LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

*T: A Multithreaded Massively Parallel Architecture

Computation Structures Group Memo 325-2
March 5, 1992

(Replaces CSG Memos 325 (Jan 1991) and 325-1 (Nov 1991))

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

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

In *Proc. 19th. Intl. Symp. on Computer Architecture*,
May 19-21, 1992, Queensland, Australia

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

R. S. Nikhil [00]
Digital Equipment Corp.

G. M. Papadopoulos [01]
MIT

Arvind [11]
MIT

Abstract

What should the architecture of each node in a general purpose, massively parallel architecture (MPA) be? We frame the question in concrete terms by describing two fundamental problems that must be solved well in any general purpose MPA. From this, we systematically develop the required logical organization of an MPA node, and present some details of *T (pronounced *Start*), a concrete architecture designed to these requirements. *T is a direct descendant of dynamic dataflow architectures, and unifies them with von Neumann architectures. We discuss a hand-compiled example and some compilation issues.

1 Introduction

There appears to be widespread consensus that general purpose supercomputers of the future will be Massively Parallel Architectures (MPA's) 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 the logical node organization of *T, that addresses these problems. Pronounced *Start*, it is named after the principal communication instruction in the machine. It can also be read as an acronym for "multi- (*) Threaded", a principal feature of the machine. In Section 4 we present some instruction set details of the logical organization of *T, and in Section 5 we discuss the coding of DAXPY. In Section 6 we discuss possibilities for physical implementation of *T, and we conclude in Section 7 with some discussion of compilation issues.

*T is the latest step in the evolution of dynamic dataflow architectures, starting with the MIT Tagged-Token Dataflow Architecture (TTDA, [5, 7]), the Manchester dataflow machine [16] and the ETL SIGMA-1 [19]. In 1987-88, the associative waiting-matching store of these machines was replaced by an explicitly managed, directly addressed token store, by Papadopoulos and Culler in the Monsoon architecture [25, 10]. Nikhil and Arvind's P-RISC architecture [24] then split the "complex" dataflow instructions into separate synchronization, arithmetic and fork/control instructions, eliminating the necessity of presence bits on the token store (also known as "frame memory"). P-RISC also permitted the compiler to assemble instructions into longer threads, replacing some of the dataflow synchronization with conventional program counter-based synchronization (in retrospect, Monsoon could also be viewed in this light [26]). *T is the latest step in this direction, eliminating the prior distinction between processor and I-structure nodes, and attempting also to be compatible with conventional MPA's based on von Neumann processors.

Throughout this evolution, our *system view* of an MPA (*i.e.,* viewing each node of an MPA as a black box) has remained constant, and is in fact quite unique to dataflow architectures. Inter-node traffic consists of tokens (*i.e.,* messages) which contain a *tag* (or *continuation*, a pair comprising a context and instruction pointer) and one or more values. All inter-node communications are performed with *split-phase* transactions (request and response messages); processors never block when issuing a remote request, and the network interface for message handling is well integrated into the processor pipeline. Thus, TTDA, Monsoon, P-RISC and *T are properly viewed as steps in the evolution of the *node architecture* of dataflow MPA's.

Important architectural influences in our thinking have been the seminal Denelcor HEP [30] and Iannucci's Dataflow/von Neumann Hybrid architecture [20], which incorporated some dataflow ideas into von Neumann processors for MPA's (similar ideas were also expressed by Buehrer and Ekanadham [9]). ETL's EM-4 [28], Sandia's Epsilon [15], Dennis and Gao's Argument-Fetching Dataflow machine [14], and Dennis' multithreaded architecture [13] are also dataflow machines supporting threads. Finally, the Network Interface Chip by Henry and Joerg [18] has had a direct influence on our implementation of *T.

[00] This work was done while the author was at MIT. *Address*: Digital Cambridge Research Laboratory, One Kendall Square, Bldg 700, Cambridge, MA 02139, USA;
email: `nikhil@crl.dec.com`; *phone*: 1 (617) 621 6639

[01,11] *Address*: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA;
email: {`greg,arvind`}`@mit.edu`; *phone*: 1 (617) 253 {2623,6090}

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.

Software considerations, *i.e.,* compiling, have played an equally important role in the design of *T. Compilation of the high-level parallel programming language Id has been central to our research since the beginning [7]. Nikhil, in [23], 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 and synchronization [29]. This experience has directly influenced the *T design.

Many researchers have arrived at similar mechanisms starting from other vantage points (*e.g.,* Halstead and Fujita's MASA [17] and Maurer's SAM [22]). There are very interesting comparisons to be made between *T and recent multithreaded architectures, notably the Tera Computer System [3] (a descendant of the HEP), Alewife [1] and the J-Machine with Message Driven Processors (MDPs) [12]. Undoubtedly, aspects of these machines will influence concrete *T designs.

2 Fundamental Problems in an MPA

A general purpose Massively Parallel Architecture (MPA) consists of a collection of nodes that communicate using a pipelined, message based interconnection network (see Figure 1). Each node contains a processor and some local memory. Given a node N, we refer to its own memory as *local memory* and to other memories as *remote memories*.

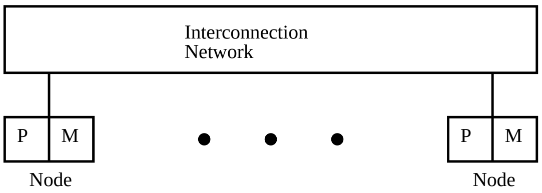

Figure 1: Structure of a massively parallel architecture.

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.

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 remote access time will vary with $\log N$, where N is the number of nodes.

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].

2.1 The Latency of Remote Loads

The remote load situation is illustrated in Figure 2. Variables `A` and `B` are located on nodes N2 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 executing on N1:

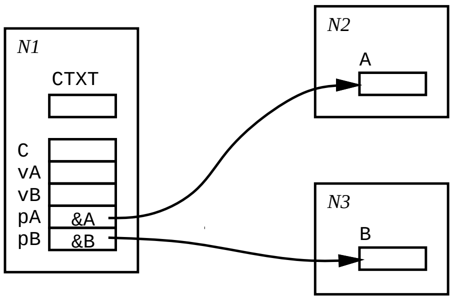

Figure 2: The remote load problem: Node N1 to compute difference of variables in nodes N2 and N3

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

where `pA` and `pB` are pointers to `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`.

The figure also shows 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, 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 N1's processor during the remote load operations. Note that in most architectures, these latencies are predictable because they depend mainly on the distance between the nodes.

2.2 Idling due to Synchronizing Loads

We extend the remote load problem by now assuming that `A` and `B` are 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. This is an example of producer-consumer parallelism.

Unlike the remote load problem, the latency here is not just an architectural property— it also depends on scheduling, and the time it takes to compute `A` and `B`, which may be much longer than the transit latency. In particular, the unpredictable latencies of synchronizing loads are a problem *even when they are local.*

3 Solutions

3.1 Caches

Classically, caches are used to mask memory latency, and we could add a cache to each node to hold copies of remote locations. However, *cache coherence* is a then significant problem: multiple copies of a location residing in different nodes' caches must be kept consistent. Further, a processor may still idle 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, to multiplex between a small number of contexts to cover cache loading. Implementing this appears non-trivial (see MIT's MASA [17] and Alewife [1], and Stanford's DASH [21], for example). In any case, *the proposals in this paper may be seen as orthogonal to cacheing solutions.* Further, while distributed cacheing helps in the remote load situation, it does not offer anything for the synchronizing load problem.

3.2 Multithreading

Another solution is to multiplex amongst many threads: when one thread issues a remote load request, the processor begins work on another thread, and so on. Clearly, the cost of thread-switching 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 it can be re-enabled on the arrival of a response. We refer to these thread identifiers as *continuations.*

Adequate continuation namespace: The longer the 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 greatly affect the programming model. 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 *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 in turn limits the programming model.

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.

The size of the hardware-supported continuation namespace varies greatly in existing designs: from 1 in DASH [21], 4 in Alewife [1], 64 in the HEP [30], and 1024 in the Tera [3], to the local memory address space in Monsoon [25], Hybrid Dataflow/von Neumann [20], MDP [12] and *T. Of course, if the hardware-supported namespace is small, one can always virtualize it by multiplexing in software, but this has an associated overhead.

3.3 Fine Grain Multithreading: forks and joins

So far, we have only talked about efficient multiplexing amongst existing threads. However, if thread creation and synchronization are also sufficiently cheap, the multithreading idea can be advantageously taken further. Instead of the `rload-rload`-subtract sequence earlier, we could fork separate threads for the two `rloads` and synchronize them when both have completed. Then, instead of taking four message-transit times, we could perhaps do it in two (since they occur in parallel):

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

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

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

This kind of parallelism is available in all dataflow architectures [4]. `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` (initialized to 0, we assume) and compares it to the terminal count (2). If `J < 2`, 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, 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 modern RISC techniques such as delayed loads, cache prefetching, *etc.* are also modest, "peephole" examples of the use of fork-join parallelism to overlap long-latency loads with useful computation, but they do not generalize well to remote loads in MPA's.

The behavior at `N` is called a *join synchronization*, and is very important. 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 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 both communication and the time to perform the subcomputations.

This leads to another 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 a message format is:

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

Node address NA is used for routing, and may be consumed in transit. `Msg_type` specifies how the message should be handled. The remaining arguments depend on the message type. Figure 3 shows the messages for an `rload` where `&A` and `*A` mean the address and contents of `A`, respectively.[1] The values `N1`, `L1` and `CTXT` together constitute a *continuation*, *i.e.,* a complete specification of the thread awaiting the response.

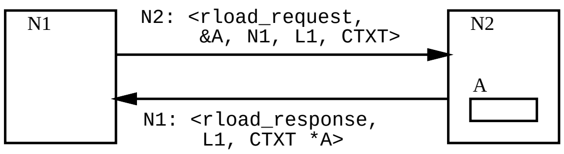

Figure 3: Messages to perform a remote load.

For fast vectoring to a message handler, it is convenient for the "message type" field to *be* the instruction pointer of the handler:[2]

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

To keep up with high incoming message rates, a hardware *network interface* can accept and queue them, to be consumed by the processor at its convenience (Figure 4). Of course, this may add to the latency of a remote load. The *message formatter* constructs outgoing messages.

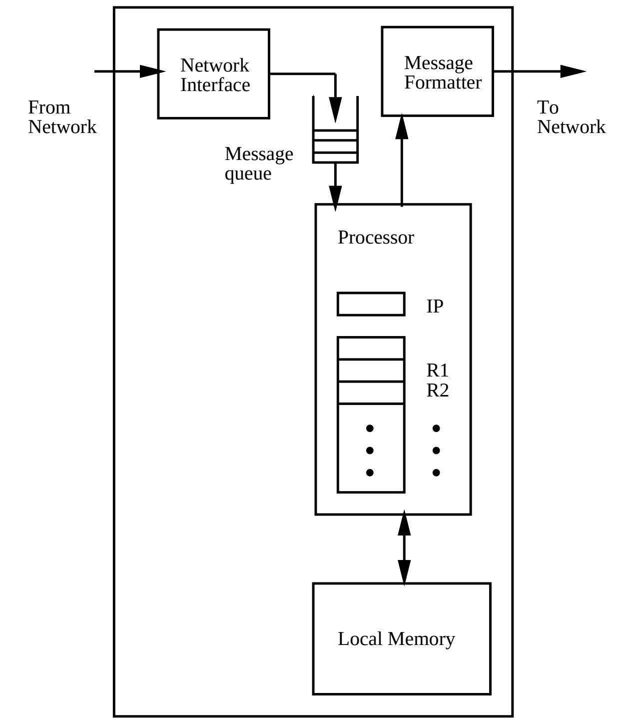

Figure 4: A node with network interface.

The actual handling of an `rload` request at N2 is simple and short: read location `A` and send the response message to N1. Handling the response at N1 may be broken into two parts: (a) completing the unfinished `rload` instruction, and (b) continuing the execution of the thread. Part (a) is always brief. Part (b) is also brief if it just executes a failing `join`.

Both N1 and N2 will typically be executing some other thread when these messages arrive. How can we handle these messages quickly (so that latency is low) and with minimum disruption, *i.e.,* how well is message-handling integrated with normal processing in the node? *The answer to this question is crucial to the efficient operation of an MPA.* The overhead can vary from hundreds of instruction-times in Thinking Machines' CM-5 and Intel's Touchstone, where message-handling is not well integrated, down to under ten instruction-times in the MDP [12], and further down to a single instruction-time in machines like Monsoon [25] and the EM-4 [28] where message-handling is totally integrated into the processor pipeline.

There is good reason to insist that message-handling *should not compromise excellent single-thread performance* in an MPA node. First, existing SIMD/SPMD applications have long sequential threads. Second, even in more dynamic applications, certain sequential critical sections (such as storage allocators) must be executed as quickly as possible. Third, users may wish to develop code on a 1-node MPA. Fourth, uniprocessor libraries should be portable to MPA nodes without degrading their performance. We note that

[1] Figure 3 depicts the logical information in messages; clever encoding can compress it significantly.

[2] Dally's Message Driven Processor [12] also uses an IP in the first field of a message.

competitive single-thread performance is an important goal in the Alewife [1] and Hybrid Dataflow/von Neumann [20] machines.

Interrupts are inadequate: An arriving message could interrupt the processor, which responds, and then resumes normal computation. Modern processors achieve high speeds through deep pipelining and more processor state (registers, caches), and any change in context incurs a large penalty draining, saving, flushing, reloading, *etc.* Thus, frequent short threads, such as `rload` handlers and messages that go into failing joins are likely to be severely disruptive. This may be alleviated by replicating processor state so that the main thread's state does not have to be saved and restored during interrupts. Still, the main thread's performance is affected adversely, because each incoming message takes away cycles and may require processor pipelines to drain and refill. Frequent interrupts for message handling are thus not desirable.

Separate Coprocessors: We can offload the burden of short, simple handlers into separate, asynchronous coprocessors, as shown in Figure 5. This is only a functional decomposition, or logical organization of the node; we will discuss physical organizations in Section 6.

We will refer to the original, conventional main processor as the *data processor* (dP). It executes threads supplied through the *continuation queue* and is optimized for long, sequential runs.

The *remote memory request coprocessor* RMem handles incoming `rload` requests. It accesses local memory, and sends the response directly, without disturbing dP. Similarly, RMem also handles `rstore` requests.

The *synchronization coprocessor* sP handles `rload` responses returning to this node. The contents of the message are loaded directly into its registers, with the first message field (IP) loaded into its PC, so that it is vectored directly to the appropriate handler. The basic action of sP is thus to repeatedly load a message from its message queue and run it to completion; the run is always non-suspensive and is expected to be very short. sP completes the unfinished `rload` by storing the value from the message (now in `sV1`) into the destination location. After this, it places the `<L1+1,CTXT>` continuation into the *continuation queue*, to be picked up later by the main processor dP. Even better, if the next instruction is a `join`, sP executes it and enqueues the continuation only if it succeeds. Thus, the main processor `dP` does not have to execute disruptive `join` instructions at all.

3.5 Handling Synchronized Loads

We observed the need for synchronization if the remote loads for `A` and `B` must wait for corresponding producer computations. We follow the well-known idea of appending *presence bits* to each memory location indicating whether it is FULL or EMPTY.[3] The variables `A` and `B` are initially marked EMPTY. When values are written there, they are marked FULL, *e.g.,* by the RMems when they perform the stores. Thus, the remote loads should succeed only after the locations become FULL.

[3] Our development does not depend on it— presence bits could be interpreted in ordinary memory.

Busy waiting: When N1's `rload` request for `A` arrives, N2 could simply return the entire location, presence bits and all. N1 would then test the response, and may choose to try again, perhaps after some delay. Unfortunately, this can waste network bandwidth, and may also waste cycles on N1 in order to test and perform the retries. This is essentially the solution in the Tera [3] and Alewife [1], using traps instead of explicit tests.

A data driven solution: We extend the presence bits to have an additional state, PENDING. For a remote synchronizing load, node N1 executes 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, the `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. In all cases, N2's response message for an `iload` looks identical to an `rload` response.

In general, many loads may be pending at `A`, so N2 needs to remember a set of pending requests at location `A`. Thus, one of the costs is that N2 must perform some storage management to construct this set. However, a simple free-list strategy may be used, since set 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 is essentially the dataflow notion of *I-structures* [7], but we note that it is more general— a variety of other synchronization primitives (such as Fetch-and-ϕ [2]) could be implemented using the same methodology. Finally, note that these message handlers are still simple enough to be handled entirely by the RMem coprocessor, without disturbing dP.

4 *T Instruction Set Architecture

Figure 5 depicts the logical view of a *T node with three separate, asynchronous coprocessors, each of which can be regarded as a conventional (*e.g.,* RISC) processor with a few extensions. In this section, we flesh out this view by describing these extensions. Since RMem and sP are very similar, our description here will not distinguish them (we postpone to Section 6 a more detailed discussion of physical implementations that may separate or multiplex them).

Both sP and dP execute instruction streams. For clarity, we distinguish labels for the two coprocessors by the subscripts S and D, respectively (*e.g.,* $\mathtt{L}_S$, $\mathtt{L}_D$, $\mathtt{M}_S$, $\mathtt{M}_D$, ...).

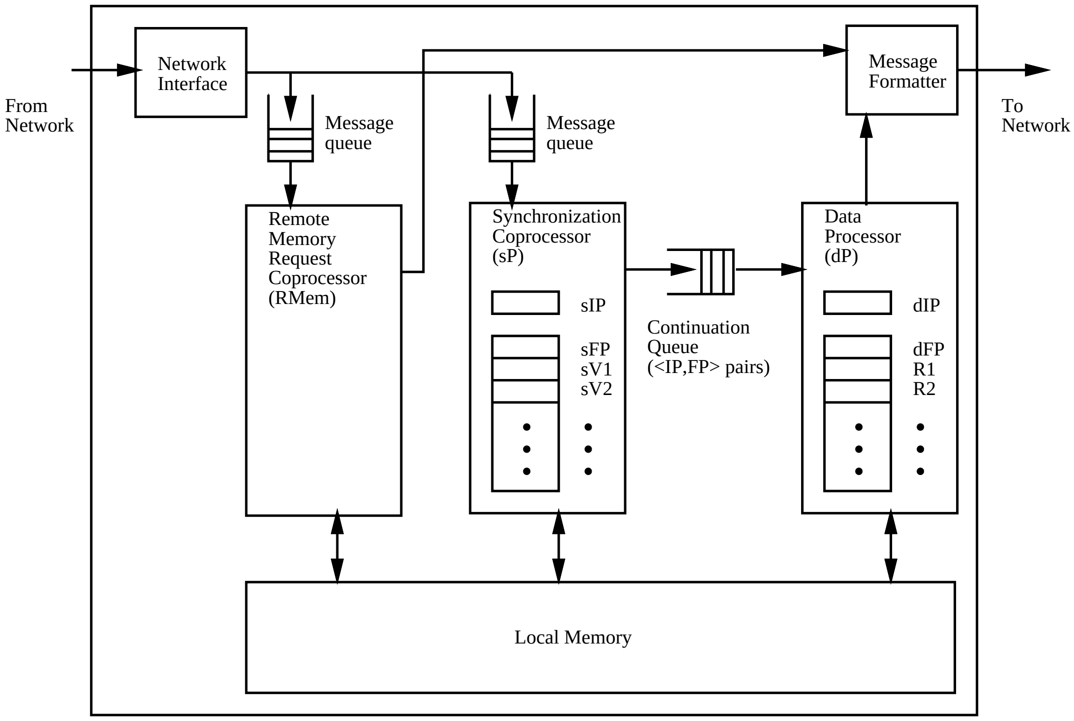

Figure 5: A *T node: separate coprocessors for short threads.

We will look at messages in more detail shortly, but for now 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 of 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 (see [23] and [11, 29] for more on programming and compilation).

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 MDP [12] or Empire.[4]

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 conventional RISC instructions, the Data Processor can execute a few additional instructions which are summarized in Figure 6. (The notation `Register[`j`]` refers to the j'th DP register.) Most of them send messages into the network. These are non-blocking, *i.e.,* the Data Processor continues executing after sending a message (so, they are implicit forks). 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.

The `start` instruction starts a new thread in a different

[4] Empire was the machine being built at IBM Research based on Iannucci's Dataflow/von Neumann Hybrid Architecture [20].

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>`

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

Figure 6: DP extensions to conventional instruction set.

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. 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 SP, not DP.[5]

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 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 [25] and the MDP's SUSPEND [12].

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

4.2 The Synchronization Coprocessor

SP, 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, it is tuned for very rapid disposition of incoming messages instead of the computation of arithmetic expressions. In particular, some of its registers (`sIP`, `sFP`, `sV1`, `sV2`, ...) are loaded directly from messages, there is datapath support for `join` operations, and it can post `<FP,`L_D`>` pairs into the continuation queue. In the following, we do not disinguish it from RMEM, which is very similar.

Like a dataflow processor, SP is triggered by the arrival of a network message (it simply waits, if there is none available). When it picks up a message, its registers `sIP`, `sFP`, `sV1`, `sV2`, *etc.* are loaded with the corresponding values from the

[5] A `start` message corresponds exactly to a dynamic dataflow token [7]— `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.

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 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`

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

Figure 7: SP extensions to conventional instruction set.

message, after which it begins executing instructions from the address in `sIP`. The unconventional instructions of `sP` are summarized in Figure 7. (The notation `Register[`j`]` refers to the j'th SP register.)

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 contains 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_handlerS:
  load   rX, sFP[0]    -- sFP has addr 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_handlerS:
  store  sFP[0], sV1   -- sFP has addr 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. Similar handlers may also be coded on the MDP [12].

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 SP 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 message may be the response to a remote load):

```
L_S:
  store  sFP[T], sV1 -- store incoming val at frame offset T
  post   L_D, sFP    -- enable thread L_D  with
                     --      this frame in DP
  next               -- done; handle next message
```

The DP thread at L_D is assured that the remote load has completed.

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 (this is similar to the wait-match operation in dataflow architectures [4]).

sP message handlers correspond almost exactly to *inlets* in the TAM model [11]. This style of message handling is also easy to code for the MDP [12], 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 DP, 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`. Since `rload` and `rstore` are also forks, we can initiate many remote accesses before receiving any response. The responses may return in any order.

Note that the typical `rload` sequence copies a remote value into a local *frame* location, and not a register. Although this means that another (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 DP 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.

Address Hashing: It is useful 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 MPA nodes. 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 with large overhead. In *T we intend to provide hardware support for address randomization in a manner similar to the IBM RP3 [27] or the Tera [3]. A logical place for this mapping 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 (in the codes below, the `dFP` and `sFP` registers point at this frame):

	...
N	Loop trip
XP	*pointer to* X[1]
YP	*pointer to* Y[1]
A	*loop constant* A
YLim	*pointer to* Y[N]
	...

5.1 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 compiler would typically do this).

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

LOOP:
  load  rXI, rXP          -- X[i]  into rXI    (L1)
  load  rYI, rYP          -- Y[i]  into rYI    (L2)
  add   rXP,rXP,8         -- incr 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         -- incr 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 within the Data Processor (in a uniprocessor, sP plays no role).

5.2 Multiprocessor Code: Using `rloads` to Mask Latency

Suppose `X[I]` and `Y[I]` are on remote nodes of a multiprocessor. The two loads (**L1**) and (**L2**) need to be changed to remote loads. We will issue `rloads` 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 sP, deposits the value into the frame, and tries to join with the other response at frame location `c1`. When the join succeeds, sP 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 sP 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
;;
L1_S:
  store sFP[XI],   rV1     -- store away incoming  X[I]
  join  c1, 2, CONTINUE_D  -- attempt continuation of loop
  next                     -- next message

L2_S:
  store sFP[YI],   rV1     -- store away incoming  Y[I]
  join  c1, 2, CONTINUE_D  -- attempt continuation of loop
  next                     -- next message

S1_S:
  load  rN, sFP[N]         -- total number of stores
  join  c2, rN, OUT_D      -- sequel when 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 const: Y  ptr 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 const: a
  load  rXP, dFP[XP]       -- load ptr to  X
  load  rYP, dFP[YP]       -- load ptr to  Y
  load  rYLim, dFP[YLim]   -- load loop const:  Y  ptr 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 ...
```

5.3 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 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,* they are all local. Suppose we have an MPA 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	Br	Local Ld	Local St	Remote Ld	Remote St
Conventional processor	$1+4k$	1	0	0	$2k$	k
*T	$1+4k$	2	$4+2k$	2	$2k$	k

Ld = `load`, St = `store`, k = degree of unrolling

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 (the 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[21]). If the cache miss rate exceeds about 2–3% then the multithreaded code would be prefered to relying on the cache.

5.4 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 `rload`s and `rstore`s 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
;;
L1_S:
  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    --  sequel when 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 const: a
  load  rYlim, dFP[YLim] --  load loop const: Y  ptr 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:              --  gamble failed; restore regs
  load  rA,  dFP[A]      --  load loop const: a
  load  rXP, dFP[XP]     --  load ptr to  X
  load  rYP, dFP[YP]     --  load ptr to  Y
  load  rYlim, dFP[YLim] --  load loop const: Y  ptr limit

FAST_CONTINUE_D:         --  directly here if gamble succeeds
  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 `rload`s 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 sP which will synchronize with the returning `rload` responses (**S4**), and switch to another thread. In sP, 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.

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

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 some software issues in Section 7.

6 Implementation Considerations

So far, *T should be viewed as a *logical* model for a node in a multithreaded MPA; specialized, separate coprocessors are otherwise familiar RISC processors extended with instructions for the rapid creation and consumption of network messages, and for the efficient synchronization and scheduling of lightweight threads (in fact, many of the "extended" instructions can be realized as memory-mapped operations). In this section we explore possible physical realizations of the *T node architecture.

6.1 Coprocessor Data Sharing and Loading Issues

All three coprocessors share the same local memory, and their view must be consistent. Generally speaking, there is a significant sharing of data between the sP and dP, but relatively little sharing between either of these and the RMEM. The data values on *every* message handled by the sP are written into frames and are subsequently read by the dP. It is also likely that the dP will initialize join counters (or query them) which are subsequently modified by the sP. In contrast, data sharing between the dP and RMEM is limited to those structures which are both globally accessed by other nodes using split phase transactions, and locally accessed using traditional load/store operations. Thus, if the three coprocessors are implemented as distinct units (presumably with each having a cache for local memory) then we can expect a large volume of coherence traffic between the sP and dP coprocessors.

The relative load among the coprocessors will also influence an implementation. The sP typically executes several more threads than the dP but the threads are apt to be proportionately shorter. Assuming an average `join` arity of about four messages corresponding to an average data thread length of approximately 15 instructions, then the two coprocessors should execute roughly the same number of instructions (assuming three or four instructions per sP message handler).

In terms of network messages, the accounting is straightforward. Every `rload` or `rstore` request generated by a dP is processed by some RMEM which subsequently generates a `start` message for the requester's sP. Added to the remote memory traffic are dP-generated `start` messages corresponding to function arguments and results. For many applications the remote memory traffic dominates this extra traffic. Thus, assuming a uniform distribution of remote memory requests, an RMEM will process about as many messages as an sP.

6.2 Multiplexing the Coprocessors

While the workload appears nicely balanced among the coprocessors, engineering considerations may suggest multiplexing them on common resources. For example, we are actively pursuing a design where the sP and dP are combined onto a single chip. The pipeline and caches are time multiplexed on a thread-by-thread basis. By sharing the same data cache the sP–dP coherence traffic is eliminated. Obviously, the throughput of the node will suffer because we will not be able to exploit concurrency between the sP and dP. A more subtle consequence is pressure on the incoming message queue. Data processor threads are typically several times longer than message handlers. If data threads execute to completion before executing message handlers, then the variance on message service times will cause increased queuing of network messages.

In the ETL EM-4 [28], the Fetch and Matching Unit (FMU) corresponds to our sP and the Execution Unit (EU) corresponds to our dP. There is no separate RMEM— its function is performed by the EU. Further, the FMU is not programmable like our sP; rather, it is specialized for various *wait-match* operations similar to Monsoon [25] (two-way joins on frame presence bits, accompanied by a single frame read/write). Also, the FMU and the EU share a single port to local memory.

An implementation might go further and multiplex all three coprocessors onto the same shared resource (as in the MDP). This might be considered when remote memory operations are network bandwidth limited anyway. However, we would still consider this an improvement over, say, a CM-5 node (represented by Figure 4), because it still provides exceptionally efficient support for `join`, `start`, `rload`, `rstore`, and `next`. That is, we distinguish *T architectures by their direct implementation of synchronization and scheduling primitives for a potentially large set of very lightweight threads *in addition to* a tightly integrated network interface wherein internode communication is exposed to the compiler.

An orthogonal dimension of multiplexing is that multiple threads can be interleaved in the processor pipeline, as in the HEP [30], the Tera [3] and Monsoon [25]. This can simplify the pipeline, since the pressure due to inter-instruction dependencies is less, but it also degrades single thread performance by the degree of multiplexing.

7 Software for *T

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 general purpose programming models in which applications can have much more dynamic structure. Here, the problem is the extraction of *excess parallism*, *i.e.,* parallelism that is reasonably greater than the number of nodes in the MPA. This will allow each node to have sufficient threads so that even though some may be waiting for incoming messages, there is a high probability that other threads are ready to run.[6].

A very promising approach is to start with *declarative languages* where the compiler can effortlessly extract large amounts of fine grain parallelism. Nikhil [23] and Culler *et al* [11] have proposed compiler flowgraph 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) 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 much data on 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

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

for *T is possible. Nevertheless, there remain serious research issues concerning dynamic storage management, load balancing, *etc.*

Acknowledgements: MIT and Motorola, Inc. are collaborating on the design and construction of a real *T machine. Bob Greiner and Ken Traub of Motorola have contributed significantly towards the architecture and software systems, respectively.

References

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

[2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1989.

[3] 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 1990.

[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, USA, 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, Aug. 1983. Revised October, 1984.

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

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

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

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

[10] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *J. Parallel and Distributed Computing*, Dec. 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 Arch. Support for Programming Languages and Systems (ASPLOS), Santa Clara, CA*, pages 164–175, Apr. 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. Ann. Intl. Symp. on Computer Architecture, Pittsburgh, PA*, pages 189–196, June 1987.

[13] 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.

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

[15] V. Grafe, G. Davidson, J. Hoch, and V. Holmes. The Epsilon Dataflow Processor. In *Proc. 16th. Ann. Intl. Symp. on Computer Architecture, Jerusalem, Israel*, pages 36–45, May 1989.

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

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

[18] D. S. Henry and C. F. Joerg. The Network Interface Chip. Technical Report CSG Memo 331, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, June 1991.

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

[20] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Ann. Intl. Symp. on Computer Architecture, Honolulu, Hawaii*, pages 131–140, June 1988.

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

[22] P. M. Maurer. Mapping the Data Flow Model of Computation onto an Enhanced von Neumann Processor. In *Proc. Intl. Conf. on Parallel Processing*, pages 235–239, Aug. 1988.

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

[24] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. 16th. Ann. Intl. Symp. on Computer Architecture, Jerusalem, Israel*, pages 262–272, May 1989.

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

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

[27] 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. Intl. Conf. on Parallel Processing*, pages 764–771, Aug. 1985.

[28] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Ann. Intl. Symp. on Computer Architecture, Jerusalem, Israel*, pages 46–53, May 1989.

[29] 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 (Springer-Verlag LNCS 523)*, pages 50–72, Aug. 1991.

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