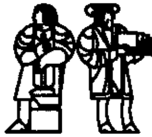
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

*T: a Killer Micro for a Brave New World

Computation Structures Group Memo 325
January 5, 1991

Rishiyur S. Nikhil
Gregory M. Papadopoulos
Arvind

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

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

*T: a Killer Micro for a Brave New World

Rishiyur S. Nikhil
Gregory M. Papadopoulos
Arvind

Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

The slogan "Attack of the Killer Micros" expresses the broadly held consensus that future supercomputers will consist of thousands of computing nodes in a fast, regular interconnection network, where each node is built using commodity microprocessors and memory parts. With this in mind, we examine the question: what should tomorrow's commodity microprocessors look like? After analyzing the complementary strengths and weaknesses of today's microprocessors and today's dataflow processors, we synthesize a new node architecture that employs the best of both designs. The new node architecture is called *T (pronounced "start"). The new architecture is fully compatible with existing sequential and parallel codes, but can also efficiently support fine grain parallelism such as that found in dataflow codes.

1 Introduction

Many of us are interested in the goal of general purpose computing that achieves very high speeds by exploiting parallelism in a scalable, cost-effective way. There seems to be widespread consensus that the architecture of such machines will be composed of a number of nodes interconnected with a high speed, regular network, where each node is built with an off-the-shelf microprocessor. Because such machines are built out of commodity parts, and because the topology is scalable, it is felt that such a machine with hundreds or thousands of nodes will be cheaper and faster than classical supercomputers, which are built with exotic technology and are thus very expensive. Classical supercomputers are thus viewed as a breed whose time has passed. This view has been eloquently summarized in the slogan "attack of the killer micros", by Eugene Brooks of Lawrence Livermore Laboratories [9].

In this paper, we focus on the architecture of each processor of such a massively parallel machine. To date, the prevailing opinion seems to be that microprocessors have their own evolutionary momentum (from CISC to RISC and, now, to multiple instruction issue), and that a massively parallel machine will simply track this wave, using whatever micros are currently available to populate the cells of its hive.

However, a massively parallel machine is in fact a hostile environment for today's micros, arising largely because certain properties of the memory system in a massively parallel machine are fundamentally different from those assumed during the evolution of these micros. In particular, most micros today assume that all memory is equally distant, and that memory access time can be made effectively small by cacheing. Both these assumptions are questionable in a massively parallel machine.

On the other hand, dataflow processors have been designed from the start by keeping in mind the properties of the memory system in a parallel machine. However, past dataflow processor designs have neglected single-thread performance, and hence must be classified as exotic, not the kind of processor to be found in commodity workstations.

How, then, can we evolve a micro that is more hardy and resilient, and is more suited to the modern environment, where it must live and flourish both in massively parallel machines and in uniprocessor workstations? In this paper, we propose such an evolutionary step—the *T processor architecture[1]. Based on the best genes of its von Neumann and dataflow parents, *T is engineered for the brave new world of massive parallelism.

The rest of this paper is organized as follows. In Section 2, we analyze the economic and technical environment under which the new micros must exist, and the strengths and weaknesses of prior microprocessors and dataflow architectures with respect to this environment. In Section 3, we present the abstract architecture for *T, our proposed new processor architecture. In Section 4, we discuss the coding of SAXPY on this machine, and briefly discuss how it supports various popular parallel programming models. In Section 5, we describe a realization of this abstract model, which may be loosely described as a conventional microprocessor coupled with a "synchronization coprocessor". Finally, in Section 6, we compare it to other innovative contemporary processor designs, and outline our plans to build a *T machine.

2 Lessons from the past

2.1 The brave new world

A killer micro must be an excellent single-thread processor

To be cost-effective, the micros used in massively parallel machines should be commodity parts, *i.e.,* they should be the same micros as those used in workstations and personal computers. Market forces are such that a lot more design effort can be expended on a stock microprocessor than on a processor that is sold only in small quantities. In addition, there is a question of software cost. Parallel programs are often evolved from sequential programs, and will continue to use components that were developed for single-thread uniprocessors (such as transcendental function libraries, Unix, *etc.*). This does not mean that we are restricted

[1]The name *T is pronounced "start", after the principal "dataflow" instruction in the machine. The name can also be read as an acronym for "multi- (*) Threaded", a principal feature of the machine. One can also imagine the star to be representative of the meteorite shower that ended the dinosaur age.

to using good, conventional microprocessors in any parallel machine that we build. All it means is that any new processor that we design for multiprocessors must also stand on its own as a cheap and viable uniprocessor.

A killer micro must exploit parallel slackness

Parallel programs contain *synchronization* events. It is well known that processor utilization suffers if it busy-waits; to avoid this, some form of multiplexing amongst threads is necessary[2]. This is true even in uniprocessors.

In order to build parallel machines that are scalable both physically and economically, we must face the fact that inter-node *latency* in the machine will grow with machine size, at least by a factor of log(N), where N is the number of nodes in the machine. Thus, access to a non-local datum in a parallel machine may take tens to hundreds of cycles, or more. If we are to maintain effective utilization of the machine, a processor must perform some other useful work instead of idling during such a remote access. This requires that the processor be multiplexed amongst many threads, and that remote accesses must be performed as *split transactions*, *i.e.,* a request and its response should be treated as two separate communication events across the machine. If we follow this argument a step further, we see that a communication entering a node will arrive at some relatively unpredictable time, and that we need some means of identifying the thread that is waiting for this communication. This is, in fact, a synchronization event.

Thus, the following picture emerges. In a parallel machine, the way to deal with long inter-node latencies is exactly the way to deal with synchronization. A program must be compiled with sufficient *parallel slackness*[3] (or "excess parallelism") so that every processor has a pool of threads instead of a single thread, and some threads are always likely to be ready to run. Each processor must be able to multiplex itself efficiently amongst these threads. All communications should be split transactions, in which (a) an issuing processor does not block to await a response, and (b) a receiving processor can efficiently identify and enable the thread that awaits an incoming communication. For a more thorough explication of this argument, please refer to [6].

2.2 von Neumann microprocessors

Modern microprocessors are excellent single-thread processors, but they are not designed to exploit parallel slackness efficiently.

First, the cost of multiplexing amongst threads is high because of the enormous processor state that is associated with the currently executing thread. This state manifests itself in the register set, instruction and data caches, all of which may have to be reloaded with the new thread's context.

[2]We will use the term *thread* uniformly; other authors also use the terms *task* and *process*.
[3]The term is apparently due to Valiant [29].

Second, for a parallel environment, there is no efficient mechanism for naming, communicating and invoking continuations for split transactions to access remote locations. These can of course be simulated in software, but the frequency of these events suggests that architectural support can help significantly.

Third, many first-generation parallel machines, such as the Cosmic Cube [26], the Inmos Transputer and the Warp [2], had very poor interfaces to the interconnection network. There was a large software cost in handling incoming messages. This was further aggravated by the fact that messages trying to cross a node had to go through the node. Happily, many of the successors of these machines have solved this problem somewhat by devoting separate resources to message handling.

The net result is a high communication and synchronization cost. Programs can be written to use these machines effectively provided they minimize the occurrence of communication and synchronization events, and there are many success stories that do so. However, there is a high software cost associated with trying to structure programs to fit this model, and it is still a far cry from our goal of truly general purpose computing.

2.3 Dataflow architectures

Dataflow architectures have evolved substantially over the years (see [5, 4, 3, 13, 16, 18]). We will focus our comments on Monsoon [22, 23, 10] as the most recent representative of that evolution.

Dataflow architectures are excellent at exploiting parallel slackness. Indeed, this has always been a major underlying rationale for dataflow architectures. Parallel slackness is achieved by partitioning a program into extremely fine grain threads; in the pure dataflow model, each instruction is a separate thread. A thread descriptor is implemented as a *token*, which consists of three parts `(FP,IP,V)`, where:

- `FP` is a frame pointer, which points at a frame relative to which the instruction will be executed;
- `IP` is an instruction pointer, which points at code, and
- `V` is a a data value.

The pool of threads in a processor is manifest as a *token queue*. On each cycle, a token is extracted from the token queue, and the instruction that it refers to is executed by the processor relative to the frame that it points to. Every instruction explicitly names its successor instruction(s) (upto two successors). As a result of this execution, zero, one, or two successor tokens are produced, which are placed back in the token queue. Thus, a dataflow processor like Monsoon can multiplex between threads on every cycle.

Split transactions are performed thus: when a processor wishes to read a remote location `A`, it executes a `fetch` instruction. This causes a "read" token to be constructed and injected into the network. Suppose the `fetch` instruction names label `L` as its successor instruction. The corresponding read token contains the following information:

```
(READ, A, FP, L)
```

Once the read token is sent out, the processor continues to execute other tokens in its token queue. When the read token reaches the remote memory, the following token is sent back:

```
(FP, L, v)
```

This token is placed in the token queue to be executed just like any other token.

In addition, Monsoon also has an efficient mechanism to synchronize two threads. Two threads that must join will arrive at a common instruction that names a frame location which contains "presence bits", which can be regarded as a synchronization counter. On arrival, each thread causes the counter to decrement. When the first thread arrives, the counter does not reach its terminal value; the instruction is aborted and the processor move on to execute another token from the token queue. When the second thread arrives, the counter reaches its terminal value and the instruction is executed.

Thus, dataflow architectures (and Monsoon in particular) provide good support for exploiting parallel slackness— fine grain threads, efficient multiplexing, cheap synchronization, and support for split transactions to mask inter-node latency.

However, *present dataflow architectures do not have good single-thread performance* (for a more thorough discussion of this point, see [24]). The fundamental problem is that present dataflow architectures do not provide adequate control over the scheduling of threads. In the pure dataflow model, successive tokens executed by the processor may refer to arbitrarily different frames and instructions. The consequence is that an instruction can transmit values to its successors only *through (slow) memory*— it cannot exploit any special high speed storage such as registers and caches. In conventional uniprocessors, caches allow fast transmission of values because the successor instruction is executed immediately, while a previously stored value is still in the cache. This locality through successor-scheduling is absent in pure dataflow models. Pure dataflow models allow exactly *one* value to be transmitted without going to memory— the value on the token.

Monsoon improves on this situation. In Monsoon, an instruction can annotate one of its successors so that it is executed *directly*, *i.e.,* instead of placing the token back into the token queue, it is recirculated directly into the processor pipeline. Thus, in a chain of such direct successors, instructions can communicate values down the thread *via* high speed registers— no other thread can intervene to disturb the registers. However, Monsoon still has some engineering limitations that limit single-thread performance, namely, (a) very few registers (only three) and (b) the processor pipeline is eight cycles long, so that each instruction in a chain takes eight cycles.[4]

In Monsoon, control over scheduling stops at this point. A chain of direct successors is broken when it reaches an instruction that is a split transaction instruction (like a `load`), or when it reaches an instruction that executes a join that fails. At this point, there is no further control on the next thread to be executed. If we had such control, we might, for

[4]This is not to say that a single processor Monsoon will perform poorly. The eight-deep pipeline is also eight-way interleaved (like the HEP), so that code that is compiled to use all eight interleaves will utilize the machine well. The point here is that a *single-threaded* program will not perform competitively. Again, please refer to [24] for more discussion of this point.

example, choose another thread from the same frame, to maintain locality with respect to the current frame.

A final architectural point: in Monsoon, synchronization memory is disjoint from data memory, but they have to be allocated in tandem (every frame location has a data cell and presence bits, so a single offset names both data and synchronization bits). Thus, the compiler does not have complete flexibility in partitioning the space in a frame into synchronization and data areas.

2.3.1 Lessons from the past: the bottom line

In *T, we aim to provide the best of both worlds, and go further. We will provide the fast single-thread execution of conventional micros, coupled with the facilities to exploit parallel slackness from dataflow architectures. In addition, we will provide tight control over the scheduling of threads.

3 *T: the abstract model

In this section, we describe the *T abstract machine, *i.e.,* the model of *T used by compilers (or assembly-level programmers).[5] The *T abstract processor architecture is a proper superset of a conventional RISC processor, with a few extra instructions that are precisely the "dataflow" instructions that address the requirements of a parallel machine. We begin, in Section 3.1, by describing our view of *frames* as the basis for locality and synchronization. In Section 3.2 we describe the architecture of the nodes of the parallel machine. In the next section (Section 4), we describe the code for SAXPY, and briefly describe how various parallel programming models (shared memory, SPMD/SIMD, *etc.*) can be easily expressed in our model.

3.1 Frames as the basis for locality

In most languages, when a procedure is invoked,

- a frame (also known as an activation record) must be allocated for it;
- arguments (if any) must be deposited in its frame, and
- execution of its code must be initiated.

When it terminates, it passes results to the frame of its continuation, and initiates computation there (usually, this is its caller).

In a parallel system, a procedure may invoke several other code blocks in parallel. Further, iterations of a loop may also be invoked in parallel and be distributed across the nodes of the machine. Where previously a loop ran in a single frame, we may now have to allocate separate frames for each iteration or group of iterations. In general, instead of a stack of frames, we now have a tree of frames.

[5] Previous versions of this abstract machine were presented in [21] and [20].

Figure 1 depicts the runtime structures of a typical parallel program. On the left, we have the tree of *frames*. On the right, we have a *heap* containing global data structures, which we call *objects*. Of course, frames may refer to global data structures. In addition, for object-based languages (such as Lisp, Id and other functional languages), objects and frames may also refer to each other.

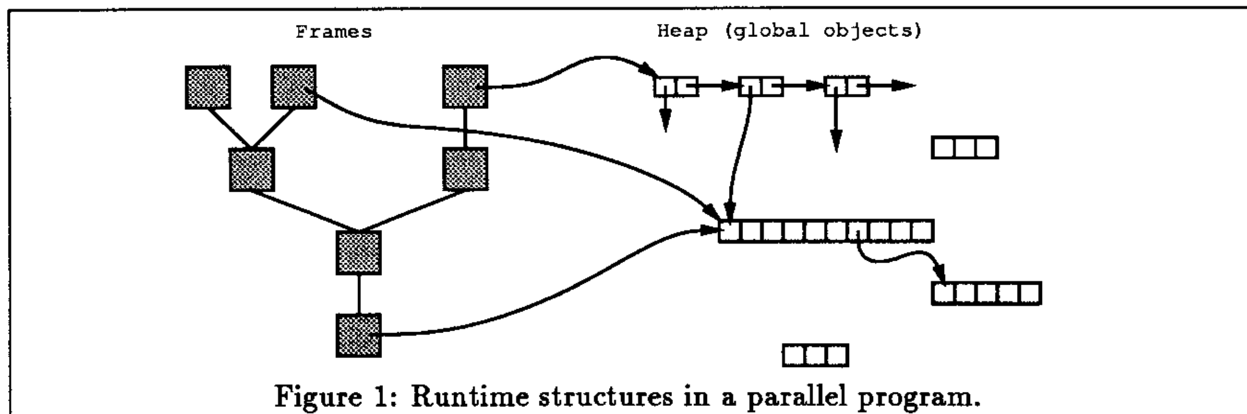


Figure 1: Runtime structures in a parallel program.

Because frames may now correspond both to procedure invocations and to loop iterations, we prefer to use the term *code block* for the segment of code that is the unit of invocation.

We are going to use frames as the basis for locality. Frames may be distributed among the nodes of the parallel machine, but *each frame must reside entirely within a single node*. There is no such restriction on global data structures— a single object may span several nodes of the machine. For each frame in a node, the corresponding code block must also be present in that node. This means that if a code block is invoked in several nodes, copies of that code block must exist in all those nodes. A simple way to achieve this is to copy all code into all nodes, but code blocks could also be loaded dynamically on demand. The key point is that a particular invocation of a code block can always access its frame locations using local memory operations. Accessing locations in other frames or in global objects, however, may involve communication. This will be reflected in the instruction set of the processor in each node.

3.2 The *T node architecture

The *T abstract model for a node in a parallel machine is shown in Figure 2. Although the memory of the machine is physically distributed amongst all the nodes, we assume a single global address space, *i.e.,* the local memory in a node implements a piece of a single address space.

The Data Processor 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 to contain a pointer to the "current frame" (which is always in its local memory). Being a conventional RISC processor, the Data Processor is optimized to run long, sequential threads efficiently. It obtains the

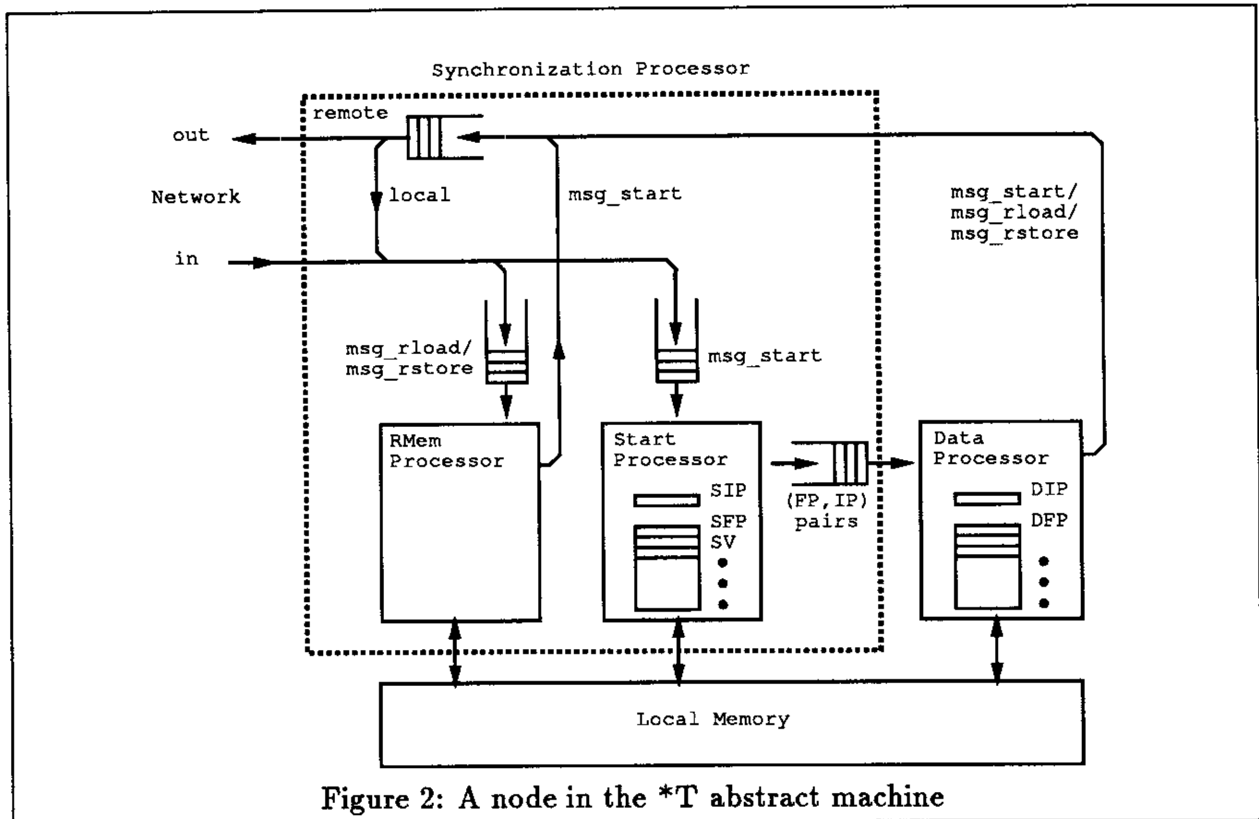


Figure 2: A node in the *T abstract machine

starting points of these threads from the Start Processor. Each thread that it obtains 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 Start Processor, the Data Processor simply waits until a thread becomes available. Clearly, for good utilization, this situation should be avoided.

In addition to its conventional RISC instructions, the Data Processor can execute a few additional "dataflow instructions" whose effect is to 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.

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

msg_op arg1, *arg2*, ...

Arg1 is always a global address that identifies a unique destination node in the parallel machine. The message is automatically routed there. Of course, messages to the current node are short-circuited back directly. Broadly speaking, *msg_ops* fall into two categories: "start" messages and "remote memory" messages. When a message arrives at a node, it is passed either to the Start Processor or to the RMem Processor based on the category of its *msg_op*.

The Start Processor has a program counter called SIP ("Start processor Instruction Pointer"), two special registers SFP and SV and, perhaps, other general purpose registers. The Start Processor is triggered by the arrival of a start message (it simply waits, if there is no start message available). When it picks up a start message, its SIP, SFP and SV registers are loaded with values from the message, after which it begins executing instructions from the address in SIP. It can read and write local memory, and it can post new (FP,L_D) pairs to be picked up by the Data Processor.

The RMem Processor processes requests to read and write this node's memory, and responds with start messages.

Because both the Start Processor and the Data Processor may execute instructions, we will distinguish labels for the two processors by the subscripts S and D, respectively (*e.g.*, L_S, L_D, M_S, ...).

We refer to the components within the dashed lines in Figure 2 collectively as a "Synchronization Processor". This is precisely our dataflow complement to a conventional uniprocessor (outside the dashed lines).

3.2.1 Threads

The instruction set of the Data Processor is a proper superset of a conventional RISC instruction set, so we will assume the reader is familiar with conventional arithmetic-logic and comparison instructions, unconditional and conditional jumps, *etc.* We will focus here only on the novel, dataflow instructions that have to do with threads and synchronization. While reading the descriptions below, please refer to Figure 3 for an overview of the thread-related instructions and messages.

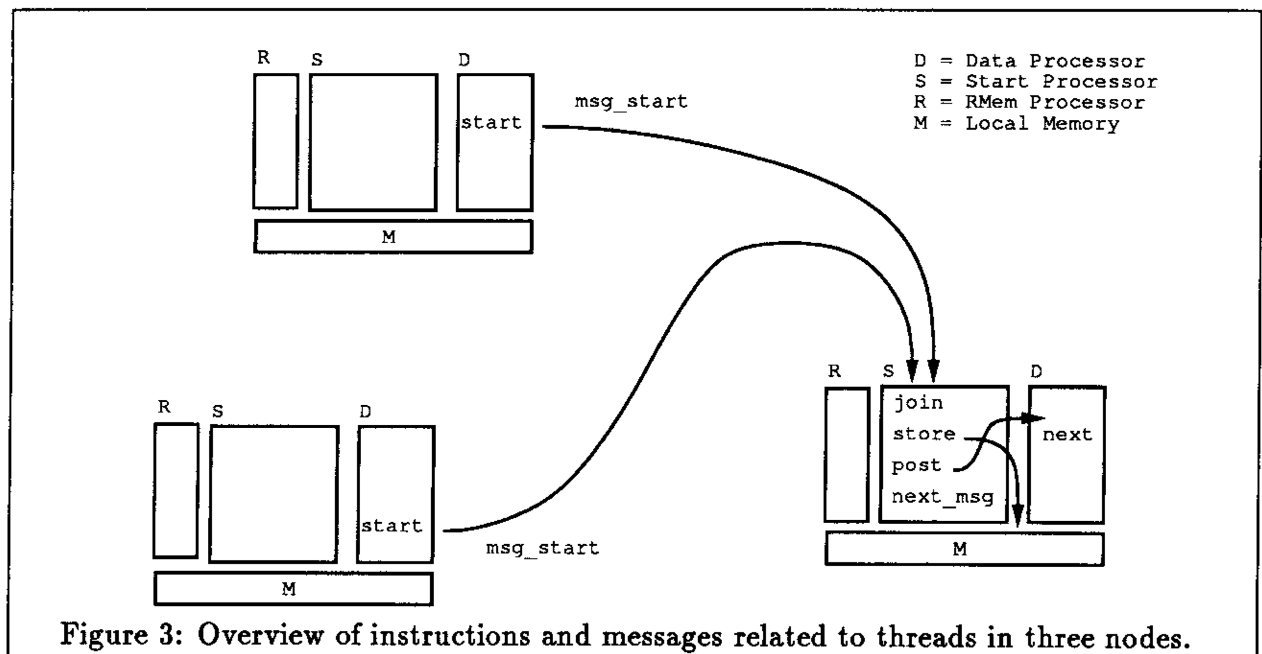


Figure 3: Overview of instructions and messages related to threads in three nodes.

Forking a new thread

A common situation where we wish to start a new thread is when one code block F calls another code block G. For example, we may wish to transport an argument from F's frame to G's frame and to initiate a thread in G that will compute with it. Similarly, we may wish to transport a result back from G's frame to F's frame and to initiate a thread in F to compute with it. In general, these frames may be on different nodes, so we need an explicit communication to perform these actions. For this, we use a **start** instruction, which has three register arguments:

Data Processor Instruction:	`start rF,rI,rV`
Semantics:	Let `FP` = `Register[rF]` Let L_S = `Register[rI]` Let `V` = `Register[rV]` Send message: `msg_start FP,`L_S`,V`

Note that **start** is effectively a fork, since the Data Processor continues to execute at the next instruction after it has initiated the message send. Note also that this is only the first half of a one-way communication, *i.e.,* the **start** instruction only emits a **msg_start** message. In other words, **start** is a non-blocking send. The instruction pointer L_S on the start message is a label for a Start Processor, not Data Processor.

Readers familiar with dataflow literature will recognize that the contents of a **msg_start** message correspond exactly to a classical dataflow "token"— `FP` is the "context", L_S is the "statement" and, of course, `V` is the value.

(Please see Appendix A for a description of the **fork** instruction, which is a special case of the **start** instruction in which the destination frame is the same as the current frame and no data value is transmitted.)

At this point, it is worth making some observations that contrast the **start** instruction with other models of forking threads.

- The **start** instruction does not involve any resource allocation. In many other fork models, a fork involves the dynamic allocation of a new stack. In our model, dynamic resource allocation is separated out into a completely orthogonal issue, and the **start** instruction is very cheap— it just sends a simple message.
- In many fork models, each fork is a sequential thread associated with a stack with possibly multiple frames. In our model, every frame can have multiple threads active in it. In fact, there is no limit to the number of threads active within a frame.

Ending a Thread

The Data Processor can terminate a thread and begin executing a new one by executing a **next** instruction:

Data Processor Instruction:	`next`
Semantics:	A new frame pointer `FP` and a new instruction pointer L_D are loaded from the Start Processor into the Data Processor into its `DFP` and `DIP` registers.

The Data Processor thus continues fetching and executing instructions from L_D.

The Start Processor, and Synchronization

The Start Processor may be thought of as a transaction processor: it dequeues a start message, does some processing, and is then ready to dequeue the next start message. Incoming messages have the form:

```
msg_start FP,Ls,V
```

In response to such a message, `FP`, L_S, and `V` are loaded into the Start Processor's `SFP`, `SIP` and `SV` registers, respectively, after which it begins executing instructions at L_S. The Start Processor may, of course, have a general instruction set, but we focus here on the instructions that it needs to interact harmoniously with the Data Processor.

The following instruction allows the Start Processor to store the value on the incoming start message into the destination frame at offset `X`:

Start Processor Instruction:	`store SFP[X],SV`
Semantics:	Let `A` = `Register[SFP]` + `X` `Memory[A] := Register[SV]`

The following instruction allows the Start Processor to cause the Data Processor to begin executing at L_D with respect to frame `FP`:

Start Processor Instruction:	`post rF,rI`
Semantics:	Let `FP` = `Register[rF]` Let L_D = `Register[rI]` Post (`FP`,L_D) to be picked up by the Data Processor

The following instruction allows the Start Processor to start processing the next message:

Start Processor Instruction:	`next_msg`
Semantics:	Reload `SFP`, `SIP` and `SV` from the next incoming `msg_start` message

Here is a typical code sequence that executes as a result of a start message that loads label L_S into `SIP`:

```
Ls:
      store SFP[X],SV      -- store incoming value into frame offset X
      post SFP, LD         -- enable thread LD with this frame in Data Processor
      next_msg             -- done; handle next message
```

Synchronization is performed in the Start Processor using synchronization counters in the frames. For example, suppose node N1 sends two arguments to a frame in node N2, using the following two messages:

```
msg_start FPx,L_S,V1
msg_start FPx,M_S,V2
```

On arrival of each message, the corresponding values are stored in the frame at offsets `X1` and `X2`, respectively. Then, a counter at offset `C` in the frame is incremented and compared with the constant 2 (we assume the counter was previously initialized to 0). The two messages may be processed in any order; the first message will find the counter equal to 1, and will go to process the next message. The second message will find the counter equal to 2 and will post `(SFP,L_D)` to the Data Processor. Here is the code:

```
L_S:
    store SFP[X1],SV       -- store incoming value into frame offset X1
    load R0,SFP[C]         -- load counter from frame offset C
    incr R0                -- increment it
    store R0,SFP[C]        -- store it back
    cmp R0,2,RB            -- compare counter value to 2
    jeq RB,N_S             -- if equal, goto N_S
    next_msg               -- else die; handle next message

M_S:
    store SFP[X2],V        -- store incoming value into frame offset X2
    load R0,SFP[C]         -- ...
    incr R0                -- ...
    store R0,SFP[C]        -- ... same as above ...
    cmp R0,2,RB            -- ...
    jeq RB,N_S             -- ...
    next_msg               -- ...

N_S:
    post SFP,L_D           -- When both messages handled, enable L_D in Data Processor
    next_msg               -- with this frame
```

Since we want to allow this kind of synchronization to happen very frequently, we abbreviate the load-increment-store-compare-and-conditionally-post sequence into a single instruction:

Start Processor Instruction:	`join rC,`tc`,rI`
Semantics:	Let `XC` = `Register[rC]`
	Let $\texttt{L}_D$ = `Register[rI]`
	`Memory[XC] := Memory[XC] + 1`
	If `Memory[XC]` = tc then post `(SFP,L_D)`

Readers familiar with dataflow literature will recognize that the input queue of start messages for the Start Processor is precisely the "token queue" of dataflow architectures.

3.2.2 Global data accesses

A Data Processor in one node can access data in a remote node using using remote load and store instructions which move the data to and from the current frame. Such instructions are implemented using split transactions. Once data has been brought into the current frame, it can be manipulated by the Data Processor using conventional instructions. While reading the descriptions below, please refer to Figure 4 for an overview of instructions and messages related to global data accesses.

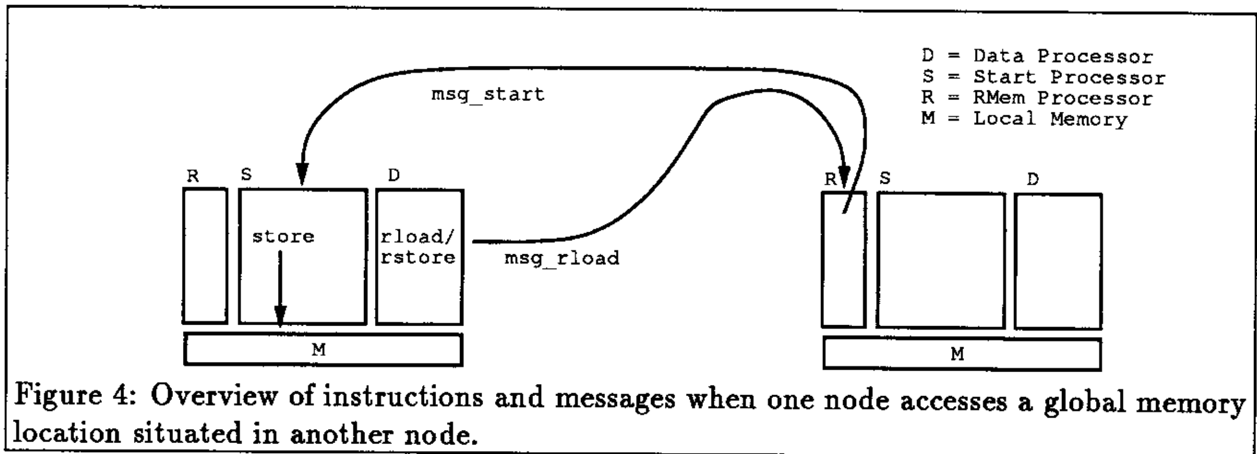


Figure 4: Overview of instructions and messages when one node accesses a global memory location situated in another node.

A remote load instruction sends a message:

Data Processor Instruction:	`rload rA, rI`
Semantics:	Let `A` = `Register[rA]`
	Let L_S = `Register[rI]`
	Let `FP` = `Register[DFP]`
	Send message: `msg_rload A,FP,`L_S

The destination node is implicit in the global address `A`, which is used to route the message. When the message arrives at the remote node, it is handled by the RMem Processor of that node:

RMem Message:	`msg_rload A,FP,`L_S
Semantics:	Let `V` = `Memory[A]`
	Send message: `msg_start FP,`L_S`,V`

We have already seen that the `msg_start` message is routed to the node specified by the address `FP`, and thus it returns to the node that issued the `rload`. There, the code at L_S will store the value `V` into the frame `FP`, and typically a thread (`FP`,L_D) in the Data Processor will be enabled to compute with it.

Note that the `rload` instruction is also a fork— it simply initiates the load and continues executing at the next instruction. Thus, it is possible to initiate many remote loads before receiving any reply. Further, the `msg_start` messages may return in any order— they carry enough information on them to know how to process them as they arrive.

Remote stores are similar. The remote store instruction initiates a store message:

Data Processor Instruction:	`rstore rA,rV,rI`
Semantics:	Let `A` = `Register[rA]`
	Let `V` = `Register[rV]`
	Let L_S = `Register[rI]`
	Let `FP` = `Register[DFP]`
	Send message: `rstore A,V,FP,`L_S

The message is routed to the node identified by the global address `A`. There, it is handled by the RMem Processor:

RMem Message:	`rstore A,V,FP,L`$_S$
Semantics:	`Memory[A] := V`
	Send message: `msg_start FP,`L_S`,foo`

Again, note that the `rstore` instruction is also a fork— it simply initiates the `rstore` and continues executing at the next instruction. Later, an acknowledgement comes back to (FP,L_S) (`foo` is an arbitrary value). The acknowledgement may be used to ensure serial consistency— the code at (FP,L_S) executes under a guarantee that the store has completed.

Global data accesses with data level synchronization

`Rload`'s and `rstore`'s are just the basic two remote memory operations. It is desirable to extend the repertoire beyond this in order to implement data level synchronization. The extensions are described in more detail in Appendix A, but we give a brief description here.

With each global location that is used with data level synchronization, we associate some extra bits called "presence bits". Two of the states encoded in these bits are called "full" and "empty".

The `rIload` and `rIstore` instructions in the Data Processor have the same instruction formats as `rload` and `rstore`, respectively, and they generate similar remote memory messages with `msg_rIload` and `msg_rIstore` opcodes. A `msg_rIload` arriving at a full location behaves just like a `msg_rload`. Arriving at an empty location, it is deferred (*i.e.,* queued) at that location. The response is sent later, when a corresponding `msg_Istore` arrives, which also deposits a value in the location and marks it full. These operations allow implementation of "I-structure" operations (see [8]) which are useful to implement producer-consumer parallelism.

The `rtake` and `rput` instructions in the Data Processor have the same instruction formats as `rload` and `rstore`, respectively, and they generate similar remote memory messages with `msg_rtake` and `msg_rput` opcodes. A `msg_rtake` arriving at a full location returns the value just like an `msg_rload`, but it also marks the location empty. Arriving at an empty location, it is deferred just like a `msg_iload`. A `msg_rput` arriving at a location with no deferred `msg_rtake`'s behaves just like a `msg_rIstore`, marking the location full. If there are deferred readers, one reader is dequeued and the value is sent to it. These operations allow implementation of atomic updates on remote locations (such as shared counters, shared queues, *etc.*).

Readers familiar with dataflow literature will recognize that if we omit the Start Processor and Data Processor in a node, leaving only the RMem Processor, the local memory and the interface to the network, the remaining node is precisely an "I-structure Memory" module.

3.2.3 Inter-thread and inter-frame scheduling control for better cacheing

So far, we have taken a simplistic view of the `post` instruction in the Start Processor, which posts a new (FP,L_D) pair to be picked up by the Data Processor when it executes a `next` instruction. Figure 2 suggests that the interface is simply a FIFO queue. By being more sophisticated about this queue, we can improve locality in the Data Processor, thereby

improving the behavior of any cache that resides between the Data Processor and the local memory.

The Start Processor can sort the `(FP,L_D)` pairs according to `FP`. In other words, for each frame in the current node, it maintains the collection of IPs for that frame. There are various ways to implement this— as a separate table mapping FPs to collections of IPs; as a list of IPs hanging off each frame, or directly as an array within each frame. The exact representation is unimportant, provided the Start Processor can access it. In fact, the responsibility for managing these structures may be shared between the Start and the Data Processors.

Now, the Start Processor can post `(FP,L_D)` threads to the Data Processor according to a priority scheduling policy. For example, it can give priority to threads that belong to the Data Processor's current frame. This is, in fact, exactly the scheduling policy advocated by Nikhil in his P-RISC compiler [20] and by Culler in his Threaded Abstract Machine [11]. The current frame is thus treated as a "hot frame" where activity is currently focused. To implement this, the Start Processor needs to know what is the current contents of `DFP` in the Data Processor. This is quite easy to implement.

A generalization of this principle of hot frames is to maintain a *set* of hot frames rather than a single hot frame. One can imagine a small "registry" of hot frames (with, say, 16 entries), with threads from this set given priority. Registry of frames into this hot set can be performed either automatically or under explicit software control. In Section 5, we describe one proposal for such a frame registry.

4 An example: SAXPY

In this section, we demonstrate programming of *T by presenting hand-compiled code for SAXPY. We will address a number of issues, such as the use of split transactions to mask long latencies, the use of registers, *etc.* Finally, in Section 4.2, we show how *T can easily support various popular parallel programming models such as shared memory, SPMD/SIMD, object-oriented programming, *etc.*

4.1 *T code for SAXPY

SAXPY is the inner loop of the Linpack benchmark. Here is the SAXPY inner loop:

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

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

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

Uniprocessor code

*T uniprocessor code for the SAXPY loop is shown below. We use names beginning with "r" as symbolic names for general purpose registers.

```
      load  DFP[XP],    rXP       -- load ptr to X
      load  DFP[YP],    rYP       -- load ptr to Y
      load  DFP[A],     rA        -- load loop constant: a
      load  DFP[YLim], rYLim      -- load loop constant: Y pointer limit
  LOOP:
      cmp   rYLim,rYP,rB         -- compare ptr to Y with limit
      jgt   rB, OUT              -- jump out of loop if greater

      load  rXP,      rXI        -- load X[i] into rXI          (L1)
      load  rYP,      rYI        -- load Y[i] into rYI          (L2)
      add   1,rXP,rXP            -- increment ptr to X
      mult  rA,rXI,r1            -- a*X[i]
      add   r1,rYI,r2            -- a*X[i] + Y[i]
      store rYP, r2              -- store into Y[i]             (S1)
      add   1,rYP,rYP            -- increment ptr to Y
      jump  LOOP

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

This code runs entirely in the Data Processor; in a uniprocessor, the Start Processor and the RMem processor are ignored completely. (If $N \geq 1$ we could improve it by moving the conditional jump to the bottom of the loop so that there would only be one jump per iteration.)

Let us consider what happens when memory latency increases dramatically, as in a multiprocessor. Each of the two loads *L1* and *L2* would be directly affected[6] Normally, the processor would stall at each of these instructions and performance would degrade. We might think of alleviating this problem using some kind of cacheing to decrease memory latency; however, the construction of coherent caches across the distributed memory of a large parallel machine is still an open problem.

Using rload's to mask latency

Some modern processors use "delayed loads" to accommodate memory latency. Executing a delayed load can be viewed as forking off a memory request and continuing at the next instruction, followed by a join a few instructions downstream. Thus, memory latency is masked by some local parallelism.

The `rload` mechanism in *T can be viewed as a generalization of this idea. 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 Start Processor, deposits the value into the frame, and tries to `join` with the other response at frame location `c1`. When the

[6] The store *S1* would also be affected, but we focus on the loads because we assume that we do not have to wait until the store has completed before proceeding to the next instruction.

join succeeds, the Start 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. Meanwhile, the rstore acknowledgments all arrive at the Start Processor and join at frame location c2. The continuation of this join is the loop sequel. Here is the augmented frame layout:

	...
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
	...

The code for this new version is shown in Figure 5. The Data Processor code is shown at left, and the Start Processor code is shown at right. We have drawn boxes to assist the reader in recognizing the structure of the code, and shown arrows corresponding to interactions between the Data Processor and Start Processor. For brevity, we have not shown arrows for jumps and conditional jumps in either processor. Note that join counters c1 and c2 are initialized by the Data Processor before the loop, and that c1 is reset to zero by the Data Processor at Ld.

An unpleasant consequence of our new organization is that 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 rload's, the registers may have changed by the time we get the Data Processor back at label Ld. 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. Further, where previously data moved directly from X[i] and Y[i] into registers rXI and rYI, they now move first to XI and YI in the frame, from where they have to be explicitly loaded into registers.

Avoiding the extra load's and store's

The above code assumed the worst— that the rload latency is so long that the Data Processor must be relinquished to another thread. Instead, the Data Processor can peek at the join counter c1 and choose to continue if both the rload responses have arrived. If successful, registers do not have to be saved and restored from the frame. The modified code is shown in Figure 6, with the register save and restore code shown in dashed boxes.

Here, as in our uniprocessor version, we preload the X pointer, Y pointer, A and YLim into registers before the loop. As in the second version, we also initialize the join counters c1 and c2 before the loop. Inside the loop, after issuing the two rload's, we peek at the join counter c1. If it is equal to 2, we jump directly to L2d. If not, we save the X and Y pointers in their frame locations, and start L3s in the Start Processor.

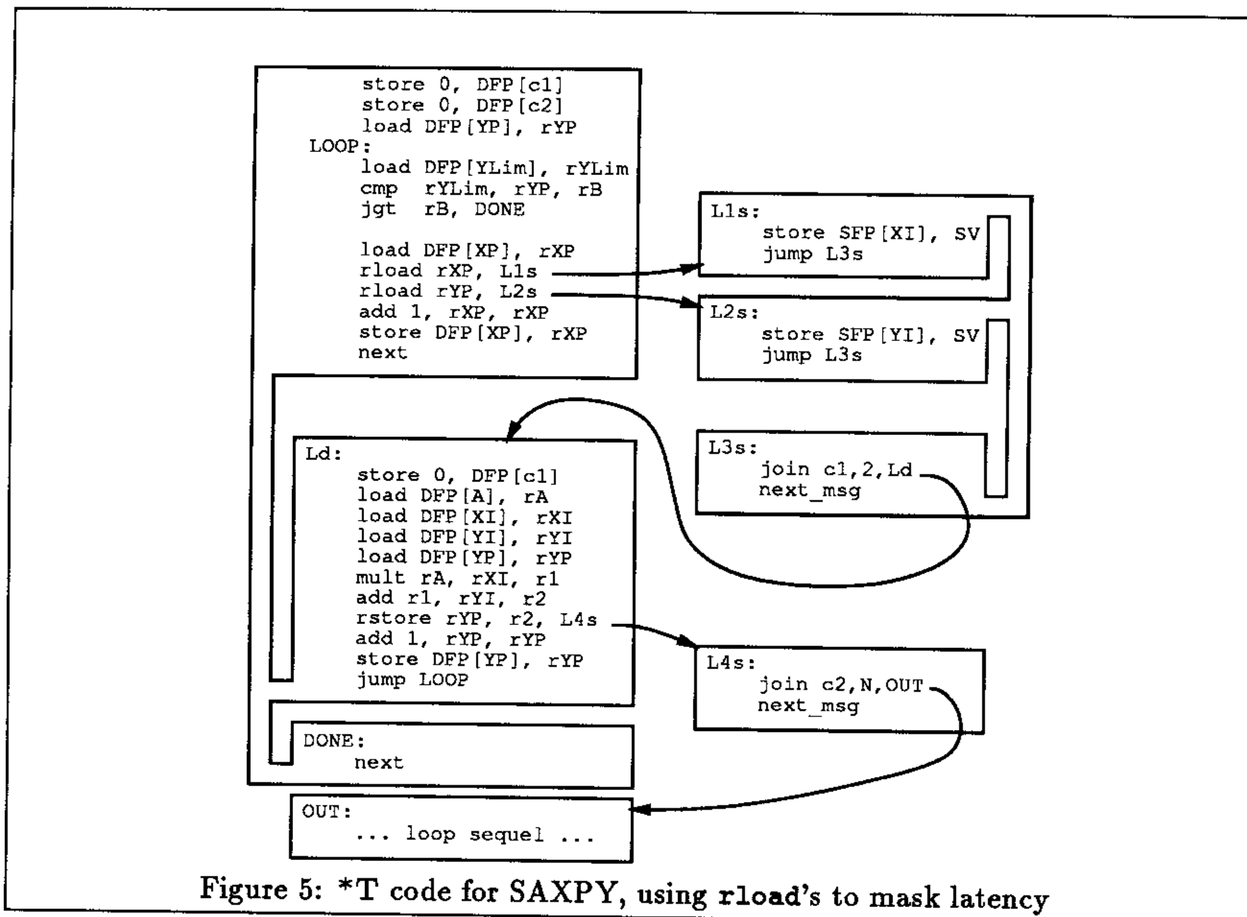


Figure 5: *T code for SAXPY, using `rload`'s to mask latency

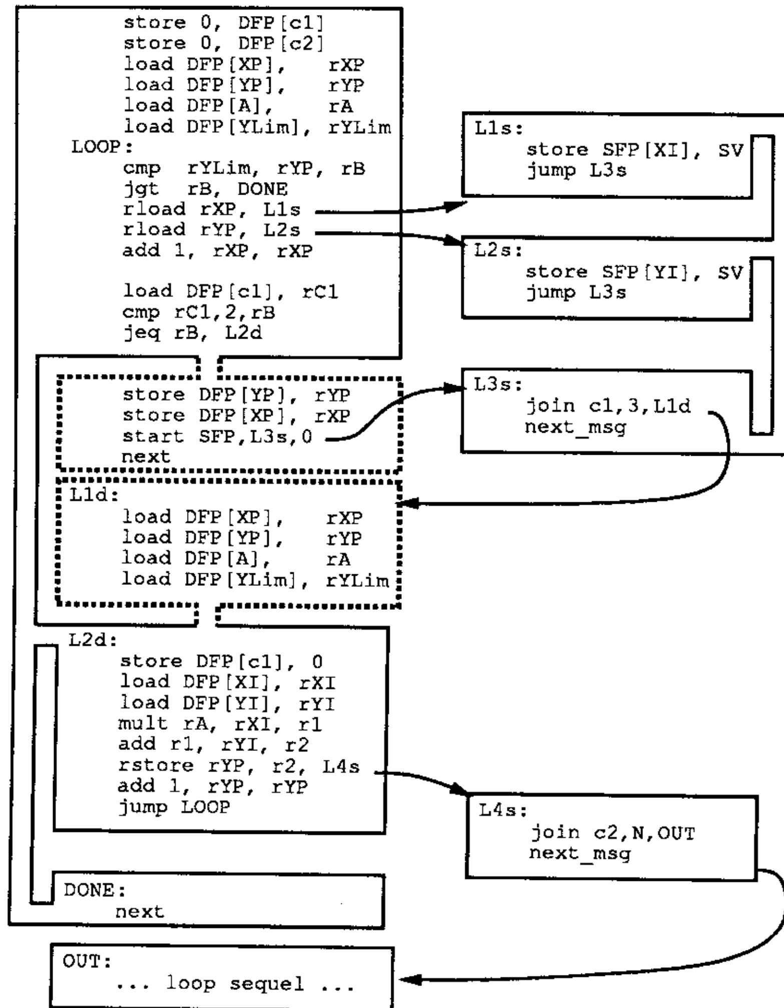


Figure 6: *T code for SAXPY, using **rload**'s to mask latency, but improved to avoid relinquishing the Data Processor when possible.

The two rload responses return to L1s and L2s, respectively, in the Start Processor. There they are saved in frame locations XI and YI, respectively, and execution continues at L3s. If the two responses arrive quickly, they both die at the join, and the Data Processor will successfully see a value of 2 for c1. Otherwise, the join is executed thrice, and this enables L1d in the Data Processor.

In the Data Processor, the code at L1d is executed only if the Data Processor had been relinquished. Thus, it reloads the X and Y pointers, and the loop constants A and XLim into registers, and falls through to L2d. The code at L2d is assured that its registers are ok. Thus, it does not have to reload the pointers or loop constants. Similarly, after incrementing the X pointer rXP, it does not have to save it to the frame.

Note that if control of the Data Processor has to be relinquished (because the rloads took too long), this version of the program will do worse than our original version. In practice, it may be necessary to fill the post-rload slots with more instructions to give them time to complete.

Further optimization

So far, we have concentrated on trying to improve a single iteration of the loop. However, there are still moments when the processor to memory pipeline lies idle, *e.g.,* after the two loads have completed. If this is a problem, one can play a variety of other tricks to alleviate it.

The loop may be unrolled so that we perform the i'th and the $i+1$'th computation in a single iteration, and increment i by two on each iteration. In this case, we could issue four rload's in parallel instead of two, and two rstore's in parallel instead of one. Of course, the loop may be unrolled by a larger factor than 2.

The loop may be split into two parallel loops, with each loop computing on half of the arrays, using the following outline:

```
    start DFP,L1s                     Ls1: post SFP,L2d
                                           next_msg
L1d:
    ... loop 1 ...
    next

L2d:
    ... loop 2 ...
    next

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

The start instruction has the ultimate effect of forking L1d and L2d in parallel. In this way, it is possible to compute in loop 1 while loop 2 is waiting for its rload's to complete, and vice versa, thus improving overall processor utilization.

4.2 Compiling for *T

The *T abstract machine model is suitable as a target for several programming models. We discuss this briefly here, with a more detailed treatment to follow in a subsequent paper.

First, we repeat some salient points about locality. It is assumed that only accesses to the current frame are local. An instruction can only manipulate the current frame directly— all other accesses are performed using instructions that initiate split transactions. In a parallel machine with distributed memory:

- Each frame must reside entirely within a single node's memory.
- Frames may be distributed among the nodes of a machine.
- A frame for code block A can only be allocated in a node that contains the code block A. If code block A is replicated on all nodes, frames for that code block can be allocated on any node.
- Global objects may be allocated on any node, and a single object may straddle many nodes.
- If the compiler can predict that other frames and objects are local (*e.g.,* through static allocation or through directives to the runtime allocator), it can replace `rload`'s and `rstore`'s with ordinary loads and stores, and replace `rstart`'s with ordinary procedure calls.
- Registers are assumed to contain only *thread-local* data. If an instruction defines a particular register, a later instruction in the same invocation of the same thread may use it. However, nothing may be assumed about the contents of the registers at the start of a thread, since it is unpredictable as to which thread ran prior to the current one. Any data that must be communicated from one thread to another must be passed through the frame. We also say that registers are "volatile" or "ephemeral" across threads and that frame locations are "persistent" across threads.

Id and functional languages

Implementing Id and functional languages on *T is no more difficult than implementing it on existing architectures. The *T abstract machine model is, in fact, the latest step in a long line of development of compilers and architectures for functional languages. The interested reader is referred to [7], [28] and [20] for details.

This does not mean imply that all problems in compiling non-strict functional languages have been solved. For example, detecting long threads, storage reuse, load balancing, *etc.* remain challenging problems.

Shared memory models

If we ignore the locality model that we have carefully built up in *T abstract machine, it is no more than a conventional, shared memory parallel machine. The tree of frames by which we describe our runtime structures properly subsumes the "cactus stack" model that is popular in implementations of parallel FORTRAN, C, and similar languages. The `rtake` and `rput` instructions may be used to implement semaphores for traditional synchronization mechanisms.

SPMD models

To date, the SPMD (Single Program Multiple Data) model and the less general SIMD model are the only successful model used to program large parallel machines. SPMD programs are usually run on so-called "multicomputers". Mapping an SPMD program to the *T abstract machine is quite easy.

The tree of frames has an especially trivial structure: the root frame has N children frames, where N is the number of nodes in the SPMD program. Each of these children then has a linear chain of descendants, *i.e.*, each chain acts as a stack. The j'th stack is allocated on the j'th node. In other words, the root frame is simply used to fork off N invocations of the same subprogram, with the j'th subprogram running entirely on the j'th node. Each subprogram is an ordinary sequential program.

The Global Data area is partitioned into exactly N equal-sized areas, and global data is statically mapped into these areas. The j'th global data area is considered to be local to the j'th node. This mapping is static, so that the compiler uses ordinary load and store instructions to access local data.

In the SPMD model, there is no direct access from the i'th node to data in the j'th node, where $i \neq j$. Instead, the program in the i'th node communicates with program in the j'th node using `send` and `receive` primitives. This is modelled easily in the *T abstract machine as follows: A part of the global data area in each node is reserved for a message queue, representing messages entering that node. A source-level `send` command from any node to node j is compiled by appending a reference to the message data structure into the j'th message queue. Implementing such shared message queues is easily done, using the `rput` and `rget` instructions. A source-level `receive` command in the j'th node is compiled by dequeing a reference to a message data structure its local message queue, and then using `rload`'s to fetch the contents of the message.

5 An Implementation of *T

The *T abstract model creates a clean separation between the execution of instructions within a thread (the data processor) and the processing of network messages and the scheduling of threads (the synchronization processor). Not surprisingly, this permits a realization wherein the data processor is a conventional RISC and the synchronization processor is a specialized function which behaves much like a memory-mapped coprocessor to the RISC.

5.1 A *T node based on M88110

We now present a concrete *T realization centered around the Motorola M88110, a highly integrated superscalar RISC microprocessor.[7] As illustrated in Figure 7, a *T **node** comprises an unmodified M88110, a memory-mapped synchronization coprocessor/network interface, second level cache, and Local DRAM. The node is fully backward compatible such that the synchronization coprocesser is completely transparent to normal M88110 programs (*e.g.* UNIX). In terms of hardware protocol, the synchronization processor will act just like another M88110 sharing the local bus.

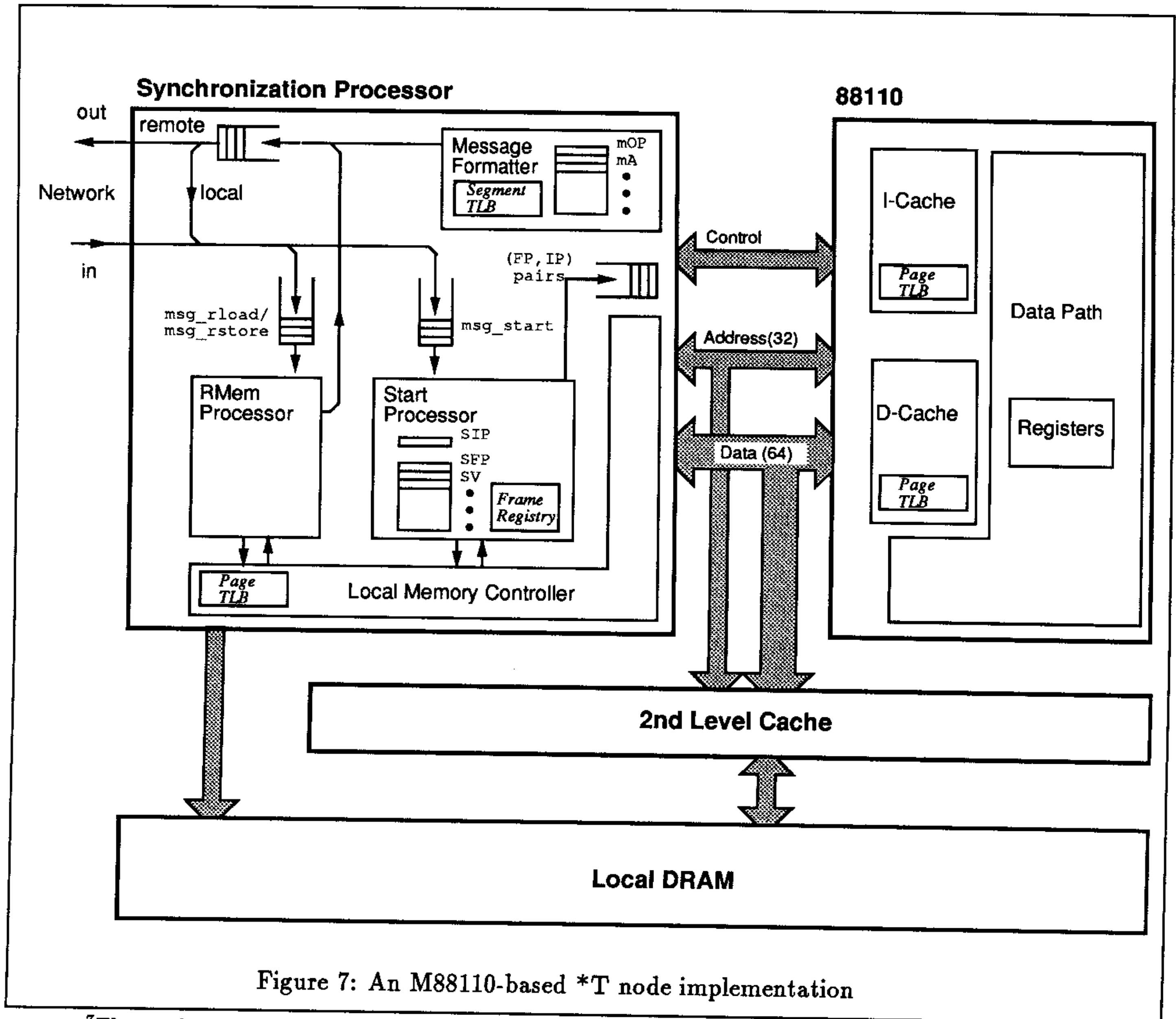


Figure 7: An M88110-based *T node implementation

[7] The synchronization coprocessor documented here is easily adapted to any microprocessor which supports cache coherence.

The synchronization processor comprises four distinct subfunctions:

- **Message Formatter.** The message formatter maintains a set of of memory-mapped registers that, when written to by an executing M88110 thread, causes the creation and transmission of `msg_rload`, `msg_rstore`, and `msg_start` messages. For remote loads and stores, the message formatter also includes segment translation hardware for translating 64-bit global virtual addresses into a destination node number and 32-bit local virtual address on the destination node.

- **RMem Processor.** The RMem processor services `msg_rload` and `msg_rstore` requests for global memory locations which map onto the current node. The RMem processor supports imperative and many styles of synchronizing data accesses (*e.g.*, I-structures, M-Structures).

- **Start Processor.** The start processor services all `msg_start` messages directed to the current node. The start processor also implements the hardware layer of the thread scheduler: queues of posted threads corresponding to a subset of "hot" frames as directed by a *frame registry.*

- **Local Memory Controller.** The local memory controller supports access to locations in local virtual address space. The controller performs page translation to map the local virtual addresses into physical addresses[8], and also provides for block transfers between DRAM and the second level cache.

The synchronization and data processors intercommunicate in two ways. First, registers and queues implemented by the synchronization processor are memory mapped in the M88110 physical address space. For example, the M88110 will execute a `next` instruction by reading the head of the thread queue that is filled by the start processor. Second, the processors share the same virtual address space and may read and write shared variables. For example, the start processor will write the value portion of `start` messages in activation frame locations which are subsequently read by a `posted` thread. All of this communication takes place over the 64-bit local data bus. Synchronization processor registers and queues are directly read and written over this bus, while shared variables are, in general, found in the second level cache or DRAM.

In the remainder of this section we will first describe a scheme for global addressing which is essential to understand the functioning of the Message-Formatter. It is followed by a description of executing dataflow instructions on a stock M88110. In Subsection 5.4 we present some other implementation issues in the design of the Synchronization processor.

5.2 Global Addresses and Virtual Memory

The message formatter maintains a set of of memory-mapped registers that, when written to by an executing M88110 thread, causes the creation and transmission of `msg_rload`, `msg_rstore`,

[8] All caches are keyed to physical addresses.

and `msg_start` messages. For remote loads and stores, the message formatter also includes segment translation hardware for translating 64-bit global virtual addresses into a destination node number and 32-bit local virtual address on the destination node.

The M88110 supports 32-bit byte-addresses, yielding a four gigabyte address space. Consider, however, a possible machine configuration comprising 4K nodes (2^{12}) with eight megabytes (2^{23}) of local memory per node. This is 128 gigabytes (2^{37}) in *physical* memory alone. Clearly, we require global addresses which are considerably larger than 32 bits. Our proposal for supporting a 64-bit global address space while still retaining efficient, and compatible, local addressing is based upon segmentation. Consider the following:

- A *Local Virtual Address (LVA)* is a 48-bit quantity,

  $$\text{LVA} = n_{16} : v_{32}$$

  where the v is a virtual address on node number n. All local memory references made by a processor (*e.g.*, a normal M88110 load or store instruction) implicitly refer to its own node, so the node part is omitted and only v is supplied.

- A *Local Physical Address (LPA)* is a 48-bit quantity,

  $$\text{LPA} = n_{16} : p_{32}$$

  where the p is a physical address on node number n. As with LVAs, the node part is usually implicit (the current node).

- A *Global Virtual Address (GVA)* is a 64-bit quantity,

  $$\text{GVA} = s_{32} : o_{32}$$

  where the o is a byte offset within segment s.

An executing program manipulates local and global virtual addresses. Local references always use local virtual addresses, while remote references always use global virtual addresses. Native M88110 page translation maps local virtual addresses into local physical addresses. That is, node n decomposes v into a virtual page frame number and an offset within the page,

$$v = vpn_{20} : offset_{12}$$

where *vpn* is the virtual page number. The page translation (PT) on node n maps the *vpn* into a physical page number, *ppn*,

$$vpn \xrightarrow{Page\ Xlate} ppn$$

So, the local physical address p is computed as,

$$v = ppn_{20} : offset_{12}$$

where *offset* is copied from v. In contrast, segment translation, supported by the synchronization coprocessor, maps a global virtual address into a local virtual address. A *segment descriptor* encodes a set of attributes (*e.g.*, valid, writable), how a segment is interleaved across nodes, and the base local virtual address for the segment,

$$segment\text{-}descriptor[s] = \langle attributes,\ interleave\text{-}map,\ LVA_{base} \rangle$$

For example, suppose that a thread issues an **rload** of the GVA **A**$= s_{32} : o_{32}$, where s is the segment and o is the offset within the segment. Before formatting an **rload_msg**, the message formatter fetches the descriptor for s in its segment translation buffer[9]. The mapping information is used to translate **A** into the LVA $n_{16} : v_{32}$, where n is a node number and v is a local virtual address on node n. The **rload_msg** is routed to node n, where the virtual address is translated, by page translation hardware, into a local physical address $n_{16} : p_{32}$. The segment translation takes place on the requesting node, while the page translation is performed by the destination node. Figure 8 summarizes.

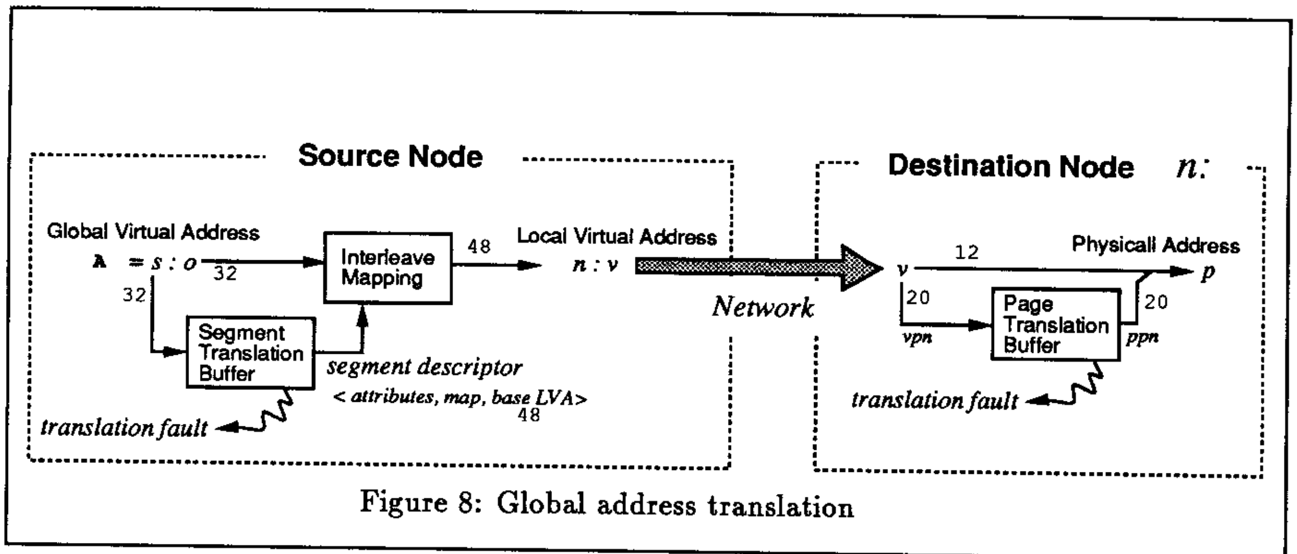


Figure 8: Global address translation

Continuations

A continuation is the pair ⟨ **FP**, **L**$_S$ ⟩ comprising a pointer to an activation frame, **FP**, and a start processor code label, **L**$_S$. It is particularly useful to pack a continuation into a single 64-bit word.

Recall, a given activation frame is mapped entirely onto a single node, and that all address arithmetic on frames are performed locally on local virtual addresses. It is thus possible to refer to frames only by their local virtual base addresses, which are 48-bit quantities. Now, assume a convention whereby the first word in a frame holds a pointer (a local virtual

[9]If the descriptor is not found in the segment translation buffer then a trap is elicited on the M88110, which can then, under software control, load the segment descriptor it finds in a global hash table, for example.

address) to the base of the start processor program text, SIP_{base}. This lets us encode L_S as a displacement from the SIP_{base},

$$L_S = \text{SIP}_{base} + \delta$$

where δ is the unsigned displacement. A continuation is encoded into 64-bits as follows,

$$C = \langle n_{16} : v_{32}, \delta_{16} \rangle.$$

It is possible to further compress the encoding by enforcing modulo-alignment constraints on v (*e.g.*,cache-line boundaries or larger) and δ. It might be desirable to reduce it to less than 52 bits, so as to fit within an IEEE double precision NaN.

5.3 Executing Dataflow Instructions on M88110

We now show how the M88110 implements `rload`, `rstore`, and `start` instructions. These instructions cause the (nonblocking) formation and transmission of `msg_rload`, `msg_rstore`, and `msg_start` messages, respectively. We will also show how the M88110 implements the `next` instruction.

The message formatter within the synchronization processor implements a number of memory-mapped registers. The main ones are shown below:

Register	Description
mOP	*Message operation*
mA	*Destination address*
mI	*Continuation start code displacement*
mV	*Message value*
mDFP	*Cached copy of* DFP

Some Message Formatter Registers

The `mDFP` register is a cached copy of the M88110's `DFP` register (the current activation frame). This register is automatically updated when the M88110 performs a `next` operation (see below).

To send a message, the M88110 first stores a global address (for `rload` or `rstore`) or an activation frame pointer (for `start`) in the `mA` register. Then, the start code displacement, δ, is written into the `mI` register. The displacement is used to form the return continuation (together with the contents of `mDFP`) in the case of `msg_rload` and `msg_rstore`. If the message has a value-part (*i.e.,* a `msg_start` or `msg_rstore`, then the M88110 stores the value into the `mV` register. Finally, the desired operation is written into the `mOP` register. Writing to this register automatically causes the appropriate type of message to be formatted and transmitted.

5.3.1 Implementing `rload`

For example, the following M88110 code sequence implements the `rload` of 64-bit (double) value. Assume that the M88110 register `rMF` contains a pointer to the message formatter

register set, register rA contains the 64-bit global virtual address of the word to read, and register rI contains the start code displacement, δ:

```
L_do_rload.d:
    st.d   rA, rMF, _mA           ; address to load into formatter reg mA
    st     rI, rMF, _mI           ; start code disp. into formatter reg mI
    or     rmsg, _rload.d, r0     ; formulate rload command to fetch 64 bits
    st     rmsg, rMF, _mOP        ; tell formatter to launch rload message
```

Note that the M88110 instruction `st.d rA, rMF, _mA` causes the contents of the double (64-bit) register rA to be stored at the address determined by adding the contents of register rMF to the immediate constant _mA (which we assume adjusts the address to point to the message formatter's mA register).

An Address-Encoding Optimization

While the above rload sequence is straight-forward, it is rather inefficient as compared with the native M88110 load instruction (ld.d). As an optimization, we will use the least significant bits of the addresses from the M88110 to pass information from the M88110 to the message formatter. Suppose that the message formatter decodes a range of 32-bit local addresses during an M88110 st.d as follows:

SELECT	δ	msg_op	000
8	16	5	3

Optimized Message Formatter Address Decoding

The message formatter is selected whenever the M88110 performs a st.d operation to any address where the upper eight bits are set to required SELECT bit pattern.[10] Here is what happens when the st.d executes:

$$\begin{aligned} \texttt{mA} &\leftarrow \text{double value written} \\ \texttt{mI} &\leftarrow \delta \\ \texttt{mOP} &\leftarrow \texttt{msg_op} \end{aligned}$$

This also causes the message encoded by msg_op to formatted and transmitted.

Now, the M88110 can issue a single instruction to initiate the rload. Assume that the M88110 register rSEL is all zeros, except for the upper eight bits, which are set to the message formatter SELECT code. Also assume that δ a small constant (< 255) called _delta:

```
L_do_rload.d:
    st.d   rA, rSEL, (_delta << 8) || (_rload.d << 3)    ; initiate rload
```

[10]That is, this feature consumes 1/256 of the local virtual address space.

Note that the expression `(_delta << 8) || (_rload.d << 3)` is evaluated by the assembler, and is reduced to a 16-bit instruction immediate. The M88110 `rst.d` instruction stores the contents of the 64-bit register `rA`, which is assumed to contain the GVA of the location to read, to an address encoded in the fashion of the table above.

Finally, this is how the message formatter generates the `msg_rload` message:

$$\begin{aligned}
\mathtt{A} &= \mathtt{Register[mA]} \\
\mathtt{FP} &= \mathtt{Register[mDFP]} \\
\delta &= \mathtt{Register[mI]} \\
\hat{n}.\hat{v} &= \mathit{SegmentXlate}(\mathtt{A}) \\
C &= \langle \mathtt{this_node.FP}, \delta \rangle \\
\text{Send message:} & \quad \mathtt{msg_rload}\ \hat{n}.\hat{v}, C
\end{aligned}$$

5.3.2 Implementing rstore

The implementation for `rstore` is similar to `rload`, except that we must also supply the value to store. This is accomplished by writing the value to the message formatter `mV` register, and then supplying the address:

```
L_do_rstore.d:
    st.d   rV, rMF, _mV                                  ; value to store
    st.d   rA, rSEL, (_delta << 8) || (_rstore.d << 3)   ; initiate rstore
```

The code assumes that the value to store is in register `rV`. The first `rst.d` writes the value to store into message formatter register `mV`. The second `rst.d` actually causes the network message to be initiated by supplying the address to write and the return continuation for the write acknowledgment. This is how the message formatter generates the `msg_rstore` message:

$$\begin{aligned}
\mathtt{V} &= \mathtt{Register[mV]} \\
\mathtt{A} &= \mathtt{Register[mA]} \\
\mathtt{FP} &= \mathtt{Register[mDFP]} \\
\delta &= \mathtt{Register[mI]} \\
\hat{n}.\hat{v} &= \mathit{SegmentXlate}(\mathtt{A}) \\
C &= \langle \mathtt{this_node.FP}, \delta \rangle \\
\text{Send message:} & \quad \mathtt{msg_rstore}\ \hat{n}.\hat{v}, \mathtt{V}, C
\end{aligned}$$

5.3.3 Implementing start

The implementation for `start` is just like `rstore`, only instead of writing the address to store, we supply a continuation for the remote frame. Assume that M88110 register `rRC` contains the continuation of the remote frame:

```
L_do_start.d:
    st.d   rV, rMF, _mV                           ; value-part of start msg
    st.d   rRC, rSEL, (_adj << 8) || (_start.d << 3)   ; initiate start
```

Here, _adj is considered an *adjustment* to the δ-part of the supplied remote continuation. Although the M88110-side looks like an rstore, the response of the message formatter quite different:

$$\begin{aligned} \texttt{V} &= \texttt{Register[mV]} \\ \texttt{adj} &= \texttt{Register[mI]} \\ \langle n.v, \delta \rangle &= \texttt{Register[mA]} \\ \hat{\delta} &= \delta + \texttt{adj} \\ C &= \langle n.v, \hat{\delta} \rangle \\ \text{Send message:} &\quad \texttt{msg_start}\ C, \texttt{V} \end{aligned}$$

Encoding Operand Size and Memory Semantics

The operand size for rload, rstore, and start can vary from one to thirty-two bytes, in powers of two. The operand size is encoded as part of the message operation stored into register mOP, and is carried on the message. *E.g.,* rstore.b stores a byte, rload.s fetches four bytes, and start.q sends sixteen bytes.

Similarly, memory semantics are also encoded into rstore and rload messages. *E.g.,* rIload.d fetches eight bytes according to I-structure semantics. I-structure and M-structure operations are only defined for eight-byte operands, or bigger. In the case of larger operands, *i.e.,* sixteen and thirty-two bytes, I-structure semantics apply to the entire aggregate. The table below summarizes.

Operand	Size (Bytes)	Extension	start	rload/rstore Imperative	I-Structure	M-Structure
null	0	.n	√	√		
byte	1	.b	√	√		
halfword	2	.h	√	√		
word/single	4	.w/.s	√	√		
double	8	.d	√	√	√	√
quadword	16	.q	√	√	√	√
octword	32	.o	√	√	√	√

Implementing next

When the M88110 wants to begin executing a new thread it executes a next instruction which, from its perspective, is simply the popping of an FP,IP pair from the synchronization processor's thread queue, and then a transfer of control to the new IP. The synchronization

processor presents the head of the thread queue as a memory-mapped 64-bit register sQ, which contains the next FP and IP:

Note that FP and IP are both 32-bit local virtual addresses. Assume that the M88110 register rQ points to the synchronization processor register sQ, and that M88110 register rDFP contains a pointer to the current activation frame. It is also assumed that M88110 register rDIP is allocated adjacent to rDFP, such that when rDFP is loaded with a double (64-bit) value, rDFP recieves the most significant thirty-two bits and rDIP receives the least significant thrity-two bits. Here is the sequence that M88110 executes to implement next:

```
ld.d    rDFP, rQ, 0    ; pop FP,IP pair from head of queue
jmp     rDIP           ; jump to the new thread
```

The act of popping the FP, IP pair from the queue also has the side-effect of setting the message formatter's cached version of the current data processor activation frame (mDFP) to the new FP.

5.4 The Synchronization Processor

5.4.1 The RMem Processor

The RMem processor is a finite-state machine that consumes msg_rload and msg_rstore messages destined for the current node, and either responds with an msg_start back to the requesting processor, or elicits a trap on the M88110 for handling conditions beyond its capability. Other than normal imperative operations, the processor will implement at least the basic layer of I-structure and M-structure protocol. There are a number of open issues:

- How should presence bits be maintained? Options are: additional bits tagged to each word (or a cache line's worth of words), bits packed into 64-bit words in the local virtual address space.
- How should multiple-deferred readers of an I-structure or M-structure be handled? Options are: Monsoon-style request threading, trap the local M88110, and local deferred list management.
- How should errors, like multiple writes, be handled? Options are: respond with a msg_error, trap the local M88110.

Note that the RMem processor never need perform segment translation, because the frame pointers of the return continuations are always represented as local virtual addresses. It simply consults the node-part of the frame address when formulating an msg_start response.

5.4.2 The Start Processor

The start processor handles all msg_start messages destined for the current node. The start processor implements the first "layer" of message handling and synchronization;

1. Writes the value-part of the start message into an offset in the activation frame also specified by the start message.
2. Performs a join operation on counter values in the the activation frame specified by the start message.
3. Posts ready threads to a queue that can be popped by the M88110 when executing a next operation.

There are three primary ways in which an M88110 and its local Start Processor interact; (1) the M88110 can execute a rload, rstore, or start which either directly or indirectly results in a msg_start message destinated to the local Start Processor; (2) in the course of processing a msg_start message, the Start Processor writes activation frame locations which are subsequently read by an M88110 thread; (3) the M88110 executes a next instruction which pops the continuation for the next thread to execute from a queue managed by the Start Processor.

Of the three modes of M88110–Start Processor interaction, communication through shared activation frame locations is the most unstructured. We can rationalize the communication by establishing a set of conventions for the usage of storage within a frame. Logically, we divide an activation frame into four areas:

Activation Frame Area	Start Proc.	M88110
Linkage	*Read-Only*	R/W
Join Counters	*R/W*	*Read-Only*
Message Values	*Write*	*Read*
Inter-Thread Values		R/W

Recall, a msg_start message comprises a continuation and a value,

$$\text{msg_start} = \langle \text{FP}, \hat{\delta} \rangle, \text{V}$$

where FP is a pointer to the base of an activation frame, and V is variable-sized value, from zero to thirty-two bytes. The code pointer for the message handler is computed as,

$$\text{SIP} = \text{SIP}_{\text{base}} + \delta$$

where SIP_{base} is, by convention, stored in the first word in the current activation frame, *i.e.*, FP[0].

Here are the Start Processor registers which are automatically loaded upon dispatching to the handler of a new message:

Register	Description
SIP	*Message handler instruction pointer*
SFP	*Current activation frame base*
SV	*Message value (MSW)*
SV1	*Message value*
SV2	*Message value*
SV3	*Message value (LSW)*

Some Start Processor Registers

One of the first actions of most every message handler is to write Message Value registers to offsets in the activation frame pointed to by SFP.

The Scheduling Hierarchy

An important new dimension of *T is an explicit hierarchy of scheduling data processor threads. In Monsoon, the only control over scheduling is the ability to force a recirculation of a token; this is key concept behind a Monsoon thread. The principle motivation to extend the control over scheduling beyond the thread level is to induce temporal locality:

- Biasing scheduling *across a small subset of frames* can enhance processor data and instruction cache hit rates.
- Biasing scheduling towards threads *within a frame* permits the speculative allocation of temporary registers. That is, if threads related to the same frame are scheduled one after the other, then the threads can (potentially) communicate values through temporary registers which might otherwise be indeterminate.

The Frame Registry

Our implementation implements the scheduling hierarchy through a very simple mechanism called a **frame registry**, a small associative table of activation frame base pointers. When the start processor attempts to post an FP,IP pair, the frame registry is queried. If the FP is found in the registry, then the pair is enqueued into a hardware-managed thread queue. There is (logically) one such thread queue for each registered frame. If the FP is not found in the registry, then a trap is elicited (probably on the M88110) and the IP is enqueued onto a software-managed list of ready, but presently-inactive, frames.

When the M88110 executes a next instruction, it pops an FP,IP pair from one the registered frames. The hardware biases scheduling within the frame by always giving an FP,IP

pairs from the same frame until that frame's queue is depleted. Only then does the popping focus on another registered frame. As an option, "cleanup" and "startup" threads can be executed whenever the scheduling crosses a frame boundary.

Execution continues in this fashion until all the queues of the registered frames are empty. When the M88110 executes a `next` instruction under this condition, it is given an "out of work" thread which, presumably, deregisters the least recently used frame

and registers a frame from the software managed queue of ready frames.

5.4.3 The Local Memory Controller

The local memory controller services local memory read and write requests from the start and RMem processors. These requests are always in terms of local virtual addresses, so the memory controller must also support page translation to local physical addresses. We envision a simple translation lookaside buffer that traps the M88110 upon a miss.

The local memory controller also acts as a DRAM controller, and is invoked whenever cache lines are moved to and from DRAM.

6 Discussion

6.1 Comparison with other Dataflow Processors

All dataflow machines that have been designed or built since 1988 (*e.g.,* Monsoon, EM-4, Sandia Epsilon, IBM Empire, P-RISC, McGill's argument-fetching machine) are frame based and, thus, have avoided an associative store for tokens. Since many researchers had considered the associative store to be totally unacceptable for high performance machines, its elimination has caused a new surge of interest in dataflow architectures. Now all the attention seems to be focused on gaining sequential-thread performance.

Ideally one would like to design a processor which can compete with the best commercial microprocessors without sacrificing the efficiency of dataflow architectures for synchronization and latency tolerance. This goal seemed out of reach even two years ago. Hopefully, this paper shows that the goal is achievable. The best proof would be if any machine built out of P*T-type processors, that is, a P-type processor with its *T co-processor, can outperform any machine built out of P-type processors alone. In the following we briefly compare the *T approach to other dataflow machines.

P-RISC

*T is the first proposal for a dataflow machine that can keep total compatibitily with an exisiting microprocessor. In this respect it goes beyond P-RISC [21] because P-RISC only tried to maintain instruction set compatibility, while the *T approach uses an existing microprocessor as is. P-RISC also suggested interleaved pipelining of threads, as in Monsoon

and the HEP, which required a number of active threads to keep a processor completely utilized. This could show poor performance on existing single-thread codes. The *T approach also takes the preparation to run a thread out of the data processor pipeline. Thus, the data processor pipeline is not disturbed by short threads which need be executed only to determine if another thread is ready to execute.

It is worth noting that maintaining compatibility with an exisiting processor does have some costs. For example, direct frame-based addressing, as described in the P-RISC paper, may be more desirable for implementing Id-like languages. It may be also be useful for parallel implementation of C or Fortran.

Monsoon, Sandia's Epsilon II and IBM Empire

Monsoon, as discussed earlier, has poor single thread performance as compared to a modern RISC processor [23]. This can be attributed to a lack of ways to address local state, *i.e.,* registers. The notion of efficient threads and registers were not the primary motivations behind Monsoon, so it is interesting that threads can be done at all. Monsoon has proven to be excellent in exploiting fine-grain parallelism. It is our goal to make the *T processor as efficient as Monsoon in executing synchronization code. *T may not quite compete with Monsoon on threads that are one or two instructions long. However, we hope that the crossover point for thread size will be small enough so that compiling Id, a non-strict language, remains relatively easy. It is certain that *T would do much better than Monsoon on existing Fortran or C codes.

The thread model in Sandia's Epsilon II machine is also quite weak as compared to a general RISC processor [15]. We do not know enough about IBM's Empire architecture to compare it properly with our approach. Neither of these machines has paid much attention to running exisiting codes.

ETL's EM4

The pipeline of the EM-4 [25] can be viewed as a two stage pipeline where the first stage takes tokens from the network and essentially prepares threads for execution, while the second stage executes the thread. These correspond roughly to our Start and Data Processors, respectively, but unlike our system, they do not operate independently and asynchronously. The EM-4 does not have any analog to our RMem Processor. In the design of EM-4, not much consideration has been given to executing existing codes efficiently. The software strategy for EM-4 has not been clearly articulated.

McGill argument-fetching machine

In the McGill argument-fetching architecture [14], the processor is partitioned into a Pipelined Instruction Processing Unit (PIPU) and a Dataflow Instruction Scheduling Unit (DISU), similar to our Data and Start Processors, respectively. The McGill partitioning appears primarily motivated by a desire to reduce data movement (and, consequently, the complexity

of data paths); the signal flow graph that is executed by the DISU still retains the full structure of the original dataflow graph, so that every instruction is still enabled with the general synchronization mechanism. Thus, single-thread performance is not likely to be very good, and it is difficult to utilize registers or caches. Their approach to non-local memory accesses is quite different from our approach (it is very similar to the approach in the HEP, where remote accesses are taken aside into a "parking" area during the access).

6.2 Comparison with other innovative von Neumann Processors

The developement of commercial RISC processors have been primarily in the direction of superscaler pipelines where mutiple instructions are dispatched from a sequential instruction stream. No attention has been paid to parallel processing except for some hooks for maintaining cache coherence. However, there are some developments in the research community where close attention *is* being paid to memory latency and synchonization issues.

Machines with Cheap Context Switching

A popular approach to tackling memory latency in parallel machines is to replicate the processor state a small number of times (say, 4 or 8) so that, when the processor has to perform a non-local access, it can switch to another thread without having to save the current processor state [1, 17, 30]. In order to to reduce the number of non-local accesses, it is sometimes proposed that non-local data be cached locally, using some mechanism (*e.g.,* directories) to keep such caches globally coherent. Hopefully, by the time the processor tries to switch back to the original thread, the non-local data will be available. If the access has not completed, a trap is taken and it is handled explicitly in software.

Effective, coherent caches for large parallel machines have yet to be demonstrated. In *T instead of replicating processor state, we have chosen to utilize those resources by organizing them into three specialized processors for intra-thread, inter-thread and remote memory functions.

Denelcor HEP and Tera machines

The Denelcor HEP, designed by Burton Smith, used multithreaded processors to mask the latency of memory access. The processor maintained upto 8 sets of registers, switching between 8 threads on each clock on a round-robin basis (so, single-thread performance was not a priority). All memory was global— there was no local memory. A thread that accessed global memory was taken out of the main pipeline and kept aside till the response arrived. The processor had a pool of upto 64 threads to keep it busy. Each memory location had presence bits for data level synchronization.

Unlike *T, which has no architectural limit on the number of threads, the HEP permitted too few threads to be able to compile effectively for it. Also, even though memory-referencing threads were taken out of the main pipeline, there was still some degree of busy-waiting on long-latency accesses. The limitations of the HEP are also discussed in [6].

Burton Smith's new machine at Tera Computer fixes many shortcomings of the HEP and provides much more control over the scheduling of memory references than any other machine we are familiar with [19, 27]. Tera is complex enough that a feature-by-feature comparison will take too long to describe here. It is likely that it has a superset of the features of *T.

The J-Machine

The J-Machine is a massively parallel machine whose processors are Message-Driven Processors [12]. It is an attempt to improve on Hypercube-like designs by providing very fast vectoring to message handlers, fast access to the contents of incoming messages, and minimal processor state, in order to reduce the latency of message handling. While vectoring to a message handler may be fast, the handler has to execute code to move data from the message into registers, whereas our Start Processor has its `SFP` and `SV` registers automatically loaded. Where we use three specialized processors in a node for three classes of activities (intra-thread, inter-thread, and remote memory accesses), the MDP is a single processor that does all the work. Where we can have a small register file in the Start Processor and a large one in the Data Processor, the MDP uniformly has a small register file, which is likely to penalize single thread performance.

6.3 Conclusion

Conventional microprocessors are excellent at executing single threads, but do not handle long latency operations or synchronization operations well. Consequently, unless we carefully craft our programs to minimize communication, a massively parallel machine built with these components is likely to have poor utilization at each node[11].

Dataflow processors have complementary strengths and weaknesses— they are very good at handling long latencies and providing cheap synchronization, but have poor single-thread performance. Consequently, a workstation built out of such components is not likely to be competitive; therefore, such processors are not likely to become commodity parts.

We believe that *T is the first proposed architecture that can execute single threaded programs as efficiently as conventional microprocessors, fine-grain parallel programs as efficiently as dataflow processors, and provide a smooth spectrum of operating points in between. Our preliminary explorations on its implementation indicate that it is eminently buildable.

A Additional Instructions for the Data Processor

A.1 The `fork` instruction

The `fork` instruction is a special case of the `start` instruction in which the destination frame is the same as the current frame, and no data value is transmitted. Of course, in this case,

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

no message need be sent into the network— it will be short-circuited back directly:

Data Processor Instruction:	`fork rI`
Semantics:	Let L_S = `Register[rI]` Let `FP` = `Register[DFP]` Send message: `msg_startv FP,`L_S`,foo`

where `foo` is an arbitrary value.

A.2 Instructions for global data access with data level synchronization

One may also associate extra state bits with a global data location that mark it *empty* or *full*. These can be exploited with instructions that perform global data access with data level synchronization. When a synchronized data location is empty, it contains a list of pairs, where each pair consists of a frame pointer and an instruction pointer. All freshly allocated synchronized heap locations are marked empty, and contain an empty list of pairs.

The `rIload` and `rIstore` instructions have the same instruction formats and behavior as the `rload` and `rstore` instructions, except that the messages that they generate have `msg_rIload` and `msg_rIstore` message opcodes:

Data Processor Instruction:	`rIload rA, rI`
Semantics:	Let `A` = `Register[rA]` Let L_S = `Register[rI]` Let `FP` = `Register[DFP]` Send message: `msg_rIload A,FP,`L_S

Data Processor Instruction:	`rIstore rA,rV,rI`
Semantics:	Let `A` = `Register[rA]` Let `V` = `Register[rV]` Let L_S = `Register[rI]` Let `FP` = `Register[DFP]` Send message: `msg_rIstore A,V,FP,`L_S

The interesting difference between these instructions and `rload`/ `rstore` is in the treatment of the messages at the remote node:

RMem Message:	`msg_rIload A,FP,`L_S

Semantics:	if full?(`Memory[A]`)
	   Let `V` = `Memory[A]`
	   Send message: `msg_start, FP,`L_S`,V`
	else
	   enqueue (`FP`,L_S) at `Memory[A]`

Note that if the location is full, an `msg_rIload` message behaves just like an `msg_rload` message. Otherwise, the message information is queued there to be handled later, in response to an `msg_rIstore` message:

RMem Message:	`msg_rIstore A,V,FP,`L_S

Semantics:	if empty?(`Memory[A]`)
	Let `queue` = `Memory[A]`
	`Memory[A] := V`
	Set presence bit of `Memory[A]` to "full"
	For each (`FP'`,M_S) in `queue`
	Send message: `msg_start FP'`,M_S,`V`
	Send message: `msg_start, FP,`L_S`,foo`
	else
	error `"Multiple writes not allowed"`

If the location is empty and no readers are queued there, it behaves just like an `rstore`, just storing the value there. If there are any queued readers, the value is also sent to them. Finally, if the location is full, it is a run time error. As in `rstore`, an acknowledgement message is also sent.

Remote loads and stores with data-level synchronization may be used to implement "I-structure" operations[12], which permit overlapped operation of the producer and consumer of a data structure (see [8]).

Two more remote load and store operations with data-level synchronization are available which make it easy to implement atomic updates on remote locations. This is useful for shared queue management, among other things. The instructions have the same format as `rload` and `rstore`:

Data Processor Instruction:	`rtake rA,rI`
Semantics:	Let `A` = `Register[rA]` Let L_S = `Register[rI]` Let `FP` = `Register[DFP]` Send message: `msg_rtake A,FP,`L_S

Data Processor Instruction:	`rput rA,rV,rI`
Semantics:	Let `A` = `Register[rA]` Let `V` = `Register[rV]` Let L_S = `Register[rI]` Let `FP` = `Register[DFP]` Send message: `msg_rput A,V,FP,`L_S

Again, the interesting difference between these instructions and `rload/rstore` is in the treatment of the messages at the remote node:

RMem Message:	`msg_rtake A,FP,`L_S
Semantics:	if full?(`Memory[A]`) Let `V` = `Memory[A]` Send message: `msg_start FP,`L_S`,V` Set presence bit of `Memory[A]` to "empty" else enqueue (`FP`,L_S) at `Memory[A]`

[12]Indeed, this is the reason for the "I" in the operation names.

Note that if the location is full, an `msg_rtake` message returns the value just like an `msg_rload` message, but it also resets the location to the empty state. Otherwise, the message information is queued there to be handled later, just like an `msg_rIload` message.

RMem Message:	`msg_rput A,V,FP,`L_S

Semantics:	if empty?(`Memory[A]`)
	  Let `queue` = `Memory[A]`
	  if `queue` is empty
	    `Memory[A] := V`
	    Set presence bit of `Heap[A]` to "full"
	  else
	    Let (`FP'`,M_S) = `head(queue)`
	    Send message: `msg_start FP'`,M_S,`V`
	    `Memory[A] := tail(queue)`
	  Send message: `msg_start FP,`L_S`,foo`
	else
	  error `"Multiple writes not allowed"`

As in `msg_rIstore`, if the location is not empty, it is a run time error. Otherwise, if no readers are queued there, it behaves just like a `msg_rstore` or `msg_rIstore`— the value is simply stored there and the location is set to the full state. If there are queued readers, the first reader is taken off the queue and the value is sent there; the location remains empty.

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] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, and A. Webb. The warp computer: Architecture, implementation and performance. *IEEE Transactions on Computers*, C-36(12), December 1987. Also: CMU-RI-TR-87-18, Carnegie Mellon University, The Robotics Insititute.

[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.* Academic Press, 1990 (to appear).

[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 Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 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] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

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

[10] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, January 1991 (to appear).

[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, August 1990.

[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. pages 48–56, November 1980.

[14] 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.

[15] 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.

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

[17] 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.

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

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

[20] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989.* Academic Press, 1990 (to appear).

[21] 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.

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

[23] 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.

[24] G. M. Papadopoulos and K. Traub. Multithreading: A Revisionist View of Dataflow Architectures, November 1990. Draft paper, submitted for publication.

[25] 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.

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

[27] M. R. Thistle and B. J. Smith. A processor architecture for horizon. In *Prof. IEEE Supercomputing Conference, Florida*, pages 35–41, 1988.

[28] K. R. Traub. A compiler for the mit tagged-token dataflow architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

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

[30] 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.

Contents

1 **Introduction** 1

2 **Lessons from the past** 2

2.1 The brave new world . . . 2

2.2 von Neumann microprocessors . . . 3

2.3 Dataflow architectures . . . 4

2.3.1 Lessons from the past: the bottom line . . . 6

3 ***T: the abstract model** 6

3.1 Frames as the basis for locality . . . 6

3.2 The *T node architecture . . . 7

3.2.1 Threads . . . 9

3.2.2 Global data accesses . . . 12

3.2.3 Inter-thread and inter-frame scheduling control for better cacheing . . 14

4 **An example: SAXPY** 15

4.1 *T code for SAXPY . . . 15

4.2 Compiling for *T . . . 21

5 **An Implementation of *T** 22

5.1 A *T node based on M88110 . . . 23

5.2 Global Addresses and Virtual Memory . . . 24

5.3 Executing Dataflow Instructions on M88110 . . . 27

5.3.1 Implementing `rload` . . . 27

5.3.2 Implementing `rstore` . . . 29

5.3.3 Implementing `start` . . . 29

5.4 The Synchronization Processor . . . 31

5.4.1 The RMem Processor . . . 31

5.4.2 The Start Processor . . . 32

5.4.3 The Local Memory Controller . . . 34

6 **Discussion** **34**

6.1 Comparison with other Dataflow Processors 34

6.2 Comparison with other innovative von Neumann Processors 36

6.3 Conclusion 37

A **Additional Instructions for the Data Processor** **37**

A.1 The `fork` instruction 37

A.2 Instructions for global data access with data level synchronization 38