This is a draft

Analyzing the Benefits of a Separate Processor to Handle Messages for Fine-grain Multithreading

David Metz and Ben Lee

Department of Electrical and Computer Engineering

Oregon State University

Corvallis, OR 97331

email: metzda, benl@ece.orst.edu

Abstract

This paper discusses the benefits of having a separate processor to handle messages in Massively Parallel Architectures and proposes hardware solutions to provide atomicity between the main processor and the processor dedicated to handle messages. The proposed design is aimed at improving the performance by relegating the responsibility of handling messages to a separate processor. The hardware modifications are kept to a minimum in order not to disturb the original functionality of a modern RISC processor.

This paper has been submitted for publication in the Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing, 1995.

1. Introduction

Multithreading allows a processor to switch among multiple threads to tolerate unpredictable latencies due to remote memory requests and synchronization. One model, called Threaded Abstract Machine (TAM), supports interleaving of multiple threads by an appropriate compilation strategy and program representation rather than through elaborate hardware [1]. Experiments on TAM have already shown that it is possible to implement the fine-grain execution model on conventional architectures and obtain reasonable performance. This has been demonstrated by compiling Id90 [5] programs to TL0, the TAM assembly language, and finally to the native machine code for a variety of platforms, mainly CM-5 [10]. Yet these studies also show a basic mismatch between the requirements for fine-grain parallelism and the underlying conventional architecture, and thus considerable improvement is possible through hardware support. One such study shows that an improvement can be made to the execution of TAM control instructions by simply incorporating a special instruction called *conditional double branch and pop* into ISA of the SPARC processor [2].

Another source of this mismatch is the handling of messages. In the TAM execution model, messages are handled by compiler generated codes that extract data from messages and dispatch them to appropriate threads. However, in TAM, message handling constitutes as much as 22%-45% of total TL0 instructions executed and therefore represents a significant portion of the overhead required to support the fine-grain execution model.

In light of the aforementioned discussions, this paper presents a design modification required to efficiently support message handling for fine-grain parallelism on a stock processor. The use of a separate processor is proposed to reduce the cost of handling messages. Atomicity is guaranteed without sacrificing performance, and the hardware modifications are kept to a minimum so as not to disturb the functionality of a conventional RISC processor. Although the discussion is based on the SPARC processor, the design issues apply to other RISC processors as well.

2. Threaded Abstract Machine

Although it grew out of work on dataflow, the TAM execution model exposes synchronization, thread scheduling, and storage management to the compiler, and is explicit in the machine language. Unlike other dataflow

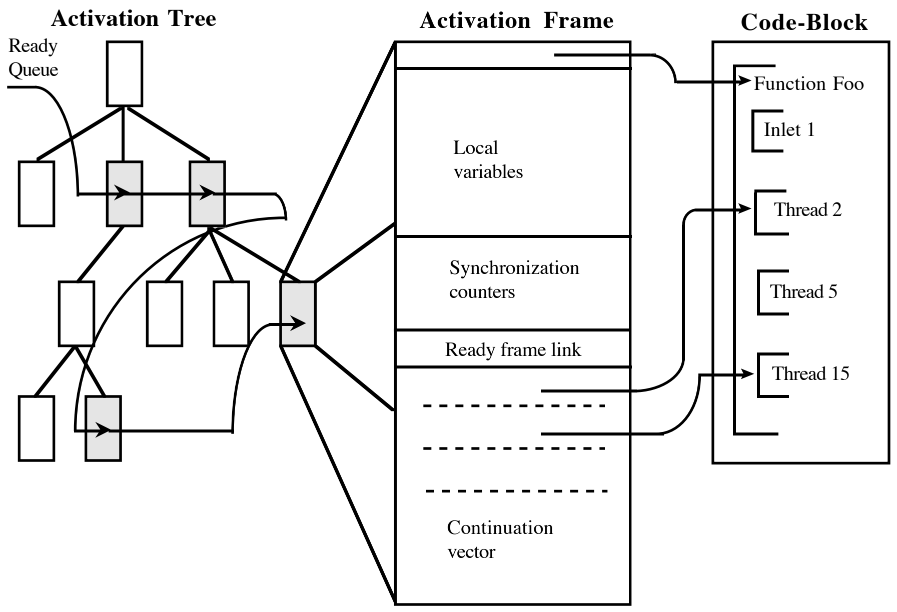

Figure 1: TAM activation tree.

proposals, TAM design aims to minimize the hardware overhead of multithreading and exploits locality even under asynchronous execution.

Figure 1 illustrates a TAM activation tree. A TAM program consists of a collection of code-blocks where each *code-block* typically represents a loop-body or a function. A code-block comprises of *threads* and *inlets*. Invoking a code-block involves allocating an *activation frame*—which is analogous to the stack frame for conventional subroutine calls—depositing argument values into the frame, and enabling threads for execution within the context of the frame. Initialization also consists of setting the values for synchronization counters, or *entry counters*, stored within the frame. Unsynchronizing threads require no counters. A frame can be in one of three states: running, ready, or idle. An *idle* frame has no enabled threads. It becomes *ready* and is queued for scheduling as soon as a thread is enabled. A scheduled frame is considered *running* or resident and is executed until it has no enabled threads. A *quantum* is the set of threads executed during a single residency of the frame.

TAM supports `FORK`[1] instructions that enable threads within the current activation. If the thread is an unsynchronizing one, the pointer to the thread is pushed onto the *local continuation vector* (LCV), which contains

[1] For clarity, all TAM TL0 instructions are written in capitalized `COURIER` font to distinguish them from SPARC assembly instructions or pointer variables, which are written in non-capitalized `courier` font.

pointers to all enabled threads within the current quantum. If the frame is not active, threads are pushed onto the frame's *remote continuation vector* (RCV). Thus, the LCV can be viewed as a fast, short-lived extension of the RCV. If the thread requires synchronization, the `FORK` instruction decrements the entry count for the thread. If the decremented count is zero, the thread is enabled and pushed onto the LCV; otherwise, the count is stored back. A `SWITCH` instruction forks one of two threads depending on a condition. A `STOP` instruction terminates the current thread and causes some other enabled thread to begin execution. This is done by popping a thread from the LCV. When the LCV is empty or the execution of the current code-block has completed, the processor executes the *leave-thread*. The `SWAP` instruction within the leave-thread then transfers the control to a frame pointed to by the *ready frame link* within the current frame. The following is a simple TL0 code for a TAM thread that adds two frame memory slots, stores the result in a register, and forks another thread.

```
THREAD 1
      ADD   ireg0.i = islot0.i islot1.i[1]   % Add frame slots islot0 and islot1
                                             and store the result in the TAM
                                             temporary register ireg0
      FORK  4.t                              % FORK the thread "4"
      STOP                                   % Pop the thread from LCV
```

TAM also supports inter-frame messages, which arise in passing arguments to an activation, returning results, and global heap accesses. This is done by associating a set of inlets with each code-block. *Inlets* are compiler generated message handlers that copy arguments into the frame and enable threads depending on the message using `POST` instructions. `POST` instructions push threads onto the LCV, if the threads are for the current frame; otherwise, the threads are pushed onto the RCV. For example, a `SEND` operation delivers a data value to an inlet relative to the target frame. A sample inlet code is shown below:

```
INLET 1
      RECEIVE islot4.i                       % Receive the value and store it in
                                             the frame slot islot4
      POST  5.t                              % Pushes the thread "5" onto LCV or RCV
      STOP                                   % Pop a thread from LCV
```

Global data structures in TAM provide synchronization on per-element basis to support I-structure and M-structure semantics [1]. If the I-structure element is empty, a read is deferred until the corresponding write takes place. A remote I-structure operation generates a request for a particular heap location and the corresponding response

[1] All registers and frame slots are statically typed. '`i`' indicates that it is of integer type.

is received by an inlet. Meanwhile, the requesting processor continues with other enabled threads. In TAM, these split-phase transactions are supported by specialized `SEND` instructions, in the form of `IFETCH` and `ISTORE`, which are used to read and write to the data structures, respectively.

Note that both threads and inlets cooperate in determining the flow of computation. But they also compete for shared resources such as synchronization counters and the LCV. Therefore, any implementation must guarantee that thread and frame scheduling operations are atomic with respect to `POST` instructions from the inlets.

2.1. Mapping TAM onto a Stock Processor

TAM is codified in a pseudo-machine language TL0. TL0 instructions are primarily three-address, where the operands are constants, registers, or frame locations. No fixed limit is placed on the number of TAM registers; however, the compiler tries to use them as efficiently as possible. TAM translator is responsible for mapping TAM registers to physical registers or spill areas.

TL0 registers are implemented on the SPARC processor using a single register window [1]. The single register window is divided into three categories: special-function registers, thread registers, and inlet registers. The special-function registers hold important variables and constants such as pointer to the top of the LCV (`lcv`), node ID, frame pointer (`fp`), pointer to the base of current code-block, and a pointer to frame scheduling queue. The TL0 instruction pointer and the inlet instruction pointer are both mapped onto the SPARC program counter register. There are sixteen thread registers that are under the control of the register allocator. The inlet registers hold message related variables such as the inlet frame pointer (`ifp`) and function arguments.

3. Message Handling

Message reception can be implemented using either polling or interrupts. For *polling*, messages are stored into an on-chip queue. The network is polled and if there is a message, it dispatches to the code indicated by the first word of the message at the head of the queue, i.e., the inlet. The inlet first loads the message data into registers and then stores it into the frame memory. For *interrupts*, the network interface signals an interrupt on a message arrival causing a trap to the kernel. The kernel forwards the interrupt to the user process by creating a stack frame for the inlet and returning to it.

Due to the prohibitive cost of interrupts, TAM's implementation on the CM-5 is to explicitly poll the network once in every thread [1]. If the thread contains an instruction which might access the network, such as a SEND, then the poll is combined with this instruction. All other threads have an explicit poll inserted at the end of the thread. Each poll incurs a cost of 9 cycles, and the overhead of polling is between 4.98%-12.59% of TL0 instructions executed depending on the program. The advantage of polling is that the compiler decides when to poll for messages. Hence, atomicity is not a problem because there is a tight coupling between computation and communication.

It is clear that neither polling nor interrupts adequately provides a fast dispatch to user-level message handlers. Even if the cost of polling or interrupts can be significantly reduced, the execution of inlets constitutes a large portion overhead and thus an alternative choice is to dedicate a separate processor, i.e., an Inlet-processor, for handling messages. The responsibility of this processor is to execute the inlet code. The primary advantage of this is most of the overhead of handling messages can be off loaded to the Inlet-processor. For example, I-structure fetches and stores can be handled by the Inlet-processor without disturbing the thread execution of the Main-processor. The idea of having a separate processor to execute inlet code was proposed in the *T project by MIT [6]. However, the important issue of ensuring that FORK, STOP, and SWAP execute atomically relative to POST has not been adequately addressed. Therefore, the following section proposes the use of a separate processor for executing inlets and considers the issues of atomicity between the two processors.

4. A Design for Efficient Handling of Messages

In this section, a design is proposed to solve the problem of atomicity between the Main-processor and the Inlet-processor in the context of TAM. The primary objective is to ensure atomicity between the operations of the two processors without compromising performance. The Main-processor/Inlet-processor interface is shown in Figure 2. The design consists of the *MBus* (a SPARC standard) that connects the *Main Memory* to the two caches. The *Common Cache* holds thread instructions and frames. The *Inlet Cache* contains inlet instructions as well as heap elements. The separation of the two caches allow message handling to be done independent of the thread execution. The Main-processor dispatches all SENDs and heap operations to the Inlet-processor. A separate bus, called *MICBus* (Mainprocessor-Inletprocessor-Cache Bus) connects the two processors to the Common Cache. The

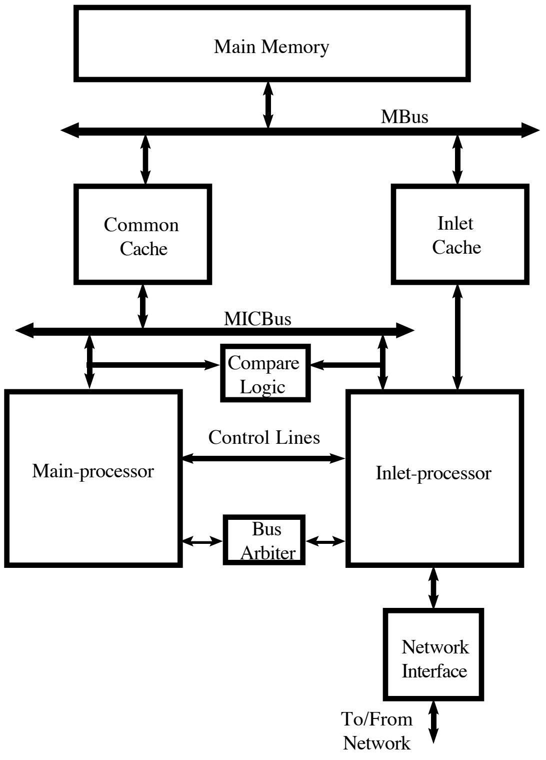

Figure 2: Main-processor/Inlet-processor interface.

Network Interface is directly integrated into the Inlet-processor freeing up the MICBus for thread execution. Since the Inlet-processor operates completely independent of the Main-processor, a separate connection is provided to the Inlet Instruction cache. Moreover, a *Bus Arbiter* is introduced that gives both processors the same priority. The set of *Control Lines* and the *Compare Logic* provide the necessary means to ensure atomicity between threads and inlets.

Based on these assumptions, the following operations must be atomic: Access to synchronization counters, and access to the LCV. The following subsections will discuss in detail each of these operations. A detailed discussion of other essential problems, such as inter-processor communication and the SWAP operation, is beyond the scope of this paper and can be found in [4].

4.1 Access to Synchronization Counters

The TL0 instructions that access synchronization counters are synchronizing `FORK`s and `POST`s. The parts of the SPARC assembly code for `FORK` and `POST` that access synchronization counters are shown below.

```
FORK                          (cycles)     POST                        (cycles)
ldb    sync[fp],tmp1          (2)          ldb    sync[ifp],tmp1       (2)
subcc  tmp1,1,tmp1            (1)          subcc  tmp1,1,tmp1          (1)
bnz,a  continue               (1/2)        bnz,a  continue             (1/2)
(or cdbp  thr_addr)           (2/3)        stb    tmp1,sync[ifp]       (3)
stb    tmp1,sync[fp]          (3)
```

Both sequences basically start by loading the synchronization counter (`ldb`) from the frame and decrementing it (`subcc`). If the decremented count is not zero, it is stored back using the delay slot (`stb`) and the next instruction is executed (at `continue`). On the other hand, if the decremented count is zero, a `FORK` will either branch to a thread[1] or push a thread onto the LCV, while a `POST` will push a thread on the LCV (if `fp=ifp`) or RCV (if `fp≠ifp`). The `cdbp`, which stands for *conditional double branch and pop*, is a special instruction that has been proposed to reduce the cost of implementing TAM control instructions. The basic idea is to hold the next thread pointer (`ntp`) from the top of the LCV in a special register called `r_ntp`. After the entry count has been decremented, if the count is zero, the `cdbp` instruction jumps to the thread at the location `thr_addr`; otherwise, it jumps to the address contained in `r_ntp`. If the control transfers to the location given by `r_ntp`, `r_ntp` is updated by popping the next enabled thread pointer into it, and then the delay slot is used to store back the entry count. This instruction eliminates the need to execute `STOP`s that pop threads from the LCV after branches to unsuccessful synchronizing threads. For a detailed explanation of `cdbp`, see [2].

An atomicity problem due to the access of a synchronization counter will occur only, if *both* the Main-processor and the Inlet-processor access the *same* synchronization counter, i.e., a problem exists if one processor tries to load the counter that has just been loaded by the other processor, but has not yet been stored back.

Most modern processors provide some form of atomic access to shared variables to facilitate multiprocessing [9]. Although these operations can be used to implement atomic accesses to synchronization counters, the generality of these instructions is unnecessary in the proposed design for the following reasons: First, unlike the UMA model, synchronization counters in the TAM model exist within the frame, i.e., they are local. Therefore, access to synchronization counters has to be atomic only between the Main-processor and the Inlet-

[1] The `FORK` sequence shown is for branch to a thread.

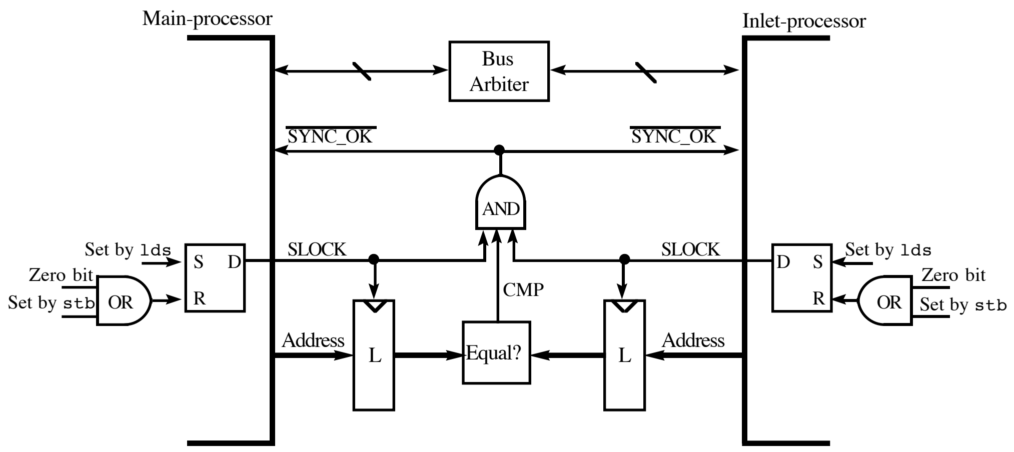

Figure 3: A design for atomic access to synchronization counters.

processor, not among an arbitrary number of processors. Second, by relying on the particularities of how the two processors interact, a specific hardware can be designed to minimize the overhead due to atomic accesses of synchronization counters (e.g., busy-waiting).

The proposed design is shown in Figure 3 and it operates as follows: The Bus Arbiter allows only one of the processors to access the MICBus. When this processor wants to access a synchronization counter it puts the address of the synchronization counter on the address bus. It also sets the S(ync)LOCK signal that is slightly delayed, so that the address can be held by the appropriate edge-triggered latch L. To ensure atomicity, the SLOCK signal must remain set until the synchronization counter is loaded, decremented and is either zero or stored back. In order to set SLOCK, a new instruction is proposed that uses the existing control logic for the common 'load byte' instruction (`ldb`). This instruction, `lds` - load synchronization counter, is similar to `ldb` but additionally clears the zero bit and then sets SLOCK at the beginning of its bus access. SLOCK is then reset by generating a special control line whenever the zero bit is set or the `stb` operation completes. The comparator basically detects if the same synchronization counter is being accessed by both processors. If the addresses are not equal, the CMP signal remains zero and thus $\overline{\text{SYNC\_OK}}$ remains zero. On the other hand, if $\overline{\text{SYNC\_OK}}$ is set, the processor that is just beginning its bus access must stall until the other processor has reset SLOCK.

Except for the rare case when both processors access the same synchronization counter simultaneously, our design basically reduces the problem of atomicity to a problem of bus contention. Thus, whenever one processor accesses a synchronization counter, the bus penalty for the other processor is simply one cycle due to the data load of the `lds` instruction plus, if the synchronization fails, another two cycles for the data store of the `stb` instruction.

4.2 Access to the LCV

Conceptually, having both processors simultaneously access the LCV creates two atomicity problems. First, each push/pop on/from the LCV by the Main-processor potentially interferes with a push on the LCV by the Inlet-processor (i.e., `FORK-POST` interference). Second, a `SWAP` can execute during a `POST` to the just terminating, yet still running frame. This operation could cause the problem that the `POST` might push a thread on the just emptied LCV (i.e., `SWAP-POST` interference).

Since the LCV and the RCV (i.e., the continuation vector that is about to become the LCV) are actually in different memory locations, it is possible to alleviate the `FORK-POST` interference entirely and simplify the design by having `FORK`s push threads only on the LCV and `POST`s push threads only on the RCV. There are two possible implementations of this scheme when inlets post threads that belong to the currently running frame (i.e., `fp=ifp`) to the RCV. The first is to move these threads to the LCV so that the current quantum can continue. The second is to schedule these threads during the next quantum. However, both methods will degrade the performance because more than 50% of all threads posted are for the currently running activation and 14%-32% of all threads are enabled by `POST`s during a quantum [1]. Therefore, the side-effect is either a large overhead of moving the posted threads to the LCV or shorter quanta and thus more frequent context switches that are expensive.

To eliminate the aforementioned overhead, the proposed method, which leads to a slightly more complex design, allows the LCV to be shared between threads and inlets.

4.2.1 `FORK-POST` Interference

The proposed design implements the LCV as a *queue-stack*. That is, the Inlet-processor always pushes threads on the bottom of the stack whereas the Main-processor pushes/pops threads on/from the top of the stack. This can be implemented by having the pointer to the top of the stack (`lcv`) in a Main-processor register and the pointer to the bottom of the stack, called `lcvend`, in an Inlet-processor register. The advantage of the queue-stack

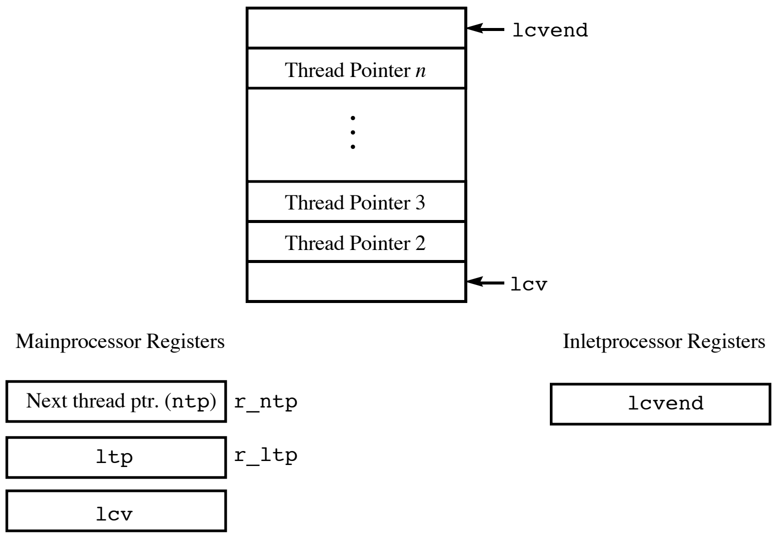

Figure 4: Queue-stack representation of the LCV.

is the two processors do not have to share `lcv` in order to push/pop threads. Figure 4 illustrates the implementation of the queue-stack.

One problem with the queue-stack is whenever a thread is pushed on the bottom of the stack, the thread pointer must be inserted in between the leave-thread pointer, `ltp`, and the pointer to the last computational thread in the stack (i.e., Thread Pointer *n*). This is because the leave-thread implements the switching to a new frame by executing a `SWAP` and thus must be the last thread executed within the quantum. This can be accomplished by having the Inlet-processor push a thread on the bottom of the stack and then exchange the bottom two thread pointers. However, this exchange operation would require four additional cycles for each post compared to the original stack used by TAM. To eliminate the additional cost, `ltp` is kept in a Main-processor register called `r_ltp` rather than at the bottom of the stack. As soon as `lcv` from the Main-processor and `lcvend` from the Inlet-processor are equal, `ltp` is moved to `r_ntp`, which can then be scheduled by the `cdbp` instruction.

The queue-stack representation of the LCV eliminates the `FORK-POST` interference; therefore, the only overhead resulting from this implementation is the bus contention. For the Main-processor, this cost depends on whether the Inlet-processor posts to an idle (7 cycles), a ready (5 cycles), or a running (2 cycles) frame. These costs represent only the time actually spent on the MICBus to load and store data [4].

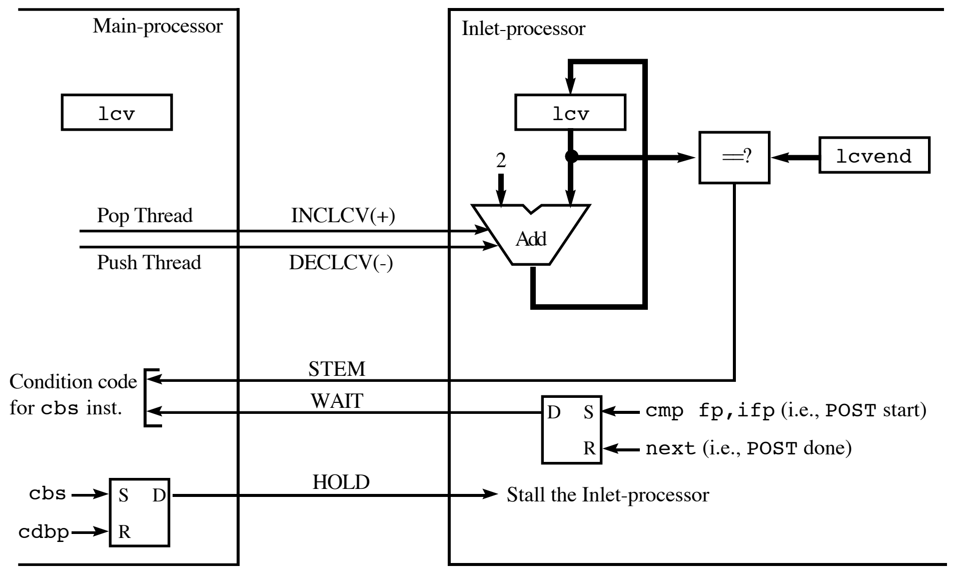

Figure 5: The design for `SWAP-POST` interference.

4.2.2 `SWAP-POST` interference

`SWAP` occurs only in the execution of a leave-thread. Thus, it is initiated when the LCV becomes empty (i.e., `lcv=lcvend`). To maintain atomicity between `SWAP` and `POST`, both processors must know the exact state each other is in. This is accomplished by having a set of control lines between the two processors. Figure 5 illustrates the proposed scheme that minimizes the changes needed to the original SPARC.

In the proposed design, identical copies of `lcv` exist in both processors. This facilitates the testing of whether the stack is empty without having a dedicated bus between the processors. In order to maintain coherence between the two `lcv`s, both are controlled only by the Main-processor by means of two control lines—INCLCV and DECLCV. This is done by having a 16-bit adder in the Inlet-processor with the sole purpose of incrementing or decrementing `lcv`. Thus, whenever `lcv` in the Main-processor is incremented/decremented, `lcv` in the Inlet-processor is also incremented/decremented during the same cycle. This allows both `lcv`s to appear identical. INCLCV is set if the decoded instruction is `cdbp` (used to pop threads) and the decremented entry count is not zero (see Section 4.1). DECLCV is set if `std` is decoded and it addresses `lcv`. The `std` instruction, which was proposed in [2], has post-decrement capability that allows pushing of threads faster. Both control lines are asserted during the execute cycle.

The STEM signal, which indicates whether or not the LCV is empty, must be available to the Main-processor at the end of the execute cycle of a `lcv` update. This is because when the entry count is zero for the `cdbp` instruction, the next stage of the pipeline must know if the stack has been emptied or not. If it is empty, `ltp` is moved into `r_ntp` during this stage instead of popping a non-existing thread pointer from the empty stack. This allows the leave-thread to be scheduled next whenever `lcv=lcvend.`

As long as the Inlet-processor is not posting a thread, the leave-thread executes a `SWAP` and begins the process of switching to the next enabled activation. However, a problem occurs if an inlet posts a thread to the currently running frame (`fp=ifp`) when `lcv` and `lcvend` are equal. The newly posted thread must be allowed to continue since it may fork other threads. Even if `POST` is not for the currently running frame, it might be for the next frame to which the Main-processor wants to swap. Thus, it is important that the Main-processor suspends the `SWAP` operation until the Inlet-processor has terminated any `POST` operation. Both cases basically require the same action to be taken, i.e., putting the Main-processor in a wait state. Therefore, a control signal WAIT is used to inform the Main-processor whether or not the Inlet-processor is executing a `POST`.

The WAIT signal is only set during the part of the `POST` sequence which must execute without interfering with `SWAP`. This part starts with a test to see if the message is for the currently running frame (i.e., `cmp fp,ifp)`, and ends after either a thread has been posted to the LCV or RCV or a new frame has been enqueued as a result of posting to RCV. WAIT is reset by the `next`[1] instruction. The HOLD signal on the other hand is used to irreversibly terminate the current frame and to stall the Inlet-processor. Once `SWAP` has finished, the HOLD signal is reset to allow the Inlet-processor to continue its normal execution.

In order to ensure atomicity between `POST` and `SWAP`, the first instruction in the leave-thread must check both STEM and WAIT, and take different actions depending on conditions. Table 1 specifies the four cases. If the stack is empty (STEM=1) and no inlet is trying to post a thread (WAIT=0), the HOLD signal is set and irreversibly terminates the current activation by letting the leave-thread continue, which ends with a `SWAP`. Basically this invalidates the `fp`, which resides on both processors, and prohibits the Inlet-processor from executing any inlets as long as the Main-processor is actually in the process of swapping to the next frame. If the stack is empty (STEM=1) but there is an inlet trying to post a thread (WAIT=1), the Main-processor is stalled until WAIT is reset.

[1] In the Inlet-processor, the '`next`' assmebly instruction is always the last instruction of an inlet code. It simply dispatches to the next inlet or waits until a new message has arrived. For more details see [4].

Table 1: Condition codes for the `cbs` instruction on the Main-processor.

	WAIT = 0	WAIT =1
STEM = 0	<u>Case 1</u> The LCV is no longer empty and there is no inlet trying to post a thread. <u>Action</u> Terminate the leave-thread and continue with the current activation by moving the posted thread into `r_ntp` and executing the `cdbp` instruction.	<u>Case 2</u> The LCV is no longer empty and there is an inlet trying to post a thread. <u>Action</u> Same as in the case STEM=WAIT=0.
STEM = 1	<u>Case 3</u> The LCV is empty and there is no inlet trying to post a thread. <u>Action</u> Assert HOLD and continue with the leave-thread.	<u>Case 4</u> The LCV is empty but there is an inlet trying to post a thread. <u>Action</u> Stall until WAIT is deasserted then go to either Case 1 or Case 3.

If the stack is not empty (STEM=0), the newly posted thread pointer is loaded into `r_ntp` and then the execution continues from the new thread. This simply means the `fp` stays valid and thread execution continues as usual.

A new TL0 instruction '`CHECK`' is proposed to accomplish the aforementioned task. `CHECK` either pops the newly posted thread or sets HOLD and lets the leave-thread continue with `SWAP` depending on STEM and WAIT. In order to map `CHECK` onto the SPARC, a new conditional assembly instruction, `cbs` - *conditional branch and stall*, is proposed. The two control lines STEM and WAIT are used as the condition code. Within format2 of the SPARC8 instructions (op=0) op2=5 has not yet been implemented [9], so it can be used for this purpose. The `cbs` instruction basically uses the existing logic for a conditional branch, except that it has the additional capability to stall the Main-processor, and it takes its branch condition information from STEM and WAIT. The mapping of `CHECK` on the SPARC is as follows:

```
CHECK
cbs,a   stop          ; Branch to stop[1] or continue with the leave-thread, and thus SWAP  (1 or 2)
lduh    [lcv],r_ntp   ; Use the delay slot to pop newly posted thread into r_ntp (2)
```

Therefore, `CHECK` takes either 2 cycles if the leave-thread continues, or 3 cycles (plus 4 cycles for `STOP`) if the leave-thread is terminated.

1 The assembly code at '`stop`' is the TL0 `STOP`.

As soon as the leave-thread terminates, HOLD must be reset. This can be easily accomplished by the `cdbp` instruction, which will occur at the end of the `SWAP`. Although `cdbp` occurs at various points during program execution, it is not harmful to reset HOLD at these times, since HOLD will be zero anyway.

5. System Impact Analysis

This section presents an analysis of the overall performance improvement using an Inlet-processor to handle messages. This is done by evaluating the impact the proposed design modifications have on the processor time. The data used for this analysis was from the results of several experiments by the TAM-group at UC Berkeley[1]. Our analysis estimates the improvement in the average *clock cycles per TAM* instruction (CPT) for the modified design and compares it against the average CPT of an original 64-node CM-5 [1] for two benchmarks, Gamteb and Paraffins. The overall results of the comparison is presented in Table 2.

Table 2: Distribution of processor time, original and modified.

	Gamteb, % of original processor time			Paraffins, % of original processor time		
	Original	Modified		Original	Modified	
		Main-proc.	Inlet-proc.		Main-proc.	Inlet-proc.
Overhead	-	6.80%	1.10%	-	7.21%	0.06%
Memory	13.97%	5.96%	8.01%	14.01%	4.67%	9.34%
Operands	8.09%	8.09%	-	7.64%	7.64%	-
ALU	5.15%	5.15%	-	1.27%	1.27%	-
Messages	5.88%	-	3.89%	0.64%	-	0.35%
Heap	33.82%	-	24.12%	49.04%	-	37.20%
Control	27.21%	17.75%	6.67%	20.38%	14.20%	3.43%
Atomicity	5.88%	-	-	7.01%	-	-
Total	100.00%	43.75%	43.79%	100.00%	34.99%	50.38%
Original CPT	13.6	5.95	5.96	15.7	5.49	7.91
Speedup		2.28			1.98	
Workload	100.00%	49.98%	50.02%	100.00%	40.99%	59.01%

The table shows the TL0-instruction types and their respective percentages for both the original and the modified design. For the modified design, the workload distribution for the two processors is also shown and it includes all the effects resulting from the proposed design—efficient access of synchronization counters, access and representation of the LCV, the elimination of polling, the introduction of `cdbp` and `std` instructions, as well as the dispatch of `SEND`s and heap operations to the Inlet-processor.

1 The sources are: 1) TAM - A Compiler Controlled Threaded Abstract Machine[1] and 2) Instruction-mix for several benchmark programs available at a UCBerkeley ftp-site (ftp.cs.berkeley.edu:/ucb/TAM/sethg/dists.tar.Z).

TL0 instructions are divided into various categories. *Overhead* describes the cost incurred due to MICBus-contention and the time needed to dispatch all message and heap instructions to the Inlet-processor. *Memory* is a result of the penalty cost from an assumed cache-miss rate of 5%. *Operands* also assumes a 5% cache-miss rate for bringing operands into the ALU. *ALU* simply represents the time spent executing arithmetic and logic instructions. *Messages* depict the cost of all explicit `SEND` and `RECEIVE` instructions. *Heap* combines all heap related costs, such as allocation and heap accesses (such as fetching and storing heap-elements). *Control* reflects the time spent for all thread scheduling instructions, such as `FORK` and `POST`. Finally, *atomicity* represents the cost of polling. In the modified design, polling is no longer required since the network interface is integrated into the Inlet-processor[1]. As can be seen, the largest improvement comes from *heap*, *control* and *messages* as well as from the elimination of the overhead *atomicity* (polling). In the following subsections, the results for *control*, *messages*, *heap*, and *overhead* are discussed in more detail.

5.1. Distribution of Control Instructions

Table 3 shows the distribution of control instructions between the two processors. The instructions are divided into the ones executing only on the Main-processor and `POST` which executes only on the Inlet-processor. The improvement in cycle costs for `FORK`, `SWITCH`, and `STOP` is due to `cdbp` and `std` assembly instructions [2]. The new `CHECK` instruction, which performs a conditional test on WAIT and STEM signals, adds to the cycle cost for the proposed design. The modifications to the `SWAP` instruction, which were necessary due to the `SWAP-POST` interference also add to the cycle cost.

To analyze the effect of these changes the metric, average clock *cycles per TL0 instruction* (CPT), is considered, which is obtained by multiplying the instruction frequency of each instruction type by its cycle cost and summing up these products. This is then used to compute how much the control instructions contribute to the CPT on each processor as compared to the overall CPT of the original execution on CM-5. As can be seen, the proposed modifications lead to 73/27 and 82/18 distributions of the workload due to control instructions for Gamteb and Paraffins, respectively. The total control overhead has been reduced by 11%/15% (Gamteb/Paraffins).

[1] It is assumed that the network interface simply sets an appropriate control signal when a message arrives. The Inlet-processor receives the message as soon as the current inlet has completed [7].

Table 3: Distribution of processor time due to control instructions on Main-processor and on Inlet-processor

Main-processor	Cycle cost		Gamteb in % of TL0-instr.		Paraffins in % of TL0-instr.	
	Original	Modified	Original	Modified	Original	Modified
`FORK`						
fall-through	0	0	1.11%		1.58%	
unsynchronizing branch	1	1	0.91%		3.86%	
synchr. branch - successful	4	5	2.34%		2.95%	
- failed	13	9	6.14%		10.97%	
unsynchronizing push	5	4	0.04%		0.00%	
synchr. push - successful	10	9	4.70%		2.85%	
- failed	7	7	3.73%		5.19%	
`SWITCH`						
unsynchronizing branch	3	3	1.48%		2.87%	
synchr. branch - successful	6	7	1.28%		0.13%	
- failed	15	11	2.26%		0.11%	
unsynchronizing push	7	6	1.54%		5.44%	
synchr. push - successful	12	11	0.54%		0.00%	
- failed	9	9	2.20%		0.00%	
`SWAP`						
basic	26	31	0.39%		0.04%	
per extra 4 threads	12	12	0.17%		0.00%	
`STOP`	5	4	2.49%		2.85%	
`SINIT` (Init. of entry counters)	4	4	1.74%		8.42%	
`CHECK`						
continued	-	2	-	0.39%	-	0.04%
terminated (plus `STOP`)	-	7	-	0.05%	-	0.01%
Computed contribution			2.78	2.42	3.21	2.69
Inlet-processor						
`POST` (without cost for synchr.)						
to idle frame	18	17	1.63%		0.13%	
to ready frame	14	13	0.98%		0.04%	
to running frame	7	7	2.61%		5.84%	
`POST` (cost for synchronization)						
failed	7	7	2.74%		1.13%	
successful	5	5	2.52%		2.67%	
Computed contribution			0.93	0.91	0.65	0.65
Total computed contribution due to control instructions			3.71	3.29	3.86	3.28
Original CPT			13.6		15.7	
Original CPT due to control instructions			3.7		3.2	
Normalized CPT on Main-processor due to control insts.†			3.7	2.41	3.2	2.23
% of total CPT			27.21%	**17.75%**	20.38%	**14.20%**
Normalized CPT on Inlet-processor due to control insts.†			-	0.91	-	0.54
% of total CPT			-	**6.67%**	-	**3.43%**

† Due to a slight discrepancy in the two data sources, the computed results were normalized by the ratio of original CPT due to control instructions and computed contribution to control instructions.

5.2. Messages and Heap

Messages consist of the cost due to two TL0-instructions: `SEND` and `RECEIVE`. *Heap*, on the other hand, includes the times spent on allocation, control, and access of heap using `IFETCH` and `ISTORE` (which are special forms of `SEND`). The savings achieved for *messages* and *heap* are a result of integrating the Network Interface into the Inlet-processor. This eliminates the expensive uncached loads/stores, which are necessary for `RECEIVE/SEND` in the CM-5 (the network interface is on the MBus which connects cache, main memory, and I/O devices) [1]. Moreover, having heap elements stored in the Inlet Cache allows the Inlet-processor to service heap requests without disturbing the thread execution in the Main-processor. Since all `SEND`s and heap operations are dispatched from the Main-processor to the Inlet-processor, their reduced cost is completely shifted to the Inlet-processor (see Table 2). There is however some penalty incurred due to dispatching to the Inlet-processor, which is included in *overhead* (see Section 5.3).

5.3. Overhead

Overhead contains all the costs incurred specifically due to the introduction of the Inlet-processor. These costs include the extra dispatch time required for messages and heap operations by the Main-processor and stalls caused by both processors due to MICBus contention and `SWAP-POST` interference. Table 4 shows the breakdown of the overhead for both the Main-processor and the Inlet-processor.

Table 4: Overhead cost on Main-processor and on Inlet-processor

Main-processor	Cycle cost	Gamteb in % of TL0-instr.	Paraffins in % of TL0-instr.
Stalls due to MICBus-accesses of Inlet-proc.			
`POST` (without cost for synchronization)			
to idle frame	7	1.63%	0.13%
to ready frame	5	0.98%	0.04%
to running frame	2	2.61%	5.84%
`POST` (cost for synchronization)			
successful	1	2.52%	2.67%
failed	3	2.74%	1.13%
`RECEIVE` (local and remote)	2	4.03%	0.28%
Stalls due to `CHECK` waiting for `POST` to finish	11.25/7.26	0.44%	0.05%
Contribution to CPT		0.53	0.34
% of original CPT		3.90%	2.17%
Other overhead in % of original CPT			
message-dispatch cost		2.35%	4.97%
heap-dispatch cost		0.55%	0.06%
Main-processor overhead in % of original CPT		**6.80%**	**7.21%**
Inlet-processor			
Stalls due to `CHECK`			
continued	2	0.39%	0.04%
Stalls due to `SWAP`			
basic	31	0.39%	0.04%
per extra 4 threads	12	0.17%	0.00%
CPT due to overhead on Inlet-processor		0.15	0.01
Original CPT		13.6	15.7
Inlet-processor overhead in % of original CPT		**1.10%**	**0.06%**

The costs for POST and RECEIVE include only the cycles when the Inlet-processor is actually on the MICBus. If POST has already started, CHECK always waits for it to finish. The penalty for this operation can be computed by multiplying the frequency of CHECK (see Table 3) with the average number of cycles spent on POST without the cost for synchronization when WAIT is set. This average number varies from benchmark to benchmark depending on the number of POSTs to idle, ready, or running frames. There is also an overhead cost for the Inlet-processor since it has to stall while the Main-processor is executing a SWAP. Thus, the Inlet-processor sees the full penalty cost of CHECK (when continued) and SWAP instructions. CHECK, when terminated, never sets HOLD, thus it does not stall the Inlet processor.

Notice that it is safe to assume no additional costs exist for the Inlet-processor when the Main-processor accesses synchronization counters or the LCV. This is because the Inlet-processor fetches instructions from the Inlet Cache. Thus, it needs to access the MICBus only for frame data (in the Common Cache), which does not occur very often compared to the total processor time. In contrast, the Main-processor uses the MICBus not only to access the frame, but also to fetch its instructions. Since the Bus Arbiter gives each processor equal priority, it is highly likely that the Inlet-processor will always be granted the bus immediately.

The results show that the cost of swapping frames for the Inlet-processor varies depending on the benchmark. For Gamteb, it takes more than 2% of the processor time on the Inlet-processor whereas in Paraffins this overhead is almost zero due to the relatively low number of SWAPs.

6. Conclusion

This paper proposed architectural modifications required to solve the atomicity problem that occur as a result of introducing a separate processor to handle messages in the context of TAM. The proposed design maximizes the performance by balancing the load between the two processors and minimizes the overhead cost by providing specific architectural support. This has been achieved without disturbing the original functionality of the SPARC processor. Our analysis indicates a separate processor with integrated network interface compensates for a major drawback of the implementation of fine-grain programs—the message overhead. The resulting speedup is very encouraging, e.g., 2.3 for Gamteb on a 64-node CM-5.

Since the Inlet-processor is an independent processor purely custom designed to primarily support TAM, we were free to design and modify it arbitrarily in a reasonable manner according to our needs. However, the design is not restricted to the TAM execution model. The conceptual implications can be applied to any other fine-grain execution models.

Acknowledgment

The authors are grateful to Ellen Spertus for her insightful comments on an earlier version of this paper.

References

[1] Culler, D. E. *et al.*, "TAM—A Compiler-controlled Threaded Abstract Machine," Journal of Parallel and Distributed Computing, June 1993.

[2] Kotikalapoodi, S. V., Lee, B., Lu, S. L., and Hurson, A. R., "Architectural Support for Fine-grain Multithreading on Stock Processors," To appear in the Journal of Mini and Micro Computers.

[3] Lee, B. and Hurson, A. R., "Dataflow Architectures and Multithreading," IEEE Computer, August, 1994, pp. 27-39.

[4] Metz, D., "Analyzing the Benefits of a Separate Processor to Handle Messages for Fine-grain Multithreading," Technical Report TRECE95.03, Oregon State University, ECE Department.

[5] Nikhil, R. S., "ID Language Reference Manual Version 90.1," Technical Report CSG Memo 284-2, MIT Lab for Comp. Science, Cambridge, MA, 1991.

[6] Nikhil, R. S., Papadopoulas, G. M., and Arvind, "*T: A Multithreaded Massively Parallel Architecture," Proc. 19th Annual Int'l. Symposium on Computer Architecture, 1992, pp. 156-167.

[7] Papadopoulos, G. M. et al., "*T: Integrated Building Blocks for Parallel Computing," Supercomputing 93, Portland, Oregon, November 19, 1993.

[8] Schauser, K. E., Culler, D. E., and von Eicken, T., "Compiler-controlled Multithreading for Lenient Parallel Languages," Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA, August 1991.

[9] Sparc International, Inc., Menlo Park, California, "The SPARC Architecture Manual, version 8," Prentice Hall, 1992.

[10] Spertus, E. *et al.*, "Evaluation of Mechanisms for Fine-grained Parallel Programs in the J-Machine and the CM-5," Proceedings of the 20th Int'l Symposium On Computer Architecture, San Diego, CA, May 1993.