J-machine does not provide any hardware method to implement address-hashing. There is a hardware system to associatively map object identifiers to physical addresses, but this is primarily used to allow relocation of entire objects, where an entire object must reside within a single node.

6.5 ETL's EM-4 dataflow system with EMC-R processors

The EM-4 and EM-5 are dataflow machines designed at the Electrotechnical Laboratory in Tsukuba, Japan [36, 43, 35]. An 80 node prototype of EM-4 has been running since April 1990. The EM-4 processor (EMC-R) is almost exactly as shown in Figure 5. Our synchronization coprocessor corresponds to EMC-R's IBU (Input Buffer Unit) and FMU (Fetch and Matching Unit), and our main processor corresponds to EMC-R's EU (Execution Unit). Our threads correspond to EMC-R's *Strongly Connected Blocks* (and INBs, or *Indivisible Node Blocks*, in the EM-5). We have the following concerns:

- The continuation queue has length 1, reducing the asynchrony between message-handling (IBU/FMU) and thread execution (EU).
- Remote loads are not handled by the synchronization coprocessor (IBU/FMU), but by the Execution Unit (in this manner it is similar to the J-machine). We remarked that this is wasteful of the Execution Unit, disrupts it, and adds to the turnaround time for rload requests.
- EM-4, Monsoon, and the Dataflow/von Neumann Hybrid architectures all have the following problem. Because of their reliance on presence bits for joins, a multi-way join must be implemented as a tree of two-way joins involving a different local memory location for each one, possibly wasting the data slots of these locations.

7 Compiling for *T

Based on our discussion in Section 6, we believe that *T will efficiently support code developed for HEP/Tera, the J-machine, EM-4, *etc.*, because in each case it provides a superset of their capabilities. Thus, any compiler techniques developed for these machines are directly applicable to *T. Further, we believe that *all* these architectures will efficiently support SIMD/SPMD programming models, again because they are more general.

The real challenge is to provide truly general purpose programming models in which applications can have much more dynamic structure. Here, the problem is for the compiler to extract *excess parallism*, *i.e.*, parallelism that is reasonably greater than the number of nodes in the MPA (Valiant calls this *parallel slackness* [41]). This will allow each node to have sufficient threads so that even though some of them may be waiting for incoming messages, there is a high probability that other threads are ready to run.[1].

A very promising approach is to start with *declarative languages* (such as Haskell [20], Id [27], or Sisal [26]) where the compiler can effortlessly extract large amounts of fine grain parallelism. We have been working on compilers for Id for several years [40] now, based on dataflow graphs. Recently, Nikhil [28] and Culler *et al* [11] have proposed compiler flow-graph formalisms which, inspired by dataflow graphs and the P-RISC model, make explicit the notion of threads as a unit of scheduling. Culler's Threaded Abstract Machine (TAM)

[1] A MIMD is a terrible thing to waste!

has been implemented and is in use for over two years on a variety of platforms (RISC uniprocessors, shared memory multiprocessors, NCUBE/2, *etc.*). His experiments have provided a wealth of data on this compilation paradigm, concerning achievable thread lengths, frequency of synchronization, instruction counts, comparisons with dataflow instructions, *etc.*, all of which have had a strong influence on *T, which almost directly mirrors the TAM compilation model. The TAM work is an existence proof that compilation of large, non-trivial programs with massive amounts of parallelism for *T is possible. Nevertheless, there remain serious research issues concerning dynamic storage management, load balancing, *etc.*

Acknowledgements: Work is proceeding towards designing and constructing a real *T machine as a collaboration between MIT and Motorola, Inc. We would like to acknowledge, in particular, the contributions of Bob Greiner and Ken Traub of Motorola towards the architecture and software systems of *T, respectively.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. 17th Annual Intl. Symp. on Computer Architecture, Seattle, Washington, U.S.A.*, pages 104–114, May 28-31 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. Intl. Conf. on Supercomputing, Amsterdam*, June 11-15 1990.

[3] Arvind and S. Brobst. The Evolution of Dataflow Architectures from Static Dataflow to P-RISC. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy*, October 1989.

[4] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[5] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Technical report, MIT Lab. for Computer Science, August 1983. Revised October, 1984.

[6] 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.

[7] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[8] E. Brooks. The Attack of the Killer Micros, 1989. Presentationtation in the Teraflop Computing Panel Discussion, Supercomputing '89, Reno, Nevada.

[9] 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.

[10] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, December 1990.

[11] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. 4th Intl. Conf. on Achitectural Support for Programming Languages and Systems (ASPLOS), Santa Clara, CA*, pages 164–175, April 8-11 1991.

[12] W. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a Message-Driven Processor. In *Proc. 14th. Annual Intl. Symp. on Computer Architecture, Pittsburgh, PA*, pages 189–196, June 1987.

[13] J. B. Dennis. Data Flow Supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[14] J. B. Dennis. The Evolution of "Static" Data-Flow Architecture. In *Advanced Topics in Data-flow Computing, J-L. Gaudiot and L. Bic (eds.)*, pages 35–91. Prentice-Hall, 1991.

[15] J. B. Dennis and G. R. Gao. An Efficient Pipelined Dataflow Processor Architecture. In *Proc. Supercomputing Conference, Orlando, FL*, pages 368–373, November 14-18 1988.

[16] V. Grafe, G. Davidson, J. Hoch, and V. Holmes. The Epsilon Dataflow Processor. In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 36–45, May 29-31 1989.

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

[18] R. H. Halstead Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the IEEE 15th. Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, June 1988.

[19] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing*, 10(4):219–226, 1987.

[20] P. Hudak, S. Peyton Jones, and P. e. Wadler. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Technical report, Yale University, Department of Computer Science, August 1991.

[21] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Lab. for Computer Science, May 1988. Ph.D. thesis.

[22] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the IEEE 15th. Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, pages 131–140. IEEE/ACM, June 1988.

[23] J. T. Kuehn and B. J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Prof. IEEE Supercomputing Conference, Florida*, pages 28–34, 1988.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. 17th. Annual Intl. Symp. on Computer Architetecture, Seattle, Washington*, pages 148–159, May 28-31 1990.

[25] P. M. Maurer. Mapping the data flow model of computation onto an enhanced von neumann processor. In *Proc. Conf. on Parallel Processing*, pages 235–239, August 1988.

[26] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, P. Hohensee, and I. Dobes. SISAL Reference Manual. Technical report, Lawrence Livermore National Lab., 1984.

[27] R. S. Nikhil. Id (Version 90.1) Reference Manual. Technical Report CSG Memo 284-2, MIT Lab. for Computer Science, July 15 1991.

[28] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy*, October 1989.

[29] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 262–272, May 29-31 1989.

[30] P. R. Nuth and W. J. Dally. The Named State Processor. Technical report, MIT AI Lab, November 22 1989.

[31] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* MIT Press, 1991.

[32] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[33] G. M. Papadopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proc. 18th. Intl. Symp. on Computer Architecture, Toronto, Canada*, March 1991.

[34] 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 *Proc. IEEE Intl. Conf. on Parallel Processing*, pages 764–771, August 1985.

[35] S. Sakai, Y. Kodama, and Y. Yamaguchi. Architectural Design of a Parallel Supercomputer EM-5. In *Proc. Japan Soc. Parallel Proc., Kobe Japan*, pages 149–156, May 14-16 1991.

[36] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, May28-June 1 1989.

[37] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, August 1991. Springer-Verlag LNCS 523.

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

[39] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *Prof. IEEE Supercomputing Conference, Florida*, pages 35–41, 1988.

[40] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Lab. for Computer Science, August 1986.

[41] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. of the ACM*, 33(8):103–111, August 1990.

[42] W.-D. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 273–280, May 29-31 1989.

[43] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba. An Architectural Design of a Highly Parallel Dataflow Machine. In *Proc. Information Processing 89, San Francisco, USA*, pages 1155–1160, August 28-September 1 1989.