AN ABSTRACT OF THE THESIS OF

David Metz for the degree of Master of Science in Electrical and Computer Engineering presented on April 28, 1995. Title: Interface Design and System Impact Analysis of a Message-Handling Processor for Fine-Grain Multithreading.

Redacted for Privacy

Abstract approved: _____
Ben Lee

There appears to be a broad agreement that high-performance computers of the future will be Massively Parallel Architectures (MPAs), where all processors are interconnected by a high-speed network. One of the major problems with MPAs is the latency observed for remote operations. One technique to hide this latency is multithreading. In multithreading, whenever an instruction accesses a remote location, the processor switches to the next available thread waiting for execution. There have been a number of architectures proposed to implement multithreading. One such architecture is the Threaded Abstract Machine (TAM). It supports fine-grain multithreading by an appropriate compilation strategy rather that through elaborate hardware. Experiments on TAM have already shown that fine-grain multithreading on conventional architectures can achieve reasonable performance.

However, a significant deficiency of the conventional design in the context of fine-grain program execution is that the message handling is viewed as an appendix rather than as an integral, essential part of the architecture. Considering that message handling in TAM can constitute as much as one fifth to one half of total instructions executed, special effort must be given to support it in the underlying hardware.

This thesis presents the design modifications required to efficiently support message handling for fine-grain parallelism on stock processors. The idea of having a separate processor is proposed and extended to reduce the overhead due to messages. A detailed hardware is designed to establish the interface between the conventional processor and the message-handling processor. At the same time, the necessary cycle cost required to guarantee atomicity between the two processors is minimized. However, the hardware modifications are kept to a minimum so as not to disturb the original functionality of a conventional RISC processor. Finally, the effectiveness of the proposed architecture is analyzed in terms of its impact on the system. The distribution of the workload between both processors is estimated to indicate the potential speed-up that can be achieved with a separate processor to handle messages.

Interface Design and System Impact Analysis of a Message-Handling Processor for Fine-Grain Multithreading

by

David Metz

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 28, 1995
Commencement June 1995

Master of Science thesis of David Metz presented on April 28, 1995

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Chair of Department of ~~Electrical and Computer Engineering~~

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

David Metz, Author

This thesis is dedicated to my Lord Jesus Christ.

ACKNOWLEDGMENTS

Throughout my studies I have received encouragement from many people. Heart filled thanks go to my parents, my sisters and my little brother for their prayers and moral support and for keeping me in touch with the old world.

Special thanks go to my major professor, Dr. Lee, for helping guide my efforts and for challenging me to develop new ideas. His classes, thoughtful advice, and availability helped me to see the big picture. Many thanks to Dr. King for his flexibility and to Dr. Lu and Dr. Peterson for their willingness to take out time for this defense.

I am very grateful to the Fulbright-Kommission for providing the much needed grants for my entire study period in the United States. Without their support this dream would never have come true.

Deep thanks to my special friends - Mark and Christine, JP (Johan), Beth, Jim, Todd and Becky, Pastor Phil, Steve, Jerome and Cynthia, Paul, Stephen (Yee-ha), Chris, Sheena, Peggy, Susanna, Yvonne, Frank, Dale, Slant, Virgil and CaraLynn, and Kent for making my stay at beautiful Oregon a pleasant and unforgettable experience and for showing me "the real America".

Thanks to all my fellow students who have always been there for help. We all learned new ideas from each other.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

LIST OF APPENDIX FIGURES

Figure Page

Interface Design and System Impact Analysis of a Message-Handling Processor for Fine-Grain Multithreading

1. INTRODUCTION

1.1 Motivation

In the past years, the demand for high performance stimulated the design and development of parallel computers. There appears to be a broad agreement that high-performance computers of the future will be Massively Parallel Architectures (MPAs) [1]. It is not surprising that today's MPAs commonly use von-Neumann style processors which have been and still are the most suitable technology for the traditional single-processor architectures. Over the years, von-Neumann style processors have been highly optimized by a variety of methods, such as pipelining, multiple functional units, and vector units [2]. Unfortunately, these highly efficient processors are often less than ideal for large-scale parallel architectures. Their major drawback is the lack of mechanisms that specifically support scalable interprocessor communication and synchronization [3] and their limited ability to support the exploitation of parallelism in programs. For example, consider a parallel computer where a processor has to fetch the contents of the memory on a remote processor. The requesting processor has to idle while waiting for the reply. Furthermore, if synchronization is required because the requested argument is not yet available, the idle time for the waiting processor is of *unbounded* latency. Thus, any large-scale parallel architectures must rely on latency-hiding techniques in order to be scalable [4].

The classical solution for masking memory latency is to provide a cache holding copies of remote locations. Although this helps, remote requests of unbounded latency (synchronizing loads) are still not dealt with [1]. An alternative technique for hiding memory latency is to multiplex amongst many threads of a program code waiting to be executed on each processor. This concept is called *multithreading*. Whenever a thread issues a remote load request, the processor switches the execution to another thread. The requesting thread can continue as soon as the reply has arrived. However, the effectiveness of multithreading depends on rapid support of context (thread) switching [5].

An alternative to the control-flow concept of computation (which includes the von-Neumann concept) is the dataflow model of execution. Conceptually, instructions are executed as soon as their operands are available. Theoretically, operands are not stored in memory, but instructions produce tagged tokens, where the tag indicates the destination of the token. The dataflow model is attractive for parallel processing, because it exposes all forms of parallelism down to the instruction level [5]. On the other hand, the dataflow approach has some drawbacks which prevent it from being a practical alternative to its

control-flow counterpart. One is the heavy overhead involved in matching tokens. Another is the inefficiency of the dataflow instruction cycle [5].

Multithreading can combine features of both execution models. It provides thread-level context switching in a dataflow fashion and sequential instruction-scheduling within threads in a control-flow fashion. Thus, besides tolerating unpredictable latencies, multithreading exploits the high, sequential efficiency achieved on conventional processors. The level of parallelism exposed is higher when the threads are shorter.

One such hybrid model, called Threaded Abstract Machine (TAM), has been developed at UC Berkeley. TAM supports fine-grain multithreading by an appropriate compilation strategy rather than through elaborate hardware [6]. It provides a means to map programs represented by dataflow graphs to conventional hardware and yet obtain reasonable performance [7]. On the other hand, experiments on TAM indicate a basic mismatch between the requirements for fine-grain parallelism and the underlying conventional architecture. This suggests that considerable improvement is possible through hardware support [8]. This has been further confirmed in a study that improves the execution of TAM control-instructions by incorporating a special instruction in the ISA of the SPARC processor [9]. Another significant deficiency of the conventional design in the context of fine-grain program execution is that the message handling is viewed as an appendix rather than as an integral, essential part of the architecture. However, considering that message handling in TAM can constitutes as much as 22% - 45% of total instructions executed, special effort must be given to support it in the underlying hardware. One project at MIT suggested the usage of a separate processor to handle messages [1].

In the light of the aforementioned discussion, this work presents the design modifications required to efficiently support message handling for fine-grain parallelism on stock processors. The idea of having a separate processor is proposed and extended to reduce the overhead due to messages. A detailed hardware is designed to establish the interface between the conventional processor and the message-handling processor. At the same time, the necessary cycle cost required to guarantee atomicity between the two processors is minimized. However, the hardware modifications are kept to a minimum so as not to disturb the original functionality of a conventional RISC processor. Finally, the effectiveness of the proposed architecture is analyzed in terms of its impact on the system. The distribution of the workload between both processors is estimated to indicate the potential speed-up that can be achieved with a separate processor to handle messages.

Although the discussion is based on the SPARC processor, the design issues discussed in this thesis apply to other RISC processors as well.

1.2 Thesis Organization

Chapter 2 gives a short introduction to the conceptual differences between the dataflow and the control-flow execution model. This is followed by a detailed description of TAM.

Chapter 3 summarizes the thesis of S. Kotikalapoodi [9]. Some of the architectural proposals from his work are described and integrated in the following discussions and analysis.

Chapter 4 proposes two specific hardware solutions to guarantee atomicity between the Main-processor and the processor handling messages (i.e., the Inlet-processor). A number of other issues concerned with processor interaction are discussed. Finally, a design of a complete processor node is proposed.

Chapter 5 analyzes the benefits of having a separate processor to handle messages in the context of the proposed processor node design. The distribution of the workload between the two processors is estimated by means of the metric *clock cycles per TAM instruction* (CPT) on each processor.

Chapter 6 discusses the general significance of the results and other areas of possible future research are pointed out.

1.3 Typeface Conventions

In order to avoid any confusion, three different font-styles are used. The typeface conventions are as follows:

Names of control lines are in	TIMES NEW ROMANS CAPS.
TL0-instructions (TAM pseudo-machine code) are in	`COURIER CAPS.`
SPARC assembly instructions are in	`lower case courier.`

2. TAM - THREADED ABSTRACT MACHINE

2.1 Background

Currently, programs are usually executed according to the *control-flow* execution model since all common architectures are optimized for and based on this model. Control-flow simply means that a program counter triggers the execution of instructions one after the other. In contrast, in the alternative execution model, called *dataflow*, the execution of an instruction is initiated by the availability of the data. Figure 2.1 illustrates the two different concepts by a simple example. The arcs indicate data dependencies. A, B, E are assumed to already exist.

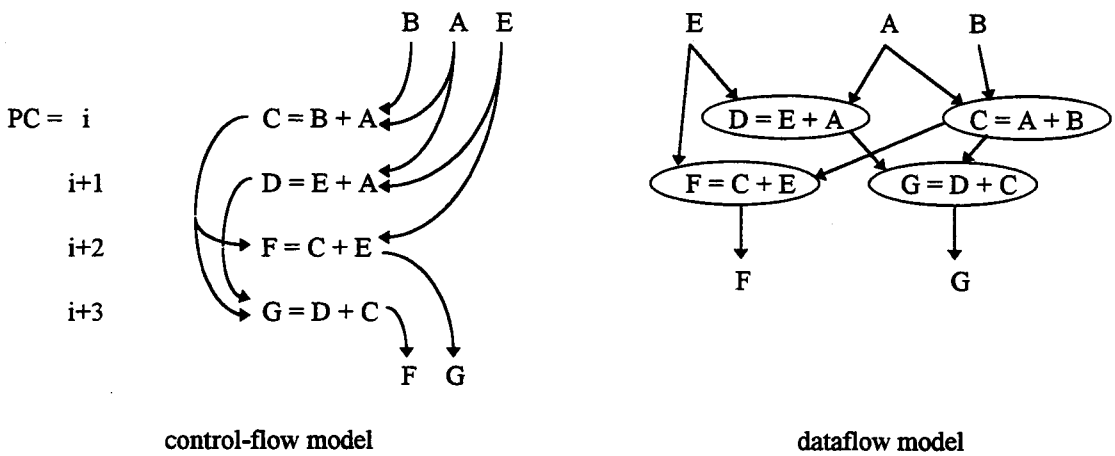

Figure 2.1 Comparison of control-flow and dataflow execution model

What makes the dataflow method attractive is that synchronization of parallel activities is implicit and self-scheduling [5]. The scheduling of instructions is only constrained by the data dependencies. Thus, the dataflow program representation exposes all the possible instruction level parallelism in a program. On the other hand there are some serious problems with the dataflow model compared to its control-flow counterpart [5]. One drawback is the substantial cost required to detect enabled instructions (i.e., all the data needed is available) and to communicate the results.

In practice, today's commonly used parallel architectures rely solely on processors, which have been optimized for the control-flow execution model. Programs are usually divided into several large blocks of instructions and then spread over the processors. There are two important issues that affect the performance of this practice. First, if a program block on a certain processor requires synchronization, it

needs to wait for two or more blocks to terminate. Second, if an instruction needs to fetch data from another processor, it needs to wait for the complete time it takes for the request to travel to the remote processor, to be serviced, and finally to return the requested data.

To eliminate these high latencies, a method called *multithreading* was introduced. In multithreading, each processor[1] holds multiple partitions of program code, called *threads*. Whenever a thread has to busy-wait (i.e., it has requested data from a remote processor and waits for the reply), another thread starts executing instead. Thus, the aforementioned latencies are hidden and all processors can be fully utilized. Basically, threads can be of arbitrary length from several (fine-grain) to several thousand (coarse-grain) instructions.

Conceptually, multithreading is a hybrid execution model since it combines aspects of the control-flow and the dataflow execution model. Within the threads, instructions are scheduled in a control-flow fashion whereas the threads themselves are scheduled in a dataflow fashion. (Thus, multithreading exploits both the execution efficiency of the control-flow model and the exposure of parallelism of the dataflow model.)

There are a variety of ways to implement multithreading. Two fundamental dimensions are: First, multithreading can be implemented for all possible grain-sizes, from instructions (fine) to large blocks of code (coarse). Second, it can be realized either in software, hardware, or arbitrary combination of the two. One such hybrid execution model is Berkeley's TAM - Threaded Abstract Machine [6]. In the following sections TAM is presented and discussed in more detail.

2.2 Concepts of TAM

TAM is a fine-grained parallel execution model that demonstrates, how the dataflow concept can be mapped efficiently on conventional parallel machines. TAM defines a machine language of parallel threads, called TL0, which is completely self-scheduled. TL0 makes it possible to run programs represented by dataflow graphs on conventional architectures.

A program in TL0 consists of a collection of *code-blocks*, which contain *threads* and *inlets*[2] (see Figure 2.2). A code-block is typically represented by a function or a loop-body in the original high-level language. When a code-block is invoked, an *activation frame* is dynamically allocated. The activation frame is a key element of the TAM execution-model.

The activation frame provides the resources required for synchronization and scheduling of threads as well as storage for local variables used by the invoked code-block. Once allocated to the memory of a specific processor node, an activation frame remains in the memory for the rest of its

[1] In the following, a parallel architecture is assumed.
[2] Inlets are code to specifically handle messages. They are explained below.

execution. After a frame is allocated and initialized, arguments are sent to the frame. Then, its execution can start. Since a caller can invoke several child code-blocks (callees) the call structure is represented by a tree rather than a stack (see Figure 2.2). This structure is dynamic since the occurrence of various conditions for a code-block invocation cannot be determined in advance. The child frames can be distributed over several processors and can all be activated concurrently. Although the frames are the basic unit of work distribution over the processors fine-grain parallelism is not wasted. This is because within a frame, thread parallelism compensates communication latency. Moreover, within each thread, instruction-level parallelism can be exploited (in superscalar processors).

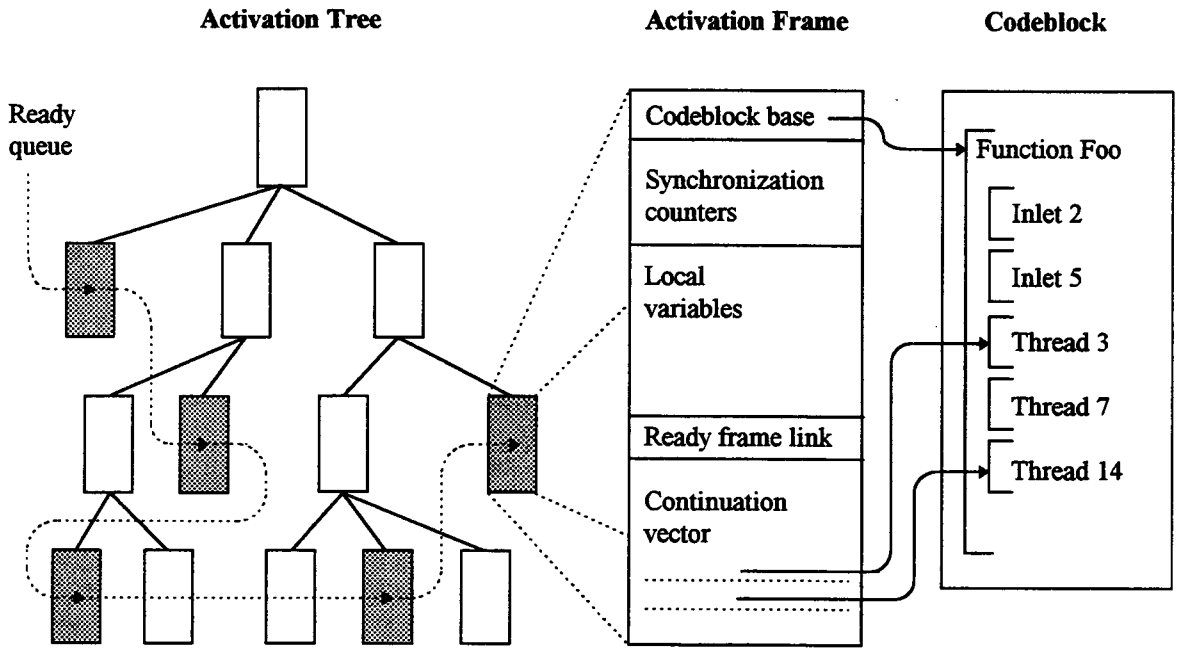

Figure 2.2 TAM activation tree and embedded scheduling queue

A frame can be in one of three states. An *idle* frame has no enabled threads. It becomes *ready* as soon as one thread is enabled (i.e., the data needed to execute this thread has become available). A scheduled frame is considered *running* or *resident* and is executed until it has no enabled threads.

Since each argument sent to a code-block can enable a thread, there can be several ready frames on one processor. The execution sequence of ready frames waiting in a queue to be scheduled is determined by a linked-list. Several threads within a single frame are scheduled by the sequence of their starting addresses (relative to the frame) in the *continuation vector* (CV). After activating a frame, all threads are executed according to the CV until no enabled threads remain. The last thread, called *leave-*

thread, deactivates the current frame and switches to the next frame in the ready queue. The number of threads executing during a single residency of a frame is called a *quantum*. Usually, threads in a quantum are related since they are from the same code-block. The advantage of having several related threads in a quantum is that it promotes locality since registers can potentially be carried over from thread to thread and since the processor can work in one frame as long as possible without context switches.

2.2.1 Storage

In terms of storage, TAM basically consists of four distinct regions: code-block storage, frame storage, heap storage, and registers. The code-block storage representing the compiled program is accessible for all processors through fast local operations (in theory). The frame storage (including local data) is distributed over all processors, but each frame is local to one processor. In contrast to code-blocks, frames are allocated to the processor dynamically during the program execution. Thus, the corresponding frame for a certain code-block does not necessarily need to be on the same processor[1]. Since the frames are distributed over all processors, communication between frames often requires interprocessor communication.

Frames consist of local variables, entry counters for synchronizing threads, the ready frame link (the pointer to the next ready frame), and the Continuation Vector (CV). If a frame is not running, the CV is called the *Remote Continuation Vector* (RCV). The RCV is a stack containing pointers to all *enabled* threads. When a frame is scheduled, the RCV is copied onto the *Local Continuation Vector* (LCV). Basically, this means that the LCV is the CV of the running frame. The processor uses the LCV to schedule all enabled threads in a quantum. An example of a frame layout is illustrated in Appendix B.1.

Finally, the heap storage contains statically or dynamically allocated arrays. Large arrays are distributed over the processors whereas small arrays are local. By definition each heap location holds three presence bits (empty, full, deferred), which provide the possibility of element-by-element synchronization. A heap element is generally accessed through a split-phase operation (except for local accesses which are optimized). The response is handled by the corresponding inlet as explained below.

2.2.2 Threads and Inlets

Threads are sequences of instructions which follow the traditional concept of control-flow. By definition the code within a thread must never suspend. This means instructions in a thread can be

[1] This is not valid, if the entire program code fits on each processor-node's memory.

statically ordered and no operation of unbounded latency can occur in between instructions (e.g., a remote reference). Each enabled thread is described by an instruction pointer in the RCV (or LCV). The instruction pointer to the thread is an offset of the code-block's base-address which is stored in the frame. In addition to the computational instructions threads contain additional control operations such as FORK, SWITCH, and STOP. FORKs attempt to enable a thread in the *running* code-block only. If the thread is synchronizing (i.e., it requires more than one synchronization events before all data needed is available), FORK first decrements the synchronization counter. If the result turns out to be zero, the thread is enabled (i.e., pushed onto the LCV). Otherwise the decremented synchronization counter is stored back[1]. SWITCH conditionally forks one of two threads. STOP terminates a thread and starts the execution of the next enabled thread by popping its pointer from the LCV.

At the bottom of the LCV is always the leave-thread. Thus, if the LCV is empty, the leave-thread is executed. It contains the TL0-instruction SWAP which terminates the running frame, copies the RCV onto the LCV[2], and transfers control to the next ready frame (pointed to by the ready frame link). Appendix A.3 gives the exact mapping of SWAP to the SPARC. Appendix B.1 shows the physical organization of the frame in the memory. The following TL0-code illustrates a simple, common thread:

```
THREAD 7
        SUB ireg0.i = islot0.i 1.i      % subtract 1 from islot0 and write result
                                        % into ireg0
        FORK 9.t                        % fork thread 9
        STOP                            % pop next thread ptr. from LCV and start
                                        % a new thread
```

Threads can also be enabled by arriving messages (from the local or a remote processor). Messages are sent from other frames and contain arguments or returning results. Each message is received by a special, corresponding inlet which is a compiler-generated message-handler. An inlet usually contains three TL0-instructions: RECEIVE, POST, and STOP. RECEIVE extracts the data from the message and stores it into the destination frame (which is not always the current one). Inlets enable threads through POST, which is similar to, but distinct from FORK since POSTs enable threads from frames in either state. This implies that an idle frame has to be queued in the ready frame queue if an enabled thread is posted to the RCV of that frame[3]. The following sequence illustrates a common inlet code in TL0:

```
INLET 3
        RECEIVE islot7.i            % store the arrived value in islot7
        POST 11.t                   % post thread 11 to either RCV or LCV
        STOP                        % pop next thread ptr. from LCV and start thread[4]
```

---

[1] The SPARC mapping for FORK is given in Appendix A.1.

[2] This is valid for the frame representation we use. In other implementations the RCV and the LCV can be in the same, physical location.

[3] The SPARC mapping for POST is given in Appendix A.4.

[4] This is only valid in a uniprocessor system when arriving messages are detected by polling and thus are executed *between* two threads. If messages interrupt threads, STOP would simply mean a return from interrupt. If a separate processor is dedicated to handle messages, STOP would mean waiting for the next message. The SPARC mapping for STOP is given in Appendix A.2.

Observe that both threads (through FORK and SWAP) and inlets (through POST) cooperate in scheduling frames and threads. Thus, they both need to have access to common resources such as synchronization counters and the LCV. For this reason, atomicity between POST and FORK/SWAP must be guaranteed by any dual processor TAM implementation.

2.3 Mapping TAM on the CM-5

2.3.1 The CM-5 Multiprocessor

The CM-5 is a massively parallel architecture with SPARC processors [8]. The processors are interconnected in two disjoint, incomplete fat trees. Each processor node comprises a 33 MHz SPARC RISC-processor chip set, 8 Mbytes of local DRAM memory, and a Network Interface. The SPARC chip set includes FPU, MMU and a 64 Kbyte direct-mapped write-through cache. The Network Interface is attached to the processor node's MBus (a SPARC standard [10]) and consists of a pair of memory mapped FIFO queues. Figure 2.3 shows the basic processor node design [11].

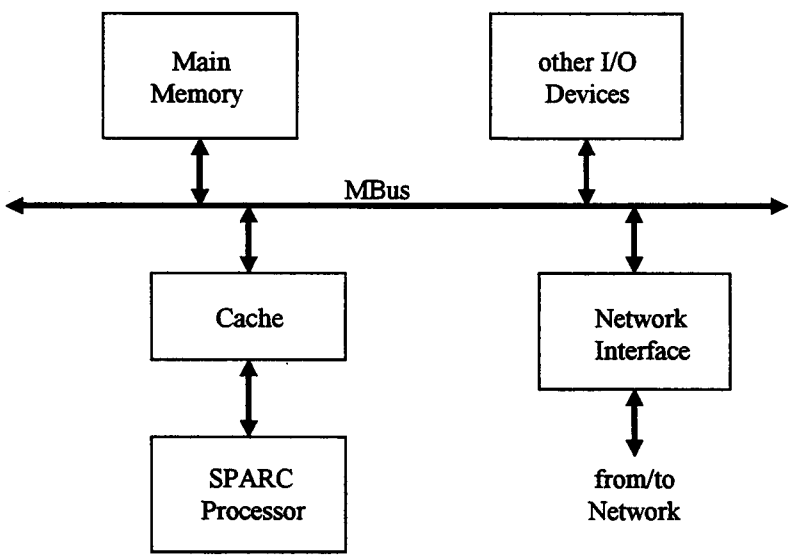

Figure 2.3 Conceptual processor node design of the CM-5

2.3.2 Storage Model

Program code is placed on every processor. Frames exist in the local memory and probably reside mostly in the cache. Small heap structures are local to a processor whereas large arrays are spread over the processors. TL0 registers are mapped on a single SPARC register window. The code generator has to spill excess TL0 registers to the activation frame. The register window is divided into special purpose registers, thread registers, and inlet registers. The special purpose registers (g0-g7) hold important variables and constants such as the pointer to the next ready frame (`queue`), the pointer to the top of the LCV (`lcv`), the processorID, the current frame pointer (`fp`), and the pointer to the base address of the current code-block (`cbbase`). Sixteen registers (i0-i7 and l0-l7) are used by threads only. Another eight registers (o0-o7) are used by inlets and may be temporarily used by threads. The TL0 instruction pointer (`ip`) and inlet instruction pointer (`iip`) are both mapped to the SPARC PC-register.

2.3.3 Message Handling

The only TL0 instructions that handle messages are `SEND` and `RECEIVE`. `SEND`s are limited on the CM-5 to three words of arguments plus two words for the continuation (`fp`, `ip`). Thus, longer messages are converted into multiple `SEND`s. Each `SEND` is paired with a corresponding `RECEIVE` in a specific inlet generated at compile-time. Table 2.1 depicts the cost for both instructions respectively.

`SEND` a message to	costs		Handling a message arrived from	costs	
	instr.	cycles		instr.	cycles
local frame			local frame[1]		
overhead	4	4	inlet overhead	6	7
push one word	1	1	`RECEIVE` one word	1.5	3
remote frame			remote frame		
overhead	10	25	inlet overhead	6	13
push one word	0.5	4	`RECEIVE` one word	1.5	6

Table 2.1 Cost of `SEND` and `RECEIVE` on the CM-5. `RECEIVE` only stands for transferring message data into the activation frame. The overhead of handling a message includes dispatching to the inlet and returning form the inlet. The overhead of `SEND` includes determining if the destination is local or remote. Push/`RECEIVE` one word assumes an average cost of 4 cycles for one word to be loaded from/stored to the Network Interface.

[1] Since we do not have the exact data the cycle cost for handling a local message is estimated. Instead of 4 cycles per word required to access the Network Interface, 1 cycle per word is now assumed since the data resides already in registers. Thus, the local overhead is 6 cycles faster than the remote one since it involves loading two words: `fp` and `ip`.

The reason for remote SENDs/RECEIVEs being so expensive is that they need to access the Network Interface. A load/store doubleword from/to the Network Interface incurs 8 cycles each. Currently, the arrival of a message on the CM-5 is detected by explicitly polling the Network Interface at the end of each thread (9 cycles). When a thread executes a SEND, polling is combined with the SEND (2 cycles + SEND). Although messages could be detected by interrupts, this method is inadequate in the CM-5 due to the prohibitive cost of interrupts.

Since heap access operations usually involve messages, they are also discussed in this section. Each structure element is represented by 64 data bits and 3 presence bits. The tags are stored separately from the data with a constant offset. An element can be empty, full or deferred indicated by the tag. Deferred fetches are queued as a linked list where each link indicates processor node, inlet and frame of the request. This information is used to satisfy the deferred read when the element is eventually written. The head of the list is stored in the element itself. TL0 instructions used for heap accesses are: IFETCH (I-structure reads), ISTORE (I-structure writes), IALLOC (I-structure allocation), and IFREE (I-structure deallocation).

All of these instructions are basically special forms of SEND with some extended functions. Their requests are received and serviced by generic inlets. For IFETCHes and IALLOCs, the service additionally includes replying (i.e., SENDing back the requested value). The reply in turn is received by an inlet on the processor that initiated the request. This final inlet is again a specialized handler corresponding to the initial request. Table 2.2 sums up the cost for IFETCHes and ISTOREs including all necessary operations.

I-structure fetch (IFETCH)	costs		I-structure store (ISTORE)	costs	
	instr.	cycles		instr.	cycles
local			local		
data present	8	11	no waiting fetches	9	15
data not-present	25	58	waiting fetches	18	30
remote			remote		
initiate request	18	38	initiate request	18	38
service - data present	29	91	service	13	44
- data not-present	39	115			

Table 2.2 Cost of TL0 heap-operations on the CM-5. The cost for remote requests include the request send and receive by the serving processor. Remote service comprises the cost of the reply and the dispatch to the inlet receiving the reply. The cost for a fetch of a non-present element represents both enqueuing the fetch in a linked list and fulfilling the request after the element has been written.

2.3.4 Scheduling of Frames and Threads

As presented in Section 2.2 FORK, SWITCH and STOP are the TL0 instructions which determine the scheduling of threads. Table 2.3 shows the cost for each of the three instructions. The cost for FORK varies. The last FORK in a thread is always specialized into a fall-through or a branch, thus, eliminating the need for an explicit STOP. An exception to this is when a synchronization fails. Since a thread must not suspend, FORKs within a thread can only push an enabled thread onto the LCV rather than branch to it. Unsynchronizing and successful synchronizing FORKs always enable threads whereas failed synchronizing FORKs only decrement the synchronization counter and store it back. Since SWITCH is identical to FORK except that it has an additional conditional code at the beginning, it is not specifically broken down into different categories. The detailed code for FORK is given in Appendix A.1.

Operation	costs	
	instructions	cycles
FORK		
fall through	0	0
branch to thread		
unsynchronizing	1	1
successful synchronizing	3	4
failed synchronizing (without required STOP)	4	8
push thread onto LCV		
unsynchronizing	3	5
successful synchronizing	6	10
failed synchronizing	4	7
SWITCH one of two threads	fork + 2	fork + 2
STOP - pop next thread from LCV	3	5

Table 2.3 Cost of FORK, SWITCH and STOP on the CM-5

A frame is scheduled by the TL0 SWAP-instruction that basically invalidates the old (current) frame pointer (fp) by replacing it with fp of the new frame to run (i.e., the next frame in the ready queue pointed to by the ready frame link). It also copies the content of the RCV of the new running frame into the LCV[1]. The *basic* SWAP copies the leave-thread plus the first three computational threads (4x16-Bit offsets copied at a time with 'load doubleword'). If the RCV holds more threads, subsequent copy operations incur additional costs (4 threads at a time). The costs for SWAP are shown in Table 2.4 and the detailed code is given in Appendix A.3.

[1] Other implementations are possible. See section 2.2.2.

Operation	costs	
	instructions	cycles
SWAP		
basic (first 3 threads plus leave-thread)	14	26
per extra 4 threads	6	12
POST		
without cost for synchronization		
to idle frame	12	18
to ready frame	9	14
to running frame	5	7
cost for synchronization		
successful	3	5
failed	4	7

Table 2.4 Cost of SWAP and POST on the CM-5

The cost of POST, which pushes enabled threads, varies depending on the state of the frame. If a frame was idle (i.e., its RCV was empty), it is made ready by queuing it in the ready frame queue. For running frames, the threads are pushed onto the LCV rather than the RCV. POST is more expensive than FORK for two reasons. First, it must determine, if the inlet's frame is running by comparing the current frame pointer (fp) with the inlet's frame pointer (ifp). Second, the pointer to the top of the RCV (rcv) is kept in the frame rather than in a register like the pointer to the top of the LCV (lcv). The cost for POST is depicted in Table 2.4 and the detailed code is given in Appendix A.4.

3. PREVIOUS WORK

This section describes the work of S. Kotikalapoodi [9]. His proposals comprise of specific architectural modifications to support fine-grain multithreading on stock processors. The proposals in this chapter provide a number of key design components for the following chapters.

3.1 A Design for Efficient Thread Scheduling

Experiments show that TL0-control instructions are one of the greatest contributors to the execution time of several benchmarks [6]. The time spent on control instructions accounts for as much as 27% of the total processor time. This cost is mainly due to three instructions: FORK, SWITCH, and STOP. The main areas for improvements are FORKs to synchronizing threads and STOPs since they account for about 30%-40% of all control instructions. Successful synchronizing FORKs at the end of a thread are combined with STOP and optimized to a branch (see Figure 3.1). But if the synchronization fails, a STOP has to be executed explicitly. In this case, the overall cost for FORK plus STOP is 13 cycles (see also Table 2.3), which is very expensive compared to the average cost for a FORK.

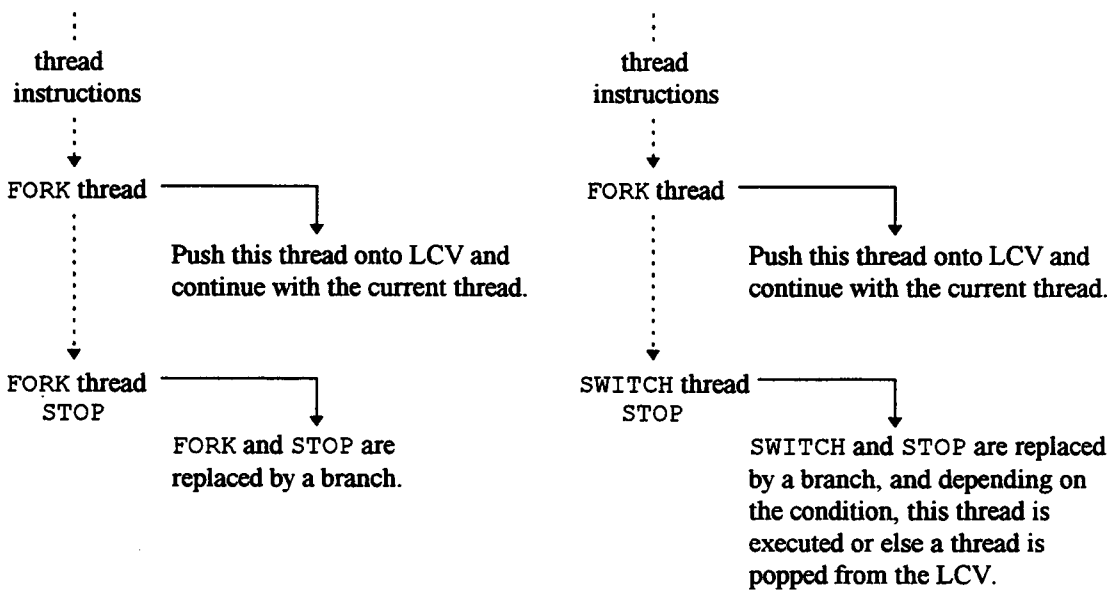

Figure 3.1 Control transfer using FORK/SWITCH

The original SPARC-code for failed synchronizing branches plus `STOP` is as follows (see Appendix A.1 for the complete code):

```
(FORK:)   ldb   sync[fp],tmp1      (2)   ; load synchr. counter into reg. tmp1
          subcc tmp1,1,tmp1        (1)   ; decrement synchronization counter
          be    thr_addr       (1 or 2)  ; branch to thr_addr, if synchr. count=0
          stb   tmp1,sync[fp]      (3)   ; synchr. counter not zero, thus store it back
(STOP:)   lduh  [2+lcv],tmp        (2)   ; load thread pointer (offset) from LCV
          add   lcv,2,lcv          (1)   ; increment pointer to top of LCV (lcv)
          jmp   [tmp+cbbase]       (2)   ; jump to new thread at cbbase+offset
```

In order to substantially reduce the cost for failed synchronizing branches a new SPARC-instruction was introduced: *conditional double branch and pop* - `cdbp` [9]. This instruction eliminates the need to explicitly access the LCV for the aforementioned case. It is assumed that a register, called `r_ntp`, holds the top of the LCV (i.e., the pointer to the thread that will be scheduled next). The register holding the decremented synchronization counter (`tmp1` in the above code) is labeled `r_cnt`. The `cdbp` instruction has the syntax '`cdbp thr_addr`' and functions as follows: '`thr_addr`' points to the thread whose synchronization counter resides in `r_cnt`. If `r_cnt` is zero, `cdbp` branches to the thread pointed to by `thr_addr` and nothing else needs to be done (2 cycles). If `r_cnt` is not zero, three actions have to be taken. First, `cdbp` branches to the thread pointed to by `r_ntp+cbbase`. Second, the next thread pointer is popped from the LCV into `r_ntp`. Finally, `r_cnt` is stored back (3 cycles altogether). The `cdbp` instruction is used for all synchronizing `FORK`s optimized to branches. The timing for the `cdbp` instruction is illustrated in Table 3.1.

cycle	operation	
i	*Fetch stage*: `cdbp` instruction is fetched	
i+1	*Decode stage*: `cdbp` is decoded; PC+`thr_addr` is computed; `r_ntp` and `cbbase` are read from the register file; instruction in the delay slot is fetched (i.e., `stb` instruction)	
i+2	if `r_cnt` = 0 *Execute stage*: `stb` in the delay slot is squashed; instruction at PC+`thr_addr` is fetched (Fetch stage of that instruction)	if `r_cnt` ≠ 0 *Execute stage 1*: PC←`r_ntp+cbbase`
i+3		*Execute stage 2*: `lcv`←`lcv`+2; instruction at PC is fetched
i+4		*Execute stage 3*: put `lcv`+2 on address bus and load data (new top of LCV) into buffer
i+5		*Write Back stage*: write buffer into `r_ntp`, and `stb` in the delay slot starts execute stage ('`stb`' could not execute earlier because `cdbp` was still on the bus)

Table 3.1 Timing of `cdbp` instruction

Using cdbp, the code for synchronizing FORKs optimized to a branch now looks like this:

```
ldb    sync[fp],tmp1          (2)     ; load synchr. counter into reg. tmp1
subcc  tmp1,1,tmp1            (1)     ; decrement synchronization counter
cdbp   thr_addr               (2 or 3) ; if r_cnt = 0, branch to thr_addr, else
                                         branch to (r_ntp)
stb    r_cnt,sync[fp]         (3)     ; delay slot: synchr. counter not zero, thus
                                         store it back
```

Another instruction proposed is a store with post-decrement capability - std. The syntax is:

```
std    rd,rs          ; store rd into [rs] and decrement rs
```

The std instruction accelerates by 1 cycle the pushing of enabled threads onto the LCV.

These two proposed instructions decrease the total processor time up to 3.4% (for Paraffins). This can be done with slight modifications to the SPARC that do not compromise the original functionality. This indicates that fine-grain parallelism on stock processors can be effectively implemented by relatively inexpensive architectural support. A detailed discussion of the above proposals including the hardware requirements can be found in the Master's thesis of S. Kotikalapoodi [9].

3.2 Message Handling

Another large percentage of the execution time comes from messages. Thus, the following suggestions attempt to decrease the cost for messages. As discussed before, messages are sent and received by the TL0 instructions, SEND and RECEIVE (including heap accesses). SENDs are always synchronous with the computation. In the CM-5 implementation, RECEIVEs are synchronous with computation as well since the Network Interface is polled rather than having an arriving message interrupt the computation. An alternative choice is to use a separate processor for receiving messages. The idea of having an Inlet-processor to execute inlet code was proposed in MIT's *T architecture [1]. However, the essential issue of atomicity between the instructions FORK, SWITCH, SWAP, STOP and POST were not adequately addressed. Thus, a complete node design with an Inlet-processor and basic solutions in order to guarantee atomicity between the two processors was proposed in [9].

Figure 3.2 shows the suggested processor node design. Coherency between both processors is supposed to be guaranteed by having all data in a common cache. The Network Interface has been integrated into the Inlet-processor to decrease the network access cost which is expensive on the CM-5 (see 2.3.3).

The Inlet-processor executes all inlets. As discussed in Section 2, a typical inlet has three instructions: RECEIVE, POST, and STOP. With an Inlet-processor, STOP is replaced by the new TL0-instruction NEXT. The NEXT instruction waits for the next message to arrive and then dispatches it to the appropriate inlet code.

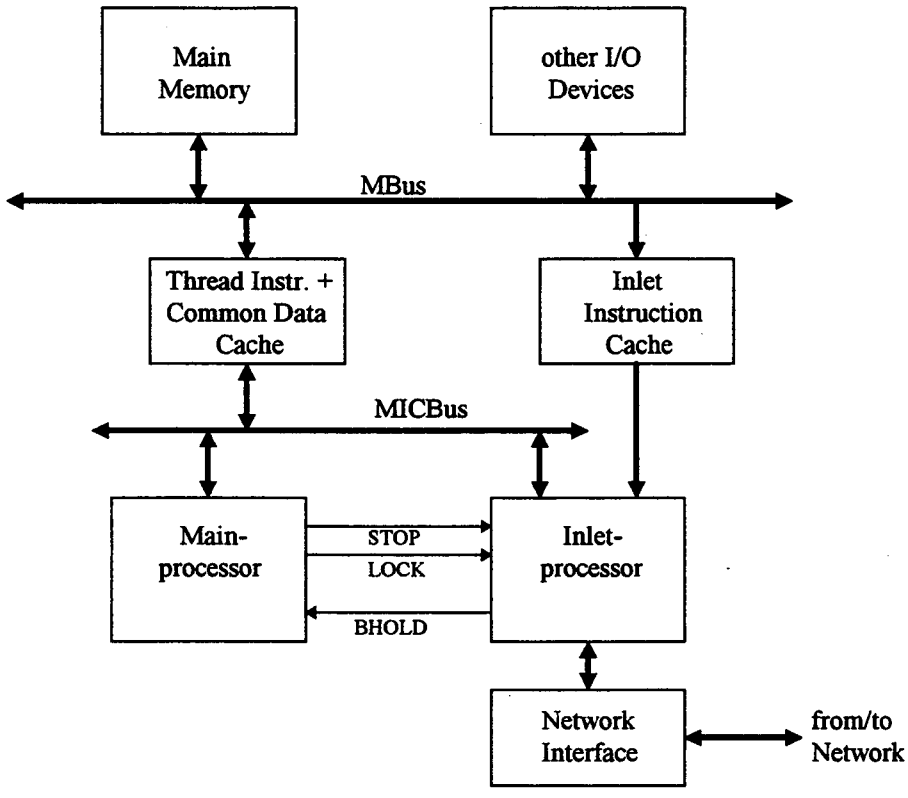

Figure 3.2 Inlet-processor interface with system

The problem of bus contention is resolved by two control signal between the processors: BHOLD and LOCK. Usually the Main-processor is on the Mainprocessor-Inletprocessor-Cache Bus (MICBus). Whenever the Inlet-processor needs the MICBus it sets BHOLD which stalls the Main-processor. On the other hand, the Main-processor can atomically access the Common Cache for synchronization counters by setting LOCK which stalls the Inlet-processor. LOCK is set when '`ldb`' and '`subcc`' occur right after each other, which happens whenever a synchronization counter is accessed (see Appendix A.1 and A.4). It is reset by either the zero-bit (i.e., `r_cnt` = 0 after successful synchronization) or the '`stb`' instruction (i.e., after failed synchronization). If LOCK was applied right after the Inlet-processor loaded a synchronization counter, then the Inlet-processor needs to load this synchronization counter again after LOCK is reset, because the Main-processor may have potentially accessed and updated the same synchronization counter.

Finally, another atomicity problem occurs whenever both processors access the LCV simultaneously (i.e., `FORK-POST` interference) since the pointer to the top of the LCV (`lcv`) resides in a Main-processor register. Consider the case when both processors want to push different threads. The

Inlet-processor reads `lcv` directly after the Main-processor reads it. Then they will push their thread pointers onto the same stack slot. The result is a loss of one thread pointer.

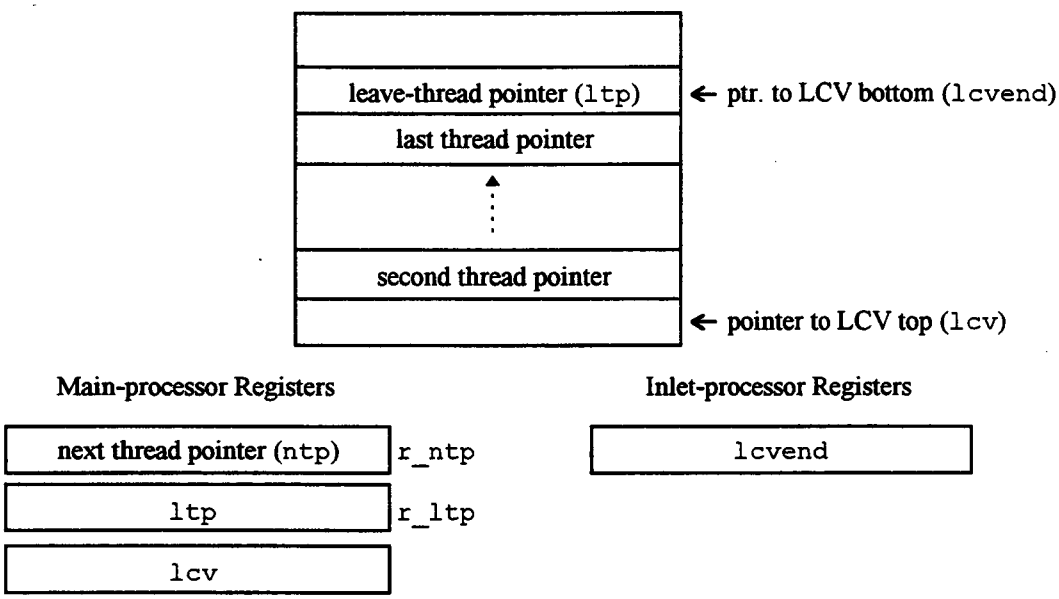

Figure 3.3 Double-ended representation of the LCV

To alleviate this problem, the LCV is proposed to be implemented as a double-ended stack (see Figure 3.3), where the Inlet-processor appends thread pointers to the bottom of the LCV instead of pushing them onto the top of the LCV. This is done by inserting the thread to be pushed between the last computational-thread pointer and the leave-thread pointer, which is implemented as follows:

```
swap   [lcvend],reg        (4)    ; swaps thread pointer to be posted (in reg) and
                                  ; leave-thread pointer (ltp) which is at LCV bottom
addcc  lcvend,2,lcvend     (1)    ; pull stack down by one slot
sth    reg,[lcvend]        (3)    ; store ltp at LCV bottom
```

However, this operation requires an additional 4 cycles for *every* inlet posting a thread.

Although the double-ended implementation of the LCV eliminates the FORK-POST interference, a problem can occur when the LCV becomes empty and the leave-thread pointer is popped into `r_ntp`. The first instruction in the leave-thread has to check if the LCV is still empty, because a new inlet could have posted another thread after the leave-thread pointer was popped (SWAP-POST interference). If the LCV is still empty, SWAP in the leave-thread stalls the Inlet-processor by means of the STOP control-line until the SWAP instruction has terminated.

3.3 Discussion

The introduction of `cdbp` and `std` decreases the total processor time on the CM-5 without Inlet-processor up to 3.4% (for Paraffins). Thus, the proposed instructions support fine-grain programs efficiently without compromising the functionality of the conventional hardware.

The introduction of an Inlet-processor frees the Main-processor from the task of receiving messages by either polling or interrupts and thus increases the overall performance.

Atomic memory access for the Main-processor is guaranteed by a simple LOCK signal. The STOP control-line is supposed to guarantee atomicity between `SWAP` and `POST`.

However, some important questions remain. There are three fundamental disadvantages to the simple LOCK approach: First, the Inlet-processor is always stalled for the *complete* synchronization operation (i.e., read/modify/write) even though the probability is very high that both processors do not access the *same* synchronization counter *simultaneously*. Second, whenever the synchronization operation by the Inlet-processor has been preempted by the Main-processor (using LOCK), the Inlet-processor needs to reload the synchronization counter and repeat the complete synchronization operation. Third, it is not explained how 'reloading the synchronization counter' can actually be implemented. For example, consider the following:

- The Inlet-processor needs to keep track of how many instructions it has already advanced after loading the synchronization counter since LOCK can occur at any time after the load. Afterward, the Inlet-processor needs to know to which address PC must be reset to in order to re-execute the synchronization code (starts with '`ldb`').
- This requires a substantial amount of hardware.
- Somehow LOCK must be disabled during the execution of code other than the one for synchronization.

In terms of the `SWAP-POST` interference another issue needs to be addressed in more detail. It is assumed that the processors can read each other's registers. However; this is not a trivial assumption since major architectural modifications may be required to implement this capability. Moreover, under certain circumstances, the proposed method will fail. For example, suppose a `POST` has just determined that an inlet is destined for the running frame[1]. At this moment, the TL0-instruction `SWAP` (in a leave-thread on the Main-processor) applies the STOP control-line thereby stalling the Inlet-processor. Then `SWAP` switches to a new frame and updates LCV by copying the RCV of the newly running frame into it. As soon as the STOP signal is reset the interrupted `POST` continues. Since `POST` had already determined before that it is for the running frame (which, in reality, is not running anymore) it posts the thread onto the updated LCV of a completely different frame (just switched to by `SWAP`).

[1] The detailed code for `POST` is given in Appendix A.4.

A problem also occurs when POST had determined before that it is not for the running frame and had already loaded the pointer to the top of the RCV (rcv) to push a thread on the RCV of the inlet's frame. Now SWAP sets the STOP control-line right after rcv has been loaded. Then SWAP empties the RCV by copying it onto the LCV. If this SWAP switches to the frame to which the interrupted POST is for, then rcv read by POST before the interrupt has become invalid. If POST simply continues executing, first, it will push the enabled thread into an RCV slot pointed to by the *invalidated* rcv and, second, it will *store back* the invalidated rcv. Yet the old, invalidated rcv points to a wrong slot in the RCV and implies there are more threads in the RCV than there actually are.

Finally, it must be considered that all *local* SENDs (from messages and heap accesses) are optimized. This implies that these SENDs execute on the Main-processor. Two questions arise. First, how can the Main-processor actually send a message, i.e., how can it access the Network Interface that is now integrated into the Inlet-processor? Second, locally optimized SENDs imply that the corresponding inlets are still run on the Main-processor instead of on the Inlet-processor. This would significantly increase the cost of POSTs since they would have to distinguish additionally, if threads must be pushed onto the *top* of the LCV (from inlet on Main-processor) or if they must be appended to the LCV *bottom* (from inlet on Inlet-processor).

The following chapter addresses the issues described above by proposing specific, detailed architectural support to guarantee atomicity between the two processors and to resolve the POST-FORK/SWAP interference. Moreover, the issue of interprocessor communication (e.g., mutual reading of registers) is addressed briefly in order to lay the foundation for an overall node design.

4. EFFICIENT INTERFACE DESIGN OF INLET-PROCESSOR

In this chapter specific hardware proposals are presented which solve the problem of atomicity between the Main-processor and the Inlet-processor in the context of TAM. At the same time, the proposed design also minimizes the utilization of the bus, which connects the Main-processor, the Inlet-processor, and the Common Data Cache (MICBus). This bus is the bottleneck of the current processor node design (see Figure 3.2).

The following assumptions are essential for the proposed solutions: First, we assume the MICBus to be a new, separate bus. Second, it is reasonable to assume that a single, common clock is used for both the Main-processor and the Inlet-processor because an asynchronous clock between the two would increase the time penalty for bus accesses of either processor.

Finally, since the Inlet-processor operates completely independent of the Main-processor, a Bus Arbiter is introduced that gives both processors the same priority. This is necessitated by the fact that the MICBus will be fully utilized by the Main-processor alone, but the Inlet-processor will also carry a substantial part of the workload. From here on, the following bus arbitration timing is assumed. The bus is requested and granted in the Execute cycle of a memory-accessing instruction. It is released at the very beginning of the Write-Back cycle. Figure 4.1 illustrates a timing example showing the pipeline stages of the load-word instruction. The timing of the load-word instruction is consistent with the SPARC [11].

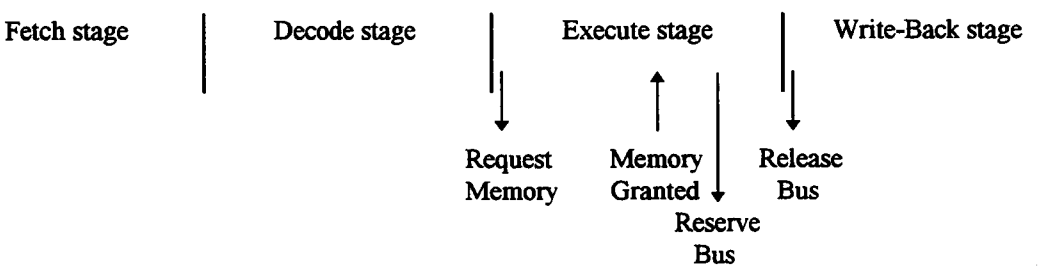

Figure 4.1 Timing example of a bus arbitration for the load-word instruction

Based on the aforementioned assumptions, the following operations must be atomic: Access to synchronization counters and access to the LCV. The SWAP operation and enabling idle frames are other sources of potential problems in the area of interprocessor communication. The following subsections will discuss in detail each of these operations.

4.1 Access to Synchronization Counters

The TL0 instructions that access synchronization counters are synchronizing FORKs and POSTs. Since SWITCH is a conditional FORK, we do not mention it further. The parts of the SPARC assembly code for FORK and POST that access synchronization counters are shown below (for the complete codes see Appendices A.1 and A.4).

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

POST		(cycles)
ldb	sync[ifp],tmp1	(2)
subcc	tmp1,1,tmp1	(1)
bnz,a	continue	(1 or 2)
stb	tmp1,sync[ifp]	(3)

Both sequences start by loading a synchronization counter from the frame and decrementing it. If the decremented synchronization counter is *not* zero, it is stored back using the delay slot and the next TL0 instruction at continue executes. If the synchronization counter is zero, POST will push the thread onto the LCV (if fp=ifp) or onto the RCV (if fp≠ifp), while FORK will either branch to the thread at thr_addr or push the thread onto the LCV[1].

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. This means 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.

One way to guarantee atomicity is to do it in software, e.g., using 'Test&Set'. Most modern processors provide atomic load/store operations required in a multiprocessor environment where variables are shared among several processors [10]. Although these operations can be used to implement atomic access to synchronization counters, their generality and (cycle) cost is unnecessary in a processor node design using an Inlet-processor. First, synchronization counters in the context of TAM are stored in the frame which resides in the local memory. Since remote processors can access the local memory only through the Inlet-processor in the suggested processor node design, access to synchronization counters has to be atomic only between the Main-processor and the Inlet-processor. Second, considering the special circumstance in which the processors interact (e.g., the TAM code for synchronization), a specific and inexpensive hardware can be designed to minimize the overhead for accessing synchronization counters, e.g., busy-waiting. The following example proves the benefits of a hardware solution compared to a software solution. Using the means provided by the SPARC, one of the cheapest ways for a software solution would be the following code (e.g., for POST):

[1] For details on when FORK branches to a thread or when it pushes a thread, see 2.3.4. For an explanation of cdbp, see 3.2.

```
start: ldstub  sync[fp],tmp1      (4)   ; atomically: tmp1←sync[fp], then
                                        sync[fp]←FF_hex (lock)
       cmp     tmp1,FF_hex        (1)   ; was sync[fp] locked?
       be      start              (1 or 2) ; if sync[fp] locked, branch to start, else
                                        continue synchronization
       subcc   tmp1,1,tmp1        (1)   ; decrement synchronization counter
       bnz,a   continue           (1 or 2) ; if not zero, execute delay slot and branch
       stb     tmp1,sync[fp]      (3)   ; store back decremented synchronization counter
```

Compared to the original code for either `POST` or `FORK`, this atomic synchronization will always incur an additional 5 cycles for every `FORK`, `SWITCH`, and `POST` to a synchronizing thread. For example, for the benchmark Gamteb, these instructions account for 28% of all TL0 instructions. With a total average cycle cost per TL0 instruction of 13.6 [6], the above solution would increase the processor time by more than 10% ($5 \times 0.28 / 13.6$). This percentage will be even higher after the introduction of an Inlet-processor. This is because most of the synchronization operations occur in `FORK`s and `SWITCH`es which execute only on the Main-processor.

Based on the above discussion, a hardware solution is proposed to efficiently handle atomic access to synchronization counters. Our design (Figure 4.2) allows one processor not accessing a synchronization counter to continue to access the MICBus while the other processor accomplishes its synchronization operation. The processor can even access *another* synchronization counter simultaneously with only one exception, it must not access the same synchronization counter. Only in this case, the processor has to busy-wait.

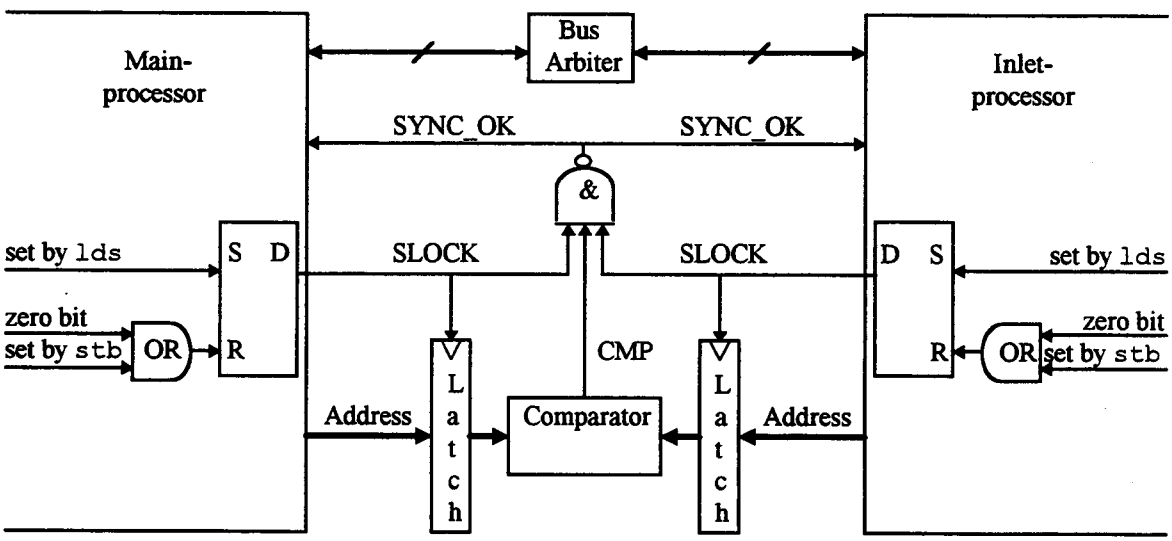

Figure 4.2 A design for atomic access to synchronization counters

The Bus Arbiter allows only one of the processors to access the MICBus. Whenever 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. To ensure atomicity, the SLOCK signal remains 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, called `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. The zero bit must also be reset because it might have been set by previous instructions. SLOCK is then reset whenever the zero bit is set (i.e., successful synchronization) or the `stb` instruction completes (i.e., failed synchronization). The comparator basically detects if the same synchronization counter is being accessed by both processors. Since both addresses are latched as long as SLOCK remains set, the comparator detects conflicts during the entire synchronization operation. On the other hand, if the addresses are not equal, the CMP signal remains zero and thus SYNC_OK remains one.

If SYNC_OK is zero, the processor that is just beginning its bus access must stall until the other processor has reset SLOCK to zero which in turn causes SYNC_OK to be set to one. SYNC_OK becomes *only* zero, if CMP is one and *both* SLOCK are one. Including SLOCK in the NAND condition is important, since CMP might unintentionally stay one. This is because the latch is edge-triggered, i.e., the latch holds the last address latched (even after SLOCK is reset to zero) until a new address is latched.

Except for the rare case when both processors access the same synchronization counter simultaneously (the average entry count for Gamteb is only 2.5 [6]), 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. Finally, the overhead is also reduced by simplifying it to a bus contention problem.

4.2 Access and Representation of LCV

Conceptually, having both processors access the LCV simultaneously creates two atomicity problems. First, each push/pop on/from the LCV by the Main-processor potentially interferes with a push onto the LCV by the Inlet-processor (i.e., `POST-FORK` interference). Second, a `SWAP` on the Main-processor can execute during a `POST` on the Inlet-processor (i.e., `POST-SWAP` interference). If the `POST` is to a terminating, yet still running frame, it might push a thread onto the just emptied LCV. Or, if the `POST` is to the next ready frame, this could cause a problem that a thread is posted in the just emptied RCV before `SWAP` could update `rcv`. In both cases the posted thread will be lost.

4.2.1 POST-FORK 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 POST-FORK interference entirely and simplify the design by having FORKs push threads only on the LCV and POSTs push threads on the RCV. There are two possible implementations when POSTs push threads onto the frame's RCV (when `fp=ifp`) instead of onto the LCV. First is to check at the end of the current quantum, if any new threads have been posted in the RCV. In this case, the threads in the RCV are copied onto the LCV and the current frame continues. The second possibility is to schedule these threads during the next quantum. However, both methods will degrade performance because more than 50% of all threads posted (Gamteb) are for the currently running activation [8] and 14%-32% of all threads are enabled by POSTs *during* the execution of a quantum [6].

The first method would roughly double the amount of leave-threads and thus SWAPs executed. This is because an additional SWAP would have to be executed as soon as one thread is *posted during* the execution of a quantum. On the average, more than one thread is posted during a quantum [6]. Thus, at least one additional SWAP will be executed in each quantum. For example, consider the benchmark QS where SWAPs account for 0.53% of all TL0 instructions at a cost of 26 cycles per SWAP and an overall average of 15.1 cycles per TL0 instruction (CPT)[1]. This would increase the overall CPT by about 1% (an additional 0.53% SWAPs × 26 cycles per SWAP). In a system with an Inlet-processor and a workload distribution of ideally 50/50, the relative increase in Main-processor time would be almost 2% (since all SWAPs execute on the Main-processor). However, the 2% increase represents the most optimistic case. The actual penalty would be higher. This is because many threads that are no longer posted during the original quantum would have forked other threads. Consequently, these forked threads would have to be scheduled later and thereby the average quantum size would decrease. Thus, the actual cost increase on the Main-processor will be more than 2%.

The second method would also shorten the quanta, this time about 14%-32%, and thereby increasing the number of SWAPs by 16%-47%. A more serious effect on the performance is the decrease of locality since the current frame might not be directly re-scheduled next.

To eliminate the aforementioned overhead, the LCV is implemented as a double-ended stack (see Section 3.2), where the Inlet-processor always appends thread pointers to the LCV bottom (if `fp=ifp`), while the Main-processor pushes/pops them on/from the LCV top. The advantage of the double-ended stack is the two processors do not have to share the pointer to the LCV (`lcv`).

As mentioned in Section 3.2, one problem with the double-ended stack is whenever a thread pointer is pushed onto the bottom of the stack, it must be inserted between the leave-thread pointer (`ltp`)

[1] These numbers are based on a system without an Inlet-processor. However, this does not invalidate the following argumentation.

and the last thread pointer. This operation requires four additional cycles for each POST compared to the original single-ended stack used by TAM. To eliminate this additional cost, the proposed idea is to keep the leave-thread pointer (ltp) in a Main-processor register, called r_ltp, rather than at the bottom of the stack. 'ltp' is moved into the r_ntp rather than popping another (non-existent) thread from the LCV as soon as lcv (in the Main-processor) and lcvend (in the Inlet-processor) are equal. Then, the cdbp instruction schedules r_ntp. Figure 4.3 illustrates the modified, double-ended stack for the LCV.

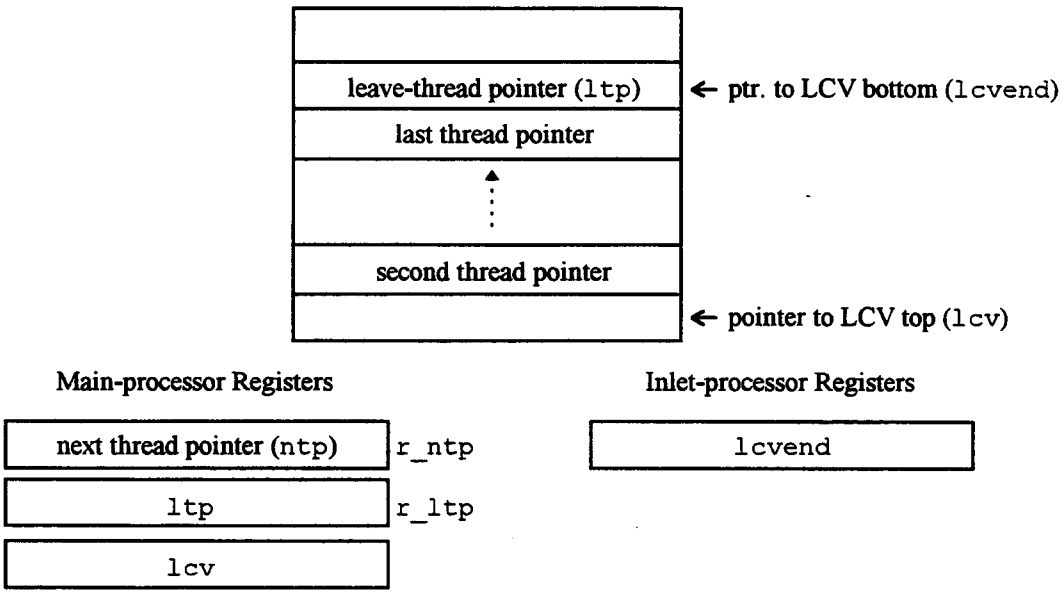

Figure 4.3 Modified representation of the LCV as a double-ended stack

The representation of the LCV as a double-ended stack eliminates the atomicity problem between FORKs and POSTs; therefore, the only overhead resulting from this implementation is the bus contention (which exists anyway because a bus is shared among processors). For the Main-processor, this cost depends on whether POST on the Inlet-processor is to an idle (7 cycles), a ready (5 cycles), or a running (2 cycles) frame, since the number of frame accesses required differs depending on the frame's state. These costs represent only the time actually spent on the MICBus to load and store data since the Inlet-processor fetches its instructions over a separate bus from the Inlet Instruction Cache. The cycle costs can be confirmed by referring to the detailed SPARC code for POST in Appendix A.4.

4.2.2 POST-SWAP interference

SWAP occurs only in the leave-thread. Thus, it is initiated only when the LCV becomes empty, which is indicated by the fact that `lcv` equals `lcvend` as mentioned in Section 4.2.1. The two basic possibilities to determine if `lcv` is equal to `lcvend` is either through software or through hardware. A software solution has the following disadvantages: First, a software compare would significantly increase the control cost because one additional cycle must be executed for each FORK or SWITCH combined with a STOP at the end of a thread (except for failed synchronization in FORKs and SWITCHes where `lcv` is not adjusted). Second, the Main-processor would have to load `lcvend` from the Inlet-processor which would cost both processors at least another additional cycle. Third, an additional atomicity problem would be created - not in the cache, but between the processor registers.

Therefore, a hardwired compare is chosen rather than a software compare (Figure 4.4). The comparison sets a control bit as soon as `lcv` and `lcvend` are equal. Since this control bit indicates that the LCV is empty, it is called the stack-empty bit - STEM.

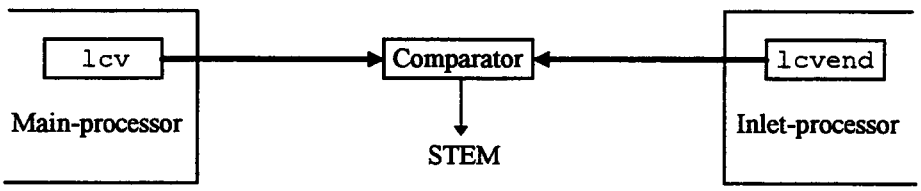

Figure 4.4 Simple, hardwired comparison of `lcv` and `lcvend`

There is a disadvantage of having a directly connected, hardwired compare. It requires a 16-bit bus between the processors. This is unacceptable for two reasons. First, 16 additional ports would be needed for both processors which contradicts our objective to keep the changes to the original processor to a minimum. On top of this, the bus' sole purpose would be the comparison of only two registers. The usefulness of this bus simply does not justify the amount of hardware and adjustments needed to implement it.

To circumvent this problem a scheme is proposed that minimizes the change needed to the original SPARC (see Figure 4.5). It is assumed the Inlet-processor loads both `lcv` and `lcvend` into registers during the SWAP operation. In order to ensure coherence between the two `lcvs`, the Inlet-processor is not allowed to access `lcv` at all. Instead, both `lcvs` 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 or decremented, lcv in the Inlet-processor is also incremented or decremented during the same cycle. This allows both lcvs to appear entirely identical to both software as well as hardware. INCLCV is set, if the decoded instruction is cdbp (used to pop threads) and the decremented synchronization counter is not zero. DECLCV is set, if std (used to push threads) is decoded and it addresses lcv. Figure 4.6 illustrates the logic needed for both control lines.

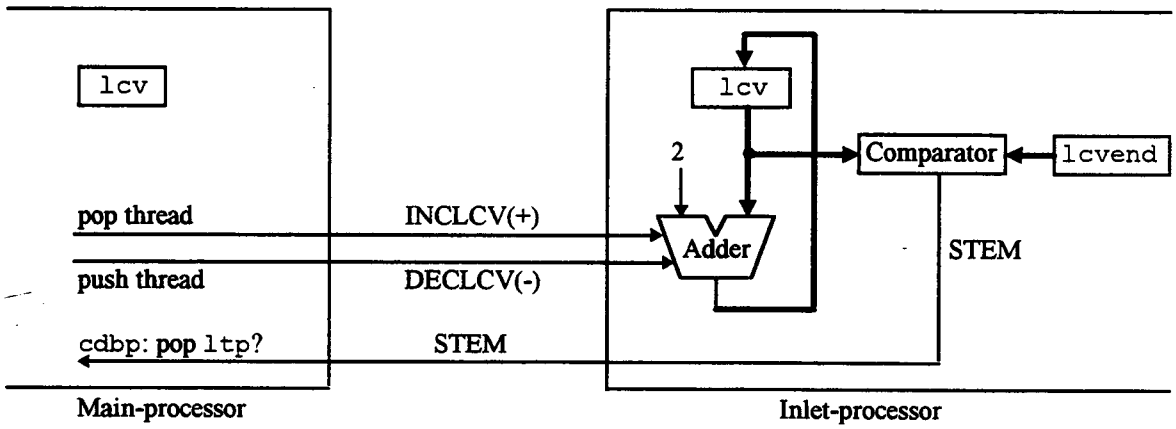

Figure 4.5 Optimized scheme for hardware comparison of lcv and lcvend

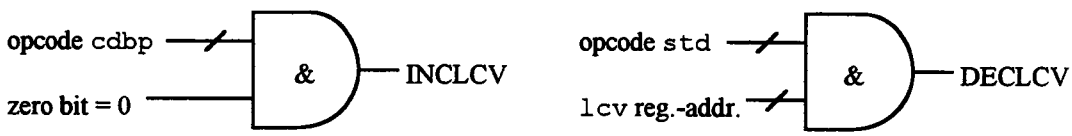

Figure 4.6 Decoding of condition for INCLCV and DECLCV

Both control lines are asserted during the execute cycle because the STEM signal must be available for the Main-processor at the end of the execute cycle of a lcv update. For successful synchronization, this cycle is execute stage 2 of the cdbp instruction (see Table 3.1). This is because execute stage 3 of cdbp must know if the LCV was emptied in the previous cycle. If the LCV is empty, the leave-thread pointer (ltp) is moved from r_ltp into r_ntp during execute stage 3. This allows the leave-thread to be scheduled next whenever lcv equals lcvend. If STEM is zero, cdbp pops a thread pointer from the LCV as discussed in Section 3.1.

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. `SWAP` might switch to the next frame although `POST` has just pushed a new thread pointer onto the LCV. This thread pointer will be lost. 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, if `SWAP` starts after `POST` pushed the tread pointer onto the LCV, but before `POST` could store back the adjusted pointer to the RCV top (`rcv`), then this thread pointer will also be lost.

The following four paragraphs will discuss in detail this interference. They are not essential for the understanding of the later proposal and thus the reader might skip them for a first reading. Two assumptions are made as a basis for the following explanations. First, the Main-processor can write to Inlet-processor registers within one cycle by means of the `movi` instruction (move to Inlet-processor). The Inlet-processor is stalled for one cycle during this operation. The `movi` instruction is proposed and explained in Section 4.3. Second, the first instruction of the `SWAP` code (`mov queue,fp`) in the decode stage causes the Inlet-processor to initiate the execution of the same instruction simultaneously (refer to Appendix A.3 for the code of `SWAP`). This assumption is important, because this instruction changes `fp` in both processors. Finally, note that both processors share the MICBus for data, but the Main-processor also fetches its instructions over the MICBus. Also note that the Bus Arbiter grants the bus to the processor that did not have it for the longest time.

As already mentioned before, the Main-processor moves `ltp` from `r_ltp` into `r_ntp` when the stack is empty (STEM=1), so that the leave-thread will be executed next. The essential instruction in every leave-thread is `SWAP`. `SWAP` accomplishes the entire process of switching to the next frame. Basically, it invalidates the current frame pointer (`fp`), sets up the new `fp`, loads some important pointers related to the new frame, and finally copies the new frame's RCV onto the LCV.

For the first interference, consider the concurrent execution of `SWAP` and `POST` with the following timing. Only the crucial parts of code are listed below:

cycle	SWAP (on Main-processor)		POST (on Inlet-processor)	
i			cmp	fp,ifp
i+1	mov	queue,fp	set	Lthr-Lcbbase,tmp2
i+2	ld	Fqueue[fp],queue	* Stall due to 'mov queue,fp' *	
i+3	movi	$queue_M$, $queue_I$[1]	be	isrunning
i+5	ld	Fcbbase[fp],cbbase	sth	tmp2,[lcvend]
		⋮	add	lcvend,2,lcvend
		copy RCV onto LCV		(done)

`POST` first determines the thread is for the running frame (`ifp=fp`). Then, it calculates the relative thread pointer (`tmp2`) and branches to the code for the case '`isrunning`'. Finally, the thread pointer is pushed onto the LCV bottom and `lcvend` is incremented. `SWAP` first sets up the new frame pointer by moving it from the `queue` register (ready-frame link) to the `fp` register which automatically is initiated

[1] moves the contents of the `queue` register on the Main-processor into the `queue` register on the Inlet-processor.

in the Inlet-processor as well. During the following two instructions, the pointer to the next ready frame is loaded from the ready-frame link of the new frame into `queue` and then moved into the corresponding Inlet-processor register (called `queue` as well). Further on, `SWAP` will copy the new frame's RCV onto the LCV which `SWAP` assumes to be empty. Thus, `SWAP` deletes the just newly posted thread of the *previous* frame by overwriting it.

For the second type of interference, the timing of `POST` and `SWAP` starts off the same way. This example illustrates when the inlet's frame is the same one to which the Main-processor wants to swap to (new `fp=ifp`).

cycle	SWAP (on Main-processor)	POST (on Inlet-processor)
i		`cmp fp,ifp`
i+1	`mov queue,fp`	`set Lthr-Lcbbase,tmp2`
i+2	`ld Fqueue[fp],queue`	*stall due to 'mov queue,fp'*
i+3	`movi` $queue_M, queue_I$[1]	`be isrunning`
i+5	`ld Fcbbase[fp],cbbase`	*delay slot flushed*
i+6	*`ld` data `Fqueue` on bus*	`ld Frcv[ifp],tmp3`
i+7	*`movi` data `queue` on bus* ⋯⋯▸	*receive new `queue`*
i+8	*`ld` data `Fcbbase` on bus*	`cmp tmp3,ifp`
i+9	*stall because Inlet-pr. has bus*	*`ld` data `rcv` on bus*
i+10	`ld Frcv[fp],tmp1`	`std tmp2,[tmp3]`
i+11	`ldfd -6[fp],ftmp`	`st tmp3,Frcv[ifp]`
i+12	`sub lcvbase,2,lcv`	`bnz continue`
i+13	*stall because Inlet-pr. has bus*	*`std` data `tmp2` on bus (thread ptr.)*
i+14	*Inlet-pr. still on bus*	*still storing*
i+15	*`ld` data `tmp1` on bus (`rcv`)*	*want to store `tmp3`, but Main-pr. has bus*
i+16	*stall because Inlet-pr. has bus*	*`st` data `tmp3` on bus (`rcv`)*
i+17	*Inlet-pr. still on bus*	*still storing*
	⋮ (continues)	⋮ (continues)

This case is very similar to the first one in that `SWAP` updates `fp` right after `POST` compared it with `ifp` (i.e., again `POST` compares the wrong `fp`). Yet this time, `POST` determines initially, that it is not for the running frame (i.e., `fp`≠`ifp`). Thus, it does not branch to '`isrunning`', although, in fact, the frame is running, because `SWAP` is in the process of switching to it (i.e., new `fp=ifp`). `POST` then pushes the thread pointer onto the running frame's RCV (cycles i+13/14) and finally wants to store back the adjusted `rcv` (cycles i+16/17). But here is the problem: `SWAP` has already read `rcv` before (cycle i+15). Thus, the newly pushed thread pointer is lost.

By comparison, the implication of both these cases are the same. Clearly, no thread pointer must be lost. Therefore, `SWAP` must stall the Inlet-processor in order to prevent it from executing a new `POST` (e.g. due to the subsequent arrival of a message) until `SWAP` itself has terminated. Additionally, `SWAP` needs to wait until any `POST` has terminated. In other words, whenever `SWAP` starts executing while a `POST` is in process on the Inlet-processor, the Main-processor must be put in a wait state. Therefore, a

[1] moves the contents of the `queue` register on the Main-processor into the `queue` register on the Inlet-processor.

control signal WAIT is used to inform the Main-processor whether or not the Inlet-processor is executing a POST.

On the other hand, when no POST is executing or as soon as POST has terminated (WAIT=0), SWAP asserts the HOLD signal which irreversibly terminates the current activation and stalls the Inlet-processor thus prohibiting it from executing any inlets. Figure 4.7 shows the final design for the POST-SWAP interference.

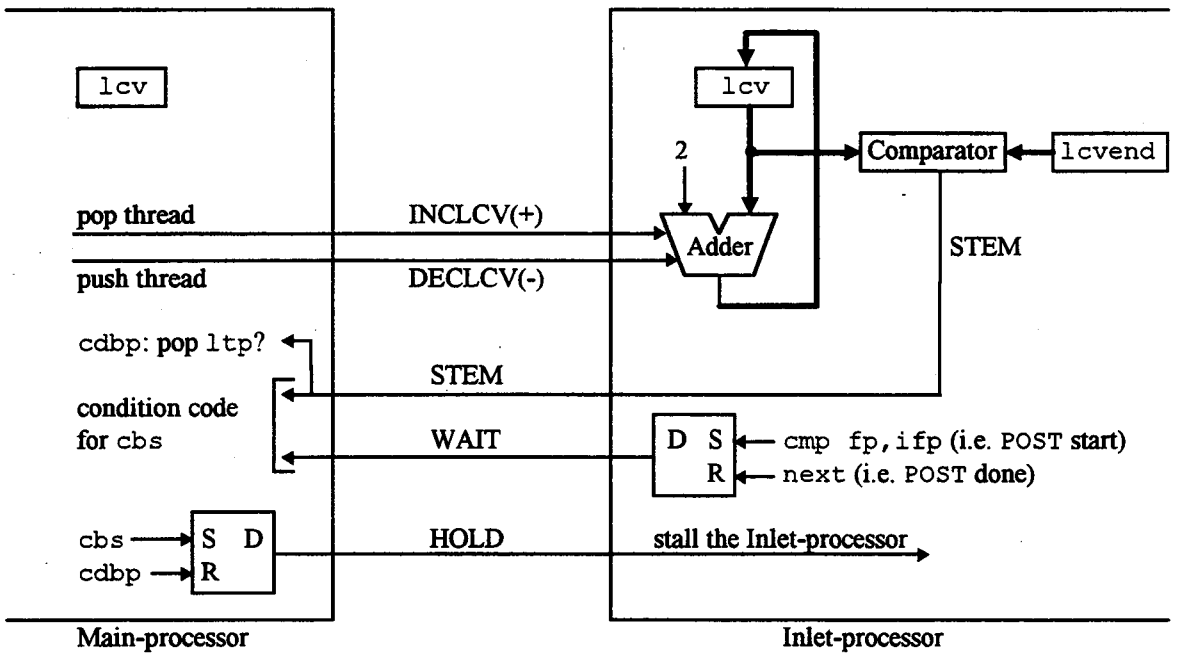

Figure 4.7 Final design for POST-SWAP interference including all control lines

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 pointer has been posted in the LCV or the RCV or a new frame has been queued as a result of posting to the RCV. WAIT is reset by the next instruction, which only exists in the inlet code. As illustrated in Section 2.2.2, inlets always end with a STOP. But in contrast to the normal STOP needed in a system with uniprocessor nodes (see Appendix A.2), the STOP on the Inlet-processor does not return to thread execution after an inlet. Instead, it dispatches to the next inlet or waits until a new message has arrived. Thus, a TL0-STOP on the Inlet-processor is simply translated to next.

As illustrated by the first atomicity problem described above, it is possible that POST pushes a new thread onto the LCV while SWAP wants to switch to the next frame in the ready queue. Atomicity between the two instructions is now guaranteed by the fact that SWAP is stalled by WAIT while a POST is executing. But clearly, the newly posted thread must be allowed to execute since it may fork other threads. Therefore, SWAP must check if the LCV is still empty (i.e., STEM=1), before it changes fp and continues to switch to the next activation. If STEM has become zero, SWAP must terminate and load the newly posted thread pointer into r_ntp. Then the execution continues from the new thread which simply means the former fp stays valid and thread execution continues as usual. An example for the aforementioned case is illustrated in Figure 4.8.

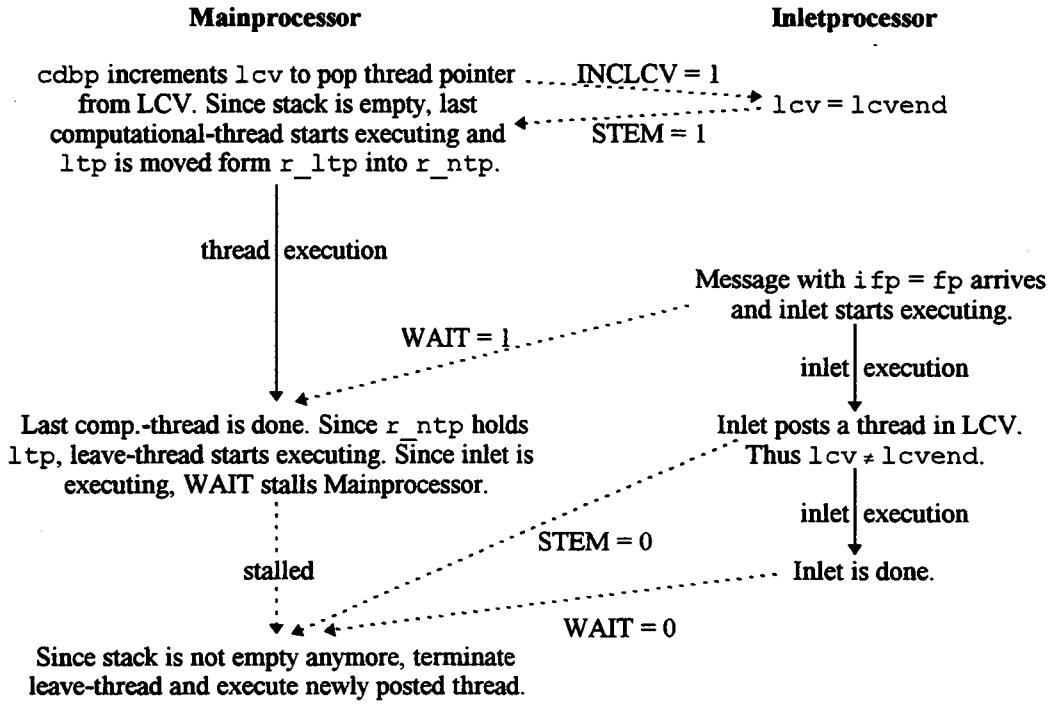

Figure 4.8 Timing example for arrival of new inlet for terminating frame

Unfortunately, SWAP is not always the only TL0 instruction in the leave-thread. Sometimes, registers might have to be saved before. This is why it is not reasonable to let SWAP check STEM and WAIT because it might turn out that the leave-thread cannot continue and must be terminated after all. Therefore, a new TL0 instruction CHECK is proposed to accomplish the aforementioned task. It is always the first instruction in a leave-thread. CHECK must examine both STEM and WAIT, and take different actions depending on their conditions. Table 4.1 specifies the four possible cases.

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 [10], 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          (1 or 2) ; branch to stop[1] or continue with leave-thread (thus SWAP)
lduh    [lcv],r_ntp   (2)      ; use delay slot to pop newly posted thread into r_ntp
```

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.

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

Table 4.1 Condition codes for the `cbs` instruction on the Main-processor

4.3 The SWAP Operation

Although the proposed hardware solution guarantees atomic execution of POST and SWAP, some questions remain about the SWAP operation itself. In the previous Sections it was mentioned that certain values are assumed to be resident in both processors. These values are `fp`, `lcv`, and `queue`. This requires that these values are actually loaded into both processors during the execution of SWAP.

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

Based on this, a hardware is required that makes the Main-processor capable of directly loading data into the Inlet-processor registers. This is a problem because the Inlet-processor is not a slave of the Main-processor. Additionally, the SPARC9 does not provide coprocessor instructions anymore [12] as opposed to the SPARC8 [10]. In order to solve this problem, a new instruction is proposed for the Main-processor which copies data from a Main-processor register into an Inlet-processor register, `movi` - *move to Inlet-processor* with the syntax:

`movi` reg_M, reg_I ; reg_I (in Inlet-processor)←reg_M (in Main-processor), 1 cycle

This instruction uses the existing hardware of the `mov` instruction[1]. Additionally, it sets a 'move to Inlet-processor' control line (MOVIP) indicating to the Inlet-processor that the Main-processor wants to write an Inlet-processor register. There will not be any interference with the execution of the Inlet-processor instructions since the Inlet-processor will be stalled a during a `SWAP` anyway. Figure 4.9 shows the hardware required for `movi`.

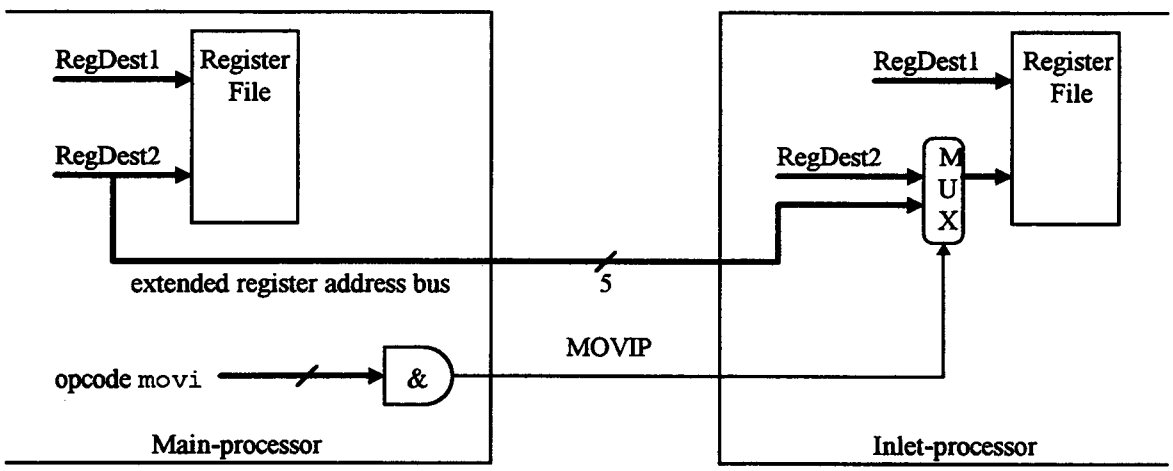

Figure 4.9 Hardware required for the `movi` instruction

The Main-processor reads reg_M (RegDst1). The 5-bit bus addressing RegDst2 is extended to the Inlet-processor. Whenever MOVIP is set, the Inlet-processor writes back the data from the MICBus into the register addressed by the extended register address bus ($_{regI}$). If there is no Inlet-processor, `movi` has no effect.

'`movi`' is executed two times during a `SWAP` operation. First, to ensure coherence of `fp` between the processors and second, to initialize `lcv` in the Inlet-processor. The complete, modified mapping for `SWAP` to the SPARC is listed in Appendix A.3.

[1] '`mov`' is a synthetic instruction. The actual instruction used on the SPARC8 would be `or`.

4.4 Enabling of Idle Frames

Using an Inlet-processor causes a problem when POST wants to enable an idle frame and queue it in the ready-frame queue. What the original POST does is depicted in Figure 4.10. It takes the content of queue[1], which points to the currently next enabled frame and stores it into the 'frame-link slot' of the inlet's frame. Then it moves ifp into queue. Thus, after the currently running frame (F_{run}) has terminated, the Main-processor will swap to the inlet's newly enabled frame (F_{in}) rather than to the frame, which was originally scheduled next (F_{next}) in the ready queue.

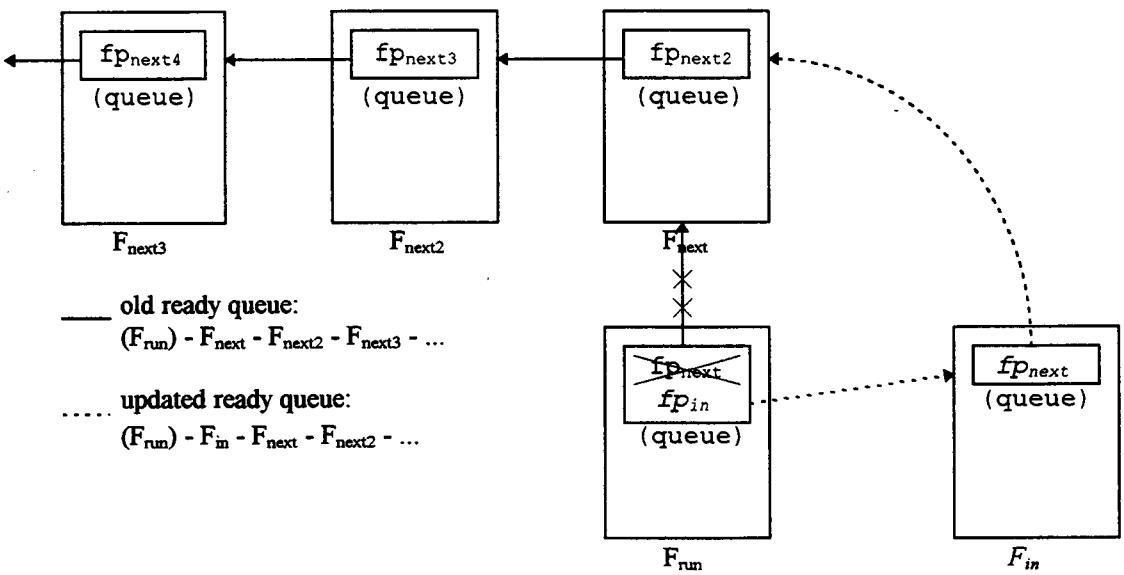

Figure 4.10 Queuing a frame in the ready queue

Note that this method first assumes that queue resides in both processors which is no problem due to the new movi instruction. In the Main-processor, queue is needed in order to switch to the next activation. In the Inlet-processor, queue is needed to queue a frame in the ready-frame queue. Since the Inlet-processor modifies queue during a queuing operation, there must be a way for it to update queue in the Main-processor as well. Fortunately, the 5-bit register address bus already exists due to the introduction of movi. So the only thing that needs to be done is to make this bus bi-directional and to add one more control line from the Inlet-processor to the Main-processor, REGW - *register write*. Thus, whenever the Inlet-processor needs to update queue, it sets REGW which stalls the Main-processor for

[1] 'queue' points to the next enabled frame in the ready queue. 'queue' is both the pointer itself and the register holding the pointer. This is no problem since the register queue always holds the pointer queue.

one cycle and makes it write the data from the MICBus into the register indicated by the register address bus. This does not cause any interference or atomicity problems, since the Main-processor accesses `queue` only during `SWAP` when the Inlet-processor is stalled anyway.

4.5 Suggestion for a Processor Node Architecture

In this Section, a complete processor node design is presented based on the proposed modifications discussed in the previous Sections (Figure 4.11). Additionally, we also propose another modification: The Inlet-processor cache holds all Heap Data and the inlet instructions. Thus, the Main-processor is isolated from Heap Data.

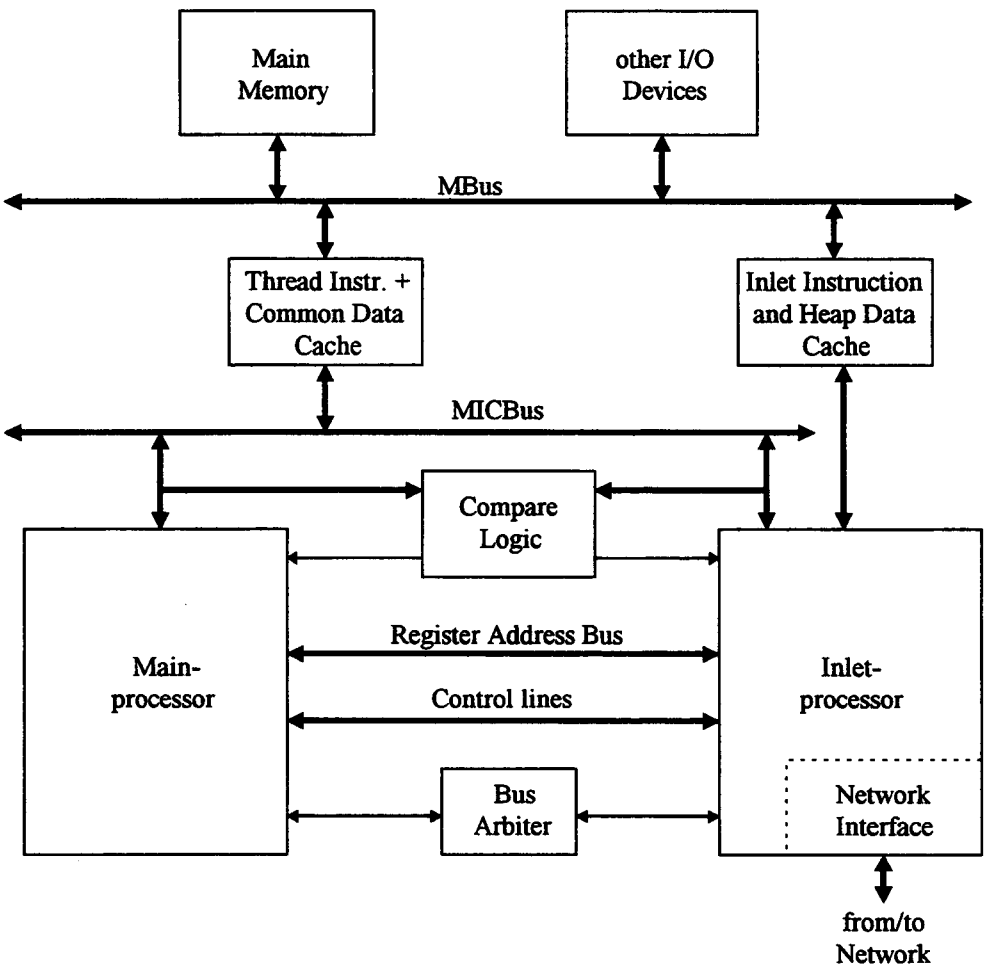

Figure 4.11 Suggestion for a processor node architecture

Our motivation to place all Heap Data only in the Inlet-processor cache and to leave the Network Interface integrated into the Inlet-processor rather than placing it on the MICBus is for the following reasons. In the previous design (Figure 3.2), the high contention on the MICBus decreases the overall system performance, since it slows down the Main-processor which carries the most part of the work load (assuming the Inlet-processor only executes inlets).

The Main-processor uses the MICBus to fetch its instructions as well as to access *any* data (frame and heap). The Inlet-processor has to access the MICBus for the Common Data Cache for both frame and heap data. Additionally, the Main-processor has to send all remote references (`SEND`, `IFETCH`, `ISTORE`, etc.) over the MICBus whether the Network Interface is actually on the MICBus or it is integrated into the Inlet-processor. The combination of allowing only the Inlet-processor to access both the Network Interface and the heap data substantially decreases the MICBus contention.

This is because all remote memory requests arriving at the Network Interface can now be handled by the Inlet-processor without the need to access the MICBus. Moreover, atomicity problem will no longer exist between the Main-processor and the Inlet-processor due to simultaneous access to the same heap element. The fact that the Main-processor has to go through the Inlet-processor to access the heap is not a problem for the following reason: The Main-processor can simply dispatch remote references and `SEND`s to the Inlet-processor by using the `movi` instruction (see next paragraph). The Inlet-processor holds several generic '`SEND` message' handlers[1]. Thus, the actual execution of `SEND`s, `IFETCH`es, `ISTORE`s, etc. takes place in the Inlet-processor, including the decision if the destination is local or remote[2]. If a message is local, the appropriate inlet is executed. Otherwise, the message is formatted and passed along to the Network Interface. This method however increases the cost for messages due to the required dispatch, but this drawback will be compensated by the fact that a large part of the workload is shifted to the Inlet-processor. The Inlet-processor executes *all* inlets, which includes the service of and replies to local and remote heap requests, which decreases the overall Main-processor time and thus increases the performance.

However, there is a cost due to the Main-processor being stalled during a message dispatch because the Inlet-processor is busy. For this reason, a short buffer in the MICBus-Inlet-processor interface is proposed. The `movi` instruction writes only remote memory accesses and `SEND`s to the buffer. In `SWAP`, `movi` writes data directly to Inlet-processor registers (by-passing the buffer). This design affects the functionality of the `next` instruction (see Section 4.2.2) on the Inlet-processor. All inlets terminate by means of `next`, which then dispatches to the next inlet or waits for the arrival of a message. Since the Main-processor dispatches all its messages over the MICBus into the buffer and remote messages can simultaneously arrive at the Network Interface, `next` has to choose between the two, if both hold

[1] All remote memory references are basically modified and extended forms of the `SEND` instruction.

[2] In this design, it is not desired to optimize local messages on the Main-processor since this would cause inlets to execute on the Main-processor which in turn would significantly degrade locality since inlet instructions are kept in the Inlet Instruction Cache.

messages. Clearly, priority is given to the Network Interface so that it can dispatch messages fast and does not unnecessarily congest the network.

Another cost is the quantum size might be decreased slightly, since the computational threads execute faster[1]. Thus, less responses due to remote memory references might return during the same quantum.

To sum up, a significant performance increase can be achieved with the proposed architecture due to the following reasons:

- Any heap access is exclusively controlled by the Inlet-processor over its own cache bus eliminating the need to go over the MICBus and thus decreasing the MICBus contention.
- A substantial part of the workload of the Main-processor has been shifted to the Inlet-processor resulting in a more balanced distribution of the workload between both processors.
- The penalty to access the Network Interface has been cut down drastically since it is integrated into the Inlet-processor.
- Atomicity between the processors is guaranteed by efficient hardware solutions avoiding substantial, additional cost of software solutions.

Note that the Main-processor is now basically isolated from all other processor nodes. It has access to the network environment only through the Inlet-processor. At the same time, the interprocessor communication is kept to a minimum. The Main-processor only needs to dispatch messages to the Inlet-processor. The different storage hierarchies of TAM make it possible to separate Heap Data from Common Data without decreasing the locality.

4.6 Changes to the SPARC

The following modifications were made to the SPARC:

- A simple bus arbitration logic is used by all memory-accessing instructions (Section 4).
- The 5-bit register address bus is extended off-chip to the Inlet-processor (bi-directional).
- This bus also requires two new control lines, REGW and MOVIP (Section 4.3 and 4.4).
- Three new instructions are incorporated into ISA which to a large extent use already existing control logic for similar instructions, so that the additional logic needed is minimal. These instructions are:

    1. `lds`: `ldb` plus sets SLOCK and, if SYNC_OK is one, it stalls (Section 4.1).

[1] So far, inlets that were responses to remote memory references were in-lined in the computational code on the Main-processor when they were local.

2. `cbs`: conditional branch using STEM and WAIT for condition and having the capability to stall (Section 4.2.2).
3. `movi`: `mov` plus sets MOVIP thus using the extended register address bus (Section 4.3).

- The `cdbp` instruction is adjusted slightly to potentially set INCLCV. Additionally, `cdbp` always resets WAIT to zero (Section 4.2.2).
- Finally, the `std` instruction is modified to potentially set DECLCV (Section 4.2.2).

Since the Inlet-processor is a completely independent coprocessor purely custom designed to primarily support TAM, we were free to design and modify it arbitrarily in a reasonable manner according to our needs.

5. SYSTEM IMPACT ANALYSIS

This Section analyzes the impact of the proposed processor node design (Figure 4.11) on the average processor time[1]. The metric used for this analysis is average *clock cycles per TL0 instruction* (CPT), which is obtained by multiplying the instruction frequency of each instruction type by its cycle cost and summing up these products. CPT can be viewed as a relative measure of the processor time because CPT simply represents the processor time divided by the number of instructions[2]. The results obtained for the modified design are compared against the average CPT on an original 64-processor CM-5 for two benchmarks, Gamteb and Paraffins [6].

The data used for this analysis was from the results of several experiments by the TAM-group at UC Berkeley. One source provides all information about the specific cycle costs of each instruction and the average CPT [6]. The other source is a table of dynamic measurements of instruction mixes and is available at a UC Berkeley FTP-site[3]. Combining the two sources makes it possible to compute the average CPT for both the Main-processor and the Inlet-processor. However, since the program execution is different each time due to TAM's dynamic nature and due to slightly different compilation policies, the data of the two sources does not match perfectly. Therefore, the results of our computation were normalized to the original average CPT, which will be the point of reference for all following calculations. The following conventions are used: *Original CPT* stands for the total, original average CPT as indicated in source 1. *Computed contribution* stands for a part of the total CPT due to a certain instruction category as computed taking into account both sources. The computed contribution still has to be normalized by multiplying it with a ratio of original CPT due to control instructions and computed contribution of control instructions. This is illustrated briefly in the following example:

> Source 1 indicates that all control instructions contribute 3.2 to the original CPT of 15.7. According to the instruction frequencies of source 2 and the instruction cycle cost from source 1, we compute a contribution to control instructions of 3.8 instead of 3.2. There is a problem when the control instructions must be divided into the ones executing on the Main-processor and the ones executing on the Inlet-processor, because their computed contribution is obviously too high. This would prohibit a comparison between the original and the modified CPT. Thus, the computed contribution of control instructions is normalized by multiplying it with 3.2/3.8. Then, it can be compared with the original CPT due to control instructions. The same method is applied to the other instruction categories.

---

[1] The Inlet-processor is assumed to provide the same performance as the SPARC.

[2] The processor time says nothing about the execution time because it does not include processor idle times due to imbalances in the distribution of the workload.

[3] ftp.cs.berkeley.edu: /ucb/TAM/sethg/dists.tar.Z

Subsection 5.1 presents the overall result of the comparison. The following subsections examine in more detail the contributions to the CPT of each instruction category. These Sections also mention the proposed designs to access synchronization counters and for the POST-SWAP interference regarding their effect on the processor time. Another issue examined is the new overhead introduced by interprocessor communication and bus contention.

5.1 Results - Overview

The overall result of the comparison is presented in Table 5.1. The TL0-instruction types and their respective percentages for both the original and the modified design are shown. For the modified design, the workload distribution over the two processors is also shown and it includes all the effects of the proposed designs (efficient access to synchronization counters, access and representation of the LCV, less expensive network access, elimination of polling, and the dispatch of SENDs and heap operations to the Inlet-processor) as well as the improvements due to the previous work (cdbp and std instructions, see Section 3.1). The reference point for all percentages (except for the workload) is the original CPT (i.e., total cycle cost of a program / total number of instructions of a program). This is the reason why the modified CPTs are so low (i.e., cycle cost on Main-processor *or* cycle cost on Inlet-processor / total number of instructions of a program). Clearly, the real CPTs will not change significantly compared to the original one, yet the presentation employed here allows for fast and easy comparison.

	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%
Orig'l 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%

Table 5.1 Distribution of processor time, original and modified

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, which accounts for about 50%/20% of the overhead cost (Gamteb/Paraffins). *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 Sections, the results for *control*, *messages*, *heap*, *overhead* and *memory* are discussed in more detail.

5.2 Distribution of Control Time

Table 5.2 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 (see Section 3.1). The new `CHECK` instruction, which performs a conditional test on the WAIT and STEM signals, slightly 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 slightly increase the cycle cost.

The double-ended representation of the LCV improves the performance because it avoids additional software cost for atomic accesses to the LCV for two reasons. First, the double-ended nature of the LCV avoids the `POST-FORK` interference altogether as far as pushing/popping thread pointers is concerned. Second, the coherence problem of having two identical `lcvs` is solved completely by the proposed design (see Section 4.2.2). The atomicity problem of having shared synchronization counters between the processors has been eliminated by the proposed synchronization hardware (see Section 4.1). Basically, the atomicity problem has been reduced to a problem of bus contention. This is true because the design allows the processors to access the bus each cycle. The only exception is when they access the same address in the same cycle. The probability of this is virtually zero. Since this cost can be viewed as a problem of bus contention, it is included in the overhead (see Section 5.4).

[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. One study shows that accessing the Network Interface (NI) can be as cheap as writing or reading a register, provided the NI is integrated into the processor [13].

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 - successf.	4	5	2.34%		2.95%	
- failed[1]	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 - successf.	6	7	1.28%		0.13%	
- failed[1]	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[2]	12	12	0.17%		0.00%	
STOP	5	4	2.49%		2.85%	
SINIT	4	4	1.74%		8.42%	
CHECK						
continued	-	2	-	0.39%	-	0.04%
terminated (plus STOP)	-	7	-	0.05%	-	0.01%
computed contribution (on Main-processor)			2.78	2.42	3.21	2.69
Inlet-processor						
POST[3] without cost for synch.						
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, only cost for synchr.						
failed	7	7	2.74%		1.13%	
successful.	5	5	2.52%		2.67%	
computed contribution (Inlet-processor)			0.93	0.91	0.65	0.65
total computed contribution due to control instructions			3.71	3.33	3.86	3.34
original CPT			13.6		15.7	
original CPT due to control instructions			3.7		3.2	
normalized CPT on Main-proc. due to control instr.			3.7	2.41	3.2	2.23
% of original CPT			27.21%	17.75%	20.38%	14.20%
normalized CPT on Inlet-proc. due to control instr.			-	0.91	-	0.54
% of original CPT			-	6.67%	-	3.43%

Table 5.2 Distribution of control time on Main-processor and on Inlet-processor

[1] The original cycle costs for failed synchronizing branches include the cost of the following STOP required. However, STOP is not needed anymore in the modified design due to the cdbp instruction.

[2] The cost for extra threads is an approximate value. The exact cycle count would be 2+N×11.

[3] The SPARC mapping of synchronizing POSTs is identical with the one for unsynchronizing POSTs except for some additional instructions needed for synchronization (see Appendix A.4). Therefore, all POSTs that actually post a thread, i.e. all unsynchronizing and successful synchronizing ones, can be combined for the purpose of this computation. Then the cost for the sequence handling the synchronization can be computed separately.

To sum up, the effect of the proposed interface design is simply that there is no new overhead cost due to atomicity or coherency despite the fact that both processors have to share common data.

As can be seen, the proposed modifications lead to 73/27 and 81/19 distributions of the workload due to control instructions for Gamteb and Paraffins, respectively. The total control overhead has been reduced by 10%/13% (Gamteb/Paraffins). The control overhead on the Inlet-processor has also been slightly reduced by 2% for Gamteb. The proposed `std` instruction saves one cycle for each `POST` to a ready or idle frame except for failed synchronizing `POST`s.

5.3 Messages

Messages consist of the cost due to two TL0 instructions: `SEND` and `RECEIVE`. Nevertheless, in order to find out the time spent on messages for each processor, replies have to be distinguished from `SEND`s, because `SEND`s originate only in the Main-processor whereas replies occur only on the Inlet-processor. Thus, Table 5.3 divides the messages cost into three main contributors.

Local `SEND`s are optimized on a uniprocessor system. Thus, local `RECEIVE`s are much cheaper since the data is already in the processor registers and it does not have to be loaded from the Network Interface. However, the data still has to be stored in the frame, which accounts for 3 cycles per word (worst case).

Table 5.3 indicates that the total time spent on messages has been decreased 25%/30% (Gamteb/Paraffins) although there is additional overhead due to the required dispatching of all `SEND`s from the Main-processor to the Inlet-processor. This improvement is achieved through a significant cost reduction in order to access the Network Interface. Instead of 8 cycles needed to load/store a doubleword from/into the Network Interface, it is now 2 cycles for moving two words into/from a Network Interface register from/into an Inlet-processor register. Thus, all remote overhead, which includes pushing the two words `fp` and `ip`, is reduced by 6 cycles. Similarly, each remote push/pop word is reduced by 3 cycles[1].

To dispatch a `SEND`, the Main-processor moves one word for the instruction type (here "`SEND`"), two words for the destination (`fp`, `ip`), and one to three words of arguments to the Inlet-processor. This operation accounts for a 3 cycle-cost for the overhead and a cost of 1 cycle per argument. Since dispatching requires additional cost due to interprocessor communication, it is included in *overhead* instead of in *messages* (see Table 5.1 and 5.5). One example for a `SEND` instruction is:

```
SEND pfslot1.pf[0.i/FIB.pc] <- ireg0.i
```

The argument to be sent resides in register `ireg0.i`. '`0.i/FIB.pc`' points to inlet0 of the code-block `FIB`. Finally, the register `pfslot1.pf` holds the frame pointer of the code-block `FIB`.

[1] The cycle cost for the overhead of local `RECEIVE`s is estimated since accurate data was not available. It is obtained by subtracting 8 from the overhead for remote `RECEIVE`s taking into account that `fp` and `ip` are already in registers, so that one load-doubleword from the NI is spared.

	cycle cost			Gamteb in % of TL0-instr.		Paraffins in % of TL0-instr.	
	orig.	modified		original	modified	original	modified
		Main.	Inlet.				
SEND							
local - overhead	4	3	4	1.10%		0.01%	
- push one word	1	1	1	1.10%		0.01%	
remote - overhead	25	3	19	2.15%		0.11%	
- push one word	4	1	1	2.16%		0.15%	
Reply (SEND)							
local - overhead	4	-	4	0.13%		0.00%	
- push one word	1	-	1	0.26%		0.01%	
remote - overhead	25	-	19	0.26%		0.04%	
- push one word	4	-	1	0.51%		0.11%	
RECEIVE							
local - overhead	5	-	5	1.36%		0.01%	
- push one word	3	-	3	1.36%		0.02%	
remote - overhead	13	-	7	2.67%		0.19%	
- push one word	6	-	3	2.67%		0.26%	
computed contribution (on Main-processor)				1.39	0.13	0.09	0.01
computed contribution (on Inlet-processor)				-	0.92	-	0.05
total computed contribution due to messages				1.39	1.05	0.09	0.06
original CPT				13.6		15.7	
original CPT due to messages				0.8		0.1	
normalized CPT on Main-processor due to messages				0.8	0.07	0.1	0.01
% of original CPT				5.88%	0.55%	0.64%	0.07%
normalized CPT on Inlet-processor due to messages				-	0.53	-	0.06
% of original CPT				-	3.89%	-	0.35%

Table 5.3 Distribution of message cost on Main-processor and on Inlet-processor

5.4 Heap

The heap cost includes all time spent on allocation, control, and access of the heap. Each of the heap operations has two parts: The first part is the request operation which is basically a SEND from an executing thread. The second part is the service operation. For ISTOREs and IFREEs, only one RECEIVE is part of the service whereas for IFETCHes and IALLOCs, the service operation also replies (basically a SEND) with the requested value. This implies that another RECEIVE is needed in the inlet that finally receives the reply.

Thus, the cycle savings for SENDs and RECEIVEs discussed in Section 5.3 can be applied to all heap messages as well. It is assumed that each SEND/RECEIVE usually has 4 words to store/load to/from the Network Interface (NI). For each word, 3 cycles are saved (see Section 5.3). Table 5.4 summarizes the frequency and the amount of cycles saved for each heap message. Since all heap instructions occur in threads (which execute on the Main-processor), they all must be dispatched to the Inlet-processor. A

general 5-cycle penalty is assumed for each dispatch. This is reasonable since one word must indicate the instruction type, two words the destination and another two words contain either the return address or the data (heap elements are 64-bit wide).

There are no cycles saved for local heap operations since they do not access the Network Interface. The overall reduction of the time spent on heap operations is 22%/14% (Gamteb/Paraffins), which is enormous. Due to the high percentage of heap instructions, the effect on the original CPT even without an Inlet-processor would be 7%. This illustrates the benefits of bringing the Network Interface closer to the processor and thus avoiding many expensive uncached loads and stores.

Since the exact mappings of the heap operations are not available, the numbers in the following table cannot be normalized which might slightly increase the error of this analysis. On the other hand, a 3 cycle saving for each word pushed/popped to/from the NI is a conservative estimate, since for all single-word load/stores from/to the NI 6 cycles are saved which has not been considered[1].

	cycles for dispatch (Mainp.)	cycle savings (Inletpr.)	Gamteb in % of TL0-instr.		Paraffins in % of TL0-instr.	
IFETCH						
local	5	0	0.95%		0.38%	
remote (2 SENDs, 2 RECEIVEs)	5	-48	2.66%		3.88%	
ISTORE						
local	5	0	2.16%		8.67%	
remote (1 SEND, 1 RECEIVE)	5	-24	0.02%		0.00%	
IALLOC						
local	5	0	0.31%		2.68%	
remote (2 SENDs, 2 RECEIVEs)	5	-48	0.0%		0.00%	
IFREE						
local	5	0	0.16%		0.00%	
remote (1 SEND, 1 RECEIVE)	5	-24	0.15%		0.00%	
additional contribution to CPT (Main-processor)			0.32		0.78	
reduction of contribution to CPT (Inlet-processor)			-1.32		-1.86	
overall effect on original CPT			-1.00		-1.08	
original CPT			13.6		15.7	
original CPT due to heap instructions			4.6		7.7	
			original	modifd.	original	modifd.
CPT on Main-processor due to heap instructions			4.60	0.32	7.70	0.78
% of original CPT			33.82%	2.35%	49.04%	4.97%
CPT on Inlet-processor due to heap instructions			-	3.28	-	5.84
% of original CPT			-	24.12%	-	37.20%

Table 5.4 Distribution of heap cost on Main-processor and on Inlet-processor

[1] It takes 7 cycles to load/store a single word from/to the NI, whereas the load/store doubleword from/to the NI takes 8 cycles.

5.5 Overhead

Overhead contains all the costs incurred specifically due to the introduction of the Inlet-processor[1]. These costs include the extra dispatch time required for messages (see Section 5.3) and heap operations (see Section 5.4) by the Main-processor and stalls caused by both processors due to MICBus contention and POST-SWAP interference. Table 5.5 shows the breakdown of the overhead for the Main-processor.

The cost 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 5.5) 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. In reality, the cost will be less since SWAP will not always occur at the beginning of a POST. There is also an overhead cost for the Inlet-processor since it has to stall while the Main-processor is executing a SWAP. The first instruction of the common leave thread is always CHECK[2] potentially followed by register-saves and finally followed by SWAP. Thus, the Inlet-processor sees at least the full penalty of CHECK (when continued) and SWAP instructions. CHECK, when terminated, never sets HOLD, thus it does not stall the Inlet-processor.

Main-processor	cycle cost	Gamteb in % of TL0-instr.	Paraffins in % of TL0-instr.
Stalls due to MICBus-accesses of Inlet-pr.			
POST (without cost for synchr.)			
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	7.96%	7.22%
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 (see 5.3)		2.35%	4.97%
heap-dispatch cost (see 5.4)		0.55%	0.07%
Main-processor overhead in % of original CPT		6.80%	7.21%

Table 5.5 Overhead cost on Main-processor

[1] Since it is a new cost, the overhead CPT cannot be normalized, so its absolute value is taken.
[2] In the final leave thread, the CHECK instruction stands after SEND and FFREE.

Inlet-processor	cycle cost	Gamteb in % of TL0-instr.	Paraffins in % of TL0-instr.
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%

Table 5.6 Overhead cost on Inlet-processor

Note 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 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.

5.6 Memory

The memory cost considers the penalty due to a 5% cache-miss rate on general memory accesses. The only exception are loads bringing operands into the ALU, which is indicated by *operands*. The memory penalty accounts for about 14% of the original processor time for both Gamteb and Paraffins. The time spent on each processor in the modified design is used to divide the memory cost up onto the Main-processor and onto the Inlet-processor. Not included in this computation are *operands* and the part of *overhead* coming from bus contention and stalls (see Section 5.5) since they do not represent memory accesses.

	Gamteb in % of original processor time		Paraffins in % of original processor time	
	Main-proc.	Inlet-proc.	Main-proc.	Inlet-proc.
ALU	5.15%	-	1.27%	-
Heap	-	24.12%	-	37.20%
Messages	-	3.89%	-	0.35%
Overhead	2.90%	-	5.04%	-
Control	17.75%	6.67%	14.20%	3.43%
% of CPT causing memory penalty, each processor	25.80%	34.68%	20.51%	40.98%
combined % of CPT causing memory penalty	60.48%		61.49%	
relative distribution of CPT causing memory penalty on processors	42.66%	57.34%	33.36%	66.64%
original CPT	13.6		15.7	
original CPT due to mem. penalty	1.9		2.2	
% of original CPT	13.97%		14.01%	
distribution of CPT due to memory penalty on processors in % of original CPT	5.96%	8.01%	4.67%	9.34%

Table 5.7 Distribution of memory penalty on Main-processor and on Inlet-processor

6. CONCLUSION AND FUTURE OUTLOOK

In this thesis, some of the most fundamental problems occurring due to the interaction of the Main-processor and the Inlet-processor were presented and solutions to these problems were proposed. The design to access synchronization counters and the representation of the LCV as a double-ended stack reduce the problem of atomicity between `FORK` and `POST` to a problem of bus contention eliminating completely the need for the programmer or compiler to use software mechanisms. These solutions increase performance by avoiding additional cycle cost due to more expensive atomic instructions that would be needed without hardware support for atomic memory accesses. A software solution would increase the Main-processor's CPT by more than 10% (see Section 4.1). The design required to guarantee atomicity between `SWAP` and `POST` also improves the performance. This is achieved by providing hardware to solve the problem which keeps down the additional cost due to necessary processor interaction. The alternative solution as discussed in Section 4 to post a thread pointer for the running frame in the frame's RCV instead of the LCV would require significantly more `SWAP` operations thus increasing the Main-processor workload.

The architectural proposals minimize the time penalties due to necessary processor interaction and communication. At the same time, the additional hardware and the modifications were successfully kept to a minimum so as not to disturb the original functionality of the SPARC. Although the modifications are many, this does not invalidate the aforementioned argument since most of the changes are minimal and build on already existing logic.

The need to add a bi-directional register address bus in order to enable *both* processors to write to the other one's register reconfirms the basic mismatch between conventional processors and the requirements for the interaction with a message-handling processor in the context fine-grain multithreading. Although many processors support a coprocessor, this does not help since coprocessors are always treated as slaves whereas the Inlet-processor must be viewed as a tightly-coupled, yet independent and equal processor.

The analysis in Section 5 proves that an Inlet-processor to handle messages significantly reduces the workload of the Main-processor. Under TAM, this is even more evident, since TAM's storage hierarchy allows the Inlet-processor to execute all heap operations as well without sacrificing locality. The results clearly confirm that it is essential for the success of executing of fine-grain programs to bring the Network Interface close to the processor due to the large message overhead. We chose to integrate the Network Interface completely into the Inlet-processor rather than connecting it to the MICBus to avoid further bus contention caused by the additional traffic of accessing the network. If the MICBus contention could be significantly decreased by letting the Main-processor have an own instruction cache, then it might be more attractive to have the Network Interface on the MICBus. A variation of this would be to

dispatch incoming messages over an extra bus to the Inlet-processor (since it is the only one receiving messages), but to still send all outgoing messages (form both processors) to the Network Interface over the MICBus. This approach would be similar to the conceptual node design of the *T [1].

The analysis illustrates clearly the benefit of dispatching all messages (including heap requests) to the Inlet-processor, which is possible since the Inlet-processor is assumed to have the same, basic datapath as the Main-processor. This method causes the workload to be much more balanced (for Gamteb even close to 50/50) and thus will improve the overall performance significantly.

To sum up, the two major implications of this work are: First, a separate processor to handle messages for fine-grain parallelism improves the performance significantly by releasing the Main-processor of a large percentage of the workload. The separate processor should provide about the same performance and datapath as the Main-processor. However, it needs to be tailored to the special need of the fine-grain execution model. Second, despite the necessity for minor modifications, stock processors can be used efficiently to support fine-grain parallelism without the need to compromise their original functionality.

The way the node design is implemented plays a decisive role in the performance and thus the success of fine-grain multithreading. Therefore, areas of future research are to examine the use of different node architectures and different processors in terms of their impact on the performance of fine-grain execution models (e.g., a separate instruction cache for the Main-processor). Another challenge is the implementation of an Inlet-processor. The question is if the design should follow conventional processor architectures or if it would be more reasonable to follow a new approach determined by the distinct purpose to handle messages.

BIBLIOGRAPHY

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

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[3] G. M. Papadopoulos et al., "*T: Integrated Building Blocks for Parallel Computing," *Proc. Supercomputing 93*, 1993, pp. 624-635.

[4] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.

[5] B. Lee, A. R. Hurson, "Dataflow Architectures and Multithreading," *IEEE Computer*, August 1994.

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

[7] D. E. Culler et al., "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. 4th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1991.

[8] E. Spertus et al., "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," *Proc. of the 20th Annual Int'l Symp. on Computer Architecture*, 1993.

[9] S. Kotikalapoodi, "Fine-Grain Parallelism on Sequential Processors," MS thesis, Oregon State University, September 2, 1994.

[10] SPARC International, Inc., Menlo Park, CA, *The SPARC Architecture Manual Version 8*, Englewood Cliffs: Prentice Hall, Inc., 1992.

[11] B. J. Catanzaro, *The SPARC Technical Papers*, Springer Verlag, 1991.

[12] SPARC International, Inc., Menlo Park, CA, *The SPARC Architecture Manual Version 9*, Englewood Cliffs: Prentice Hall, Inc., 1993.

[13] D. S. Henry and C. F. Joerg, "The Network Interface Chip," *Computation Structures Group at MIT*, Memo 331, June 1991.

APPENDICES

A. TL0-INSTRUCTION MAPPINGS TO THE SPARC PROCESSOR

This section lists the mappings of some TL0 instructions that are relevant to this work. Specifically, these instructions are FORK (including SWITCH), STOP, SWAP, and POST. The numbers in brackets indicate the cycle cost for each assembly instruction. Both the original mappings used in [6] and the modified mappings are shown to illustrate the changes made.

A.1 FORK mappings

SWITCH is not listed since it is identical to the FORK mapping. The only difference is that SWITCH has additionally a conditional branch at the beginning of the FORK code accounting for an extra 2 cycles to be added to the cost of FORKs.

A.1.1 Original FORK mapping

Branch to an unsynchronizing thread:

```
ba      thr_addr            (1)     ; branch to thread address
```

Branch to a synchronizing thread:

```
ldb     sync[fp],tmp1       (2)     ; load the synchronization counter into register tmp1
subcc   tmp1,1,tmp1         (1)     ; decrement the synchronization counter
be      thr_addr            (1 or 2) ; if count zero, branch and annul stb in delay slot
stb     tmp1,sync[fp]       (3)     ; store back the (non-zero) synchronization counter
```

Note that failed synchronizing branches require a STOP *additionally.*

Push an unsynchronizing thread:

```
set     Lthr-cbbase,tmp2    (1)     ; tmp2 gets thread pointer
sth     tmp2,[lcv]          (3)     ; push thread pointer on LCV stack
sub     lcv,2,lcv           (1)     ; decrement the pointer to the top of the LCV (lcv)
```

Push a synchronizing thread:

```
ldb     sync[fp],tmp1       (2)     ; load the synchronization counter into register tmp1
subcc   tmp1,1,tmp1         (1)     ; decrement the synchronization counter
bnz,a   continue            (1 or 2) ; if tmp1 ≠ 0, branch and execute stb in delay slot
 stb    tmp1,sync[fp]       (3)     ; store back the (non-zero) synchronization counter
set     Lthr-cbbase,tmp2    (1)     ; tmp2 gets thread pointer
sth     tmp2,[lcv]          (3)     ; push thread pointer on LCV stack
sub     lcv,2,lcv           (1)     ; decrement the pointer to the top of the LCV (lcv)
continue:
```

A.1.2 Previous FORK mapping

This is the modified mapping according to S. Kotikalapoodi [9]. It includes the proposed `cdbp` and `std` instructions.

Branch to an unsynchronizing thread:

```
        ba      thr_addr            (1)     ; branch to thread address
```

Branch to a synchronizing thread:

```
        ldb     sync[fp],tmp1       (2)     ; load the synchronization counter into register tmp1
        subcc   tmp1,1,tmp1         (1)     ; decrement the synchronization counter
        cdbp    thr_addr          (1 or 2)  ; if count zero, branch and annul stb in delay slot
        stb     tmp1,sync[fp]       (3)     ; store back the (non-zero) synchronization counter
```

Note that failed synchronizing branches do not require a following `STOP` anymore.

Push an unsynchronizing thread:

```
        std     r_ntp,[lcv]         (3)     ; first push existing thread pointer on LCV
        set     Lthr-cbbase,r_ntp   (1)     ; r_ntp gets new thread pointer
```

Push a synchronizing thread:

```
        ldb     sync[fp],tmp1       (2)     ; load the synchronization counter into register tmp1
        subcc   tmp1,1,tmp1         (1)     ; decrement the synchronization counter
        bnz,a   continue          (1 or 2)  ; if tmp1 ≠ 0, branch and execute stb in delay slot
         stb    tmp1,sync[fp]       (3)     ; store back the (non-zero) synchronization counter
        std     r_ntp,[lcv]         (3)     ; first push existing thread pointer on LCV
        set     Lthr-cbbase,r_ntp   (1)     ; r_ntp gets new thread pointer
continue:
```

A.1.3 Final, modified FORK mapping

The only difference to the previous `FORK` mapping (A.1.2.) is that `ldb sync,[fp]` is substituted by `lds sync,[fp]`. '`lds`' is required in order to set the SYNC control line.

Branch to an unsynchronizing thread:

```
        ba      thr_addr            (1)     ; branch to thread address
```

Branch to a synchronizing thread:

```
        lds     sync[fp],tmp1          (2)     ; load the synchronization counter into register tmp1
        subcc   tmp1,1,tmp1            (1)     ; decrement the synchronization counter
        cdbp    thr_addr               (2 or 3) ; if tmp1 ≠ 0, branch to thr_addr and annul stb
                                                  in delay slot, else branch to thread pointed to by
                                                  r_ntp and pop next thread pointer into r_ntp
        stb     tmp1,sync[fp]          (3)     ; store back the (non-zero) synchronization counter
```

Push an unsynchronizing thread:

```
        std     r_ntp,[lcv]            (3)     ; first push existing thread pointer on LCV
        set     Lthr-cbbase,r_ntp      (1)     ; r_ntp gets new thread pointers
```

Push a synchronizing thread:

```
        lds     sync[fp],tmp1          (2)     ; load the synchronization counter into register tmp1
        subcc   tmp1,1,tmp1            (1)     ; decrement the synchronization counter
        bnz,a   continue               (1 or 2) ; if tmp1 ≠ 0, branch and execute stb in delay slot
         stb    tmp1,sync[fp]          (3)     ; store back the (non-zero) synchronization counter
        std     r_ntp,[lcv]            (3)     ; first push existing thread pointer on LCV
        set     Lthr-cbbase,r_ntp      (1)     ; r_ntp gets new thread pointer
continue:
```

A.2 STOP mappings

There is only two STOP mappings, the original one and the one modified by S. Kotikalapoodi. STOP has not been further modified in this work.

A.2.1 Original STOP mapping

```
        lduh    [2+lcv],tmp            (2)     ; load next thread pointer from LCV
        add     lcv,2,lcv              (1)     ; increment the pointer to the top of the LCV (lcv)
        jmp     [tmp+cbbase]           (2)     ; compute absolute thread address and jump there
```

A.2.2 Modified STOP mapping

```
        xnorcc          g0,g0,g0       (1)     ; clear the zero flag
        cdbp            nowhere        (3)     ; unconditionally branch to thread pointed to by
                                                  r_ntp and pop next thr. offset addr. into r_ntp
```

A.3 SWAP mappings

In the following the original and the modified SWAP mappings are shown. However; since we had only some fractions of the original SWAP mapping, the code below might differ from the 'real, original" mapping. Here is a rough description of what the code does: First, the frame pointer (fp) is replaced. Second, the pointers to the next ready frame (queue), to the codeblock base address (cbbase) and to the top of the RCV (rcv) are loaded. Then the leave-thread and the first three computational thread pointers are copied from the RCV onto the LCV and rcv is reset to fp (which indicates the RCV is empty). If there are still more thread pointers in the RCV, then another four tread pointers are copied and so on until the RCV is empty. Also, the pointer to the top of the LCV (lcv) is updated. Finally, SWAP transfers control to the enter-thread. Frame slots holding specific variables, e.g. rcv, are called 'F' plus the name of the variable, e.g. Frcv.

A.3.1 Original SWAP mapping

```
      mov    queue,fp              (1)    ; setup fp register ← head of ready frame queue
      ld     Fqueue[fp],queue      (2)    ; queue ← next ready frame pointer
      ld     Fcbbase[fp],cbbase    (2)    ; setup thread address base register
      ld     Frcv[fp],tmp1         (2)    ; tmp1 ← pointer to top of RCV (rcv)
      ldfd   -4[fp],ftmp           (3)    ; get 4 bottom thread pointers (one is leave-thread)
      sub    lcv,2,lcv             (1)    ; adjust pointer to top of LCV (lcv)
      stfd   ftmp,-4[lcv]          (4)    ; stash the 4 bottom thread pointers
      sub    fp,tmp1,tmp1          (1)    ; tmp1← number of thread pointers in RCV
      cmp    tmp1,6                (1)    ; number of computational threads ≤ 3(each 2bytes)?
      lduh   Fenter[fp],tmp2       (2)    ; tmp2 ← enter-thread pointer
      st     fp,Frcv[fp]           (3)    ; reset rcv
      ble,a  lessthan4          (1 or 2)  ; if RCV empty, execute delay slot and branch
       sub    lcv,tmp1,lcv         (1)    ; adjust lcv
      set    -12,tmp3              (1)    ; initialize loop counter (copy-loop)
morethan4:
      ldfd   tmp3[fp],ftmp         (3)    ; get next 4 thread pointers
      addcc  tmp1,tmp3,tmp4        (1)    ; tmp4 ← number of thread pointers left
      stfd   ftmp,tmp3[lcv]        (4)    ; stash the next 4 thread pointers
      cmp    tmp4,2                (1)    ; any thread pointers left ?
      bg,a   morethan4          (1 or 2)  ; if yes, execute delay slot and branch to morethan4
       sub    tmp3,8,tmp3          (1)    ; decrement loop counter
      sub    lcv,tmp1,lcv          (1)    ; adjust lcv
lessthan4:
      jmp    [tmp2+cbbase]         (2)    ; jump to enter-thread
```

Note: The leave-thread pointer is always at the bottom of the RCV (in Fleave) which is at +2[fp]. The first computational thread pointer is at [fp], the next at -2[fp] and so on.

A.3.2 Modified SWAP mapping

This is the final, modified SWAP instruction mapping. Note the changes compared to the original SWAP: queue and lcv are loaded into both processors by means of the proposed movi instruction. The leave-thread pointer is loaded explicitly into r_ltp. This is why the pointers to the first *four* (instead of three) computational threads are copied from the RCV onto the LCV during the first copy-process. Finally, the cdbp instruction is used to jump to the enter-thread. The following assumptions are made. They are not further explained in detail.

- A register lcvbase exists in both processors which holds the base address of the LCV.
- The HOLD control line automatically initiates two immediate processes in the Inlet-processor:
  - Reset lcvend by moving lcvbase to lcvend.
  - Set up the new fp by moving queue to fp.

```
        mov    queue,fp              (1)    ; setup fp register ← head of ready frame queue
        ld     Fqueue[fp],queue      (2)    ; queue ← next ready frame pointer
        movi   queue_M,queue_I       (1)    ; move new queue to Inlet-processor
        ld     Fcbbase[fp],cbbase    (2)    ; setup thread address base register
        ld     Frcv[fp],tmp1         (2)    ; tmp1 ← pointer to top of RCV (rcv)
        ldfd   -6[fp],ftmp           (3)    ; get 4 bottom thread pointers (all computational!)
        sub    lcvbase,2,lcv         (1)    ; adjust pointer to top of LCV (lcv)
        stfd   ftmp,-6[lcv]          (4)    ; stash the 4 bottom thread pointers
        sub    fp,tmp1,tmp1          (1)    ; tmp1← number of thread pointers in RCV
        cmp    tmp1,8                (1)    ; number of computational threads ≤ 4(each 2bytes)?
        lduh   Fenter[fp],r_ntp      (2)    ; r_ntp ← enter-thread pointer
        lduh   Fleave[fp],r_ltp      (2)    ; r_ltp ← leave-thread pointer
        st     fp,Frcv[fp]           (3)    ; reset rcv
        ble    lessthan5             (1 or 2) ; branch, if RCV empty
        set    -14,tmp2              (1)    ; initialize loop counter (copy-loop)
morethan5:
        ldfd   tmp2[fp],ftmp         (3)    ; get next 4 thread pointers
        addcc  tmp1,tmp2,tmp3        (1)    ; tmp3 ← number of thread pointers left (+2)
        stfd   ftmp,tmp2[lcv]        (4)    ; stash the next 4 thread pointers
        cmp    tmp3,2                (1)    ; any thread pointers left ?
        bg,a   morethan5             (1 or 2) ; if yes, execute delay slot and branch to morethan5
         sub    tmp2,8,tmp2          (1)    ; decrement loop counter
lessthan5:
        subcc  lcv,tmp1,lcv          (1)    ; adjust lcv (also sets zero-flag)
        movi   lcv_M,lcv_I           (1)    ; move lcv to Inet-procssor
        cdbp   nowhere               (3)    ; unconditionally branch to enter-thread (in r_ntp)
                                            and pop next thread pointer into r_ntp
```

A.4 POST mappings

Since the main parts of POSTs to both synchronizing and unsynchronizing threads are identical, only one mapping is shown. The only difference of the synchronizing POST is additional code at the beginning (4 instructions) for the synchronization operation. A brief description of what POST does: First, it determines if the inlet is for the running frame. If yes, thread pointers are posted to the LCV. Otherwise, thread pointers are posted to the RCV. In the latter case, if the RCV was empty (i.e. the frame was idle), then the frame pointer is queued in the frame ready queue. For POSTs to synchronizing threads, the appropriate synchronization counter is decremented first. If it becomes zero, the thread pointer is posted. If it is not zero, the decremented synchronization counter is stored back only. Note that thread pointers are 16-bit offsets of cbbase.

A.4.1 Original POST mapping

Code for POSTs to *synchronizing* threads only:

```
      ldb    sync[ifp],tmp1        (2)     ; tmp1 ← synchronization counter
      subcc  tmp1,1,tmp1           (1)     ; decrement synchronization counter
      bne,a  continue              (1 or 2) ; if count ≠ 0, execute delay slot + branch to continue
       stb    tmp1,sync[fp]        (3)     ; store back decremented synchronization counter
```

Code for *all* POSTs:

```
      cmp    fp,ifp                (1)     ; is inlet's frame running?
      set    Lthr-Lcbbase,tmp2     (1)     ; tmp2 ← thread pointer
      be     isrunning             (1 or 2) ; if inlet's frame running, branch
      ld     Frcv[ifp],tmp3        (2)     ; tmp3 ← pointer to top of RCV (rcv)
      sth    tmp2,[tmp3]           (3)     ; push thread pointer
      cmp    tmp3,ifp              (1)     ; is inlet's frame idle?
      sub    tmp3,2,tmp3           (1)     ; update rcv (in tmp3)
      st     tmp3,Frcv[ifp]        (3)     ; store back adjusted pointer
      bnz    continue              (0)[1]  ; if frame is not idle, branch to continue
      st     queue,Fqueue[ifp]     (3)     ; store back old ready frame link
      mov    ifp,queue             (1)     ; make inlet's frame next ready frame
      jmp    continue              (0)[1]  ; jump to continue
isrunning:
      sth    tmp2,[lcv]            (3)     ; push thread pointer onto LCV
      sub    lcv,2,lcv             (1)     ; update pointer to top of LCV (lcv)
continue:
```

[1] The cost for these instructions is included in the cost for inlet overhead as indicated in Table 2.1.

A.4.2 Modified POST mapping

POST changed due to the introduction of std [9]. Note that std automatically updates the pointer to the top of the RCV (rcv).

Code for POSTs to *synchronizing* threads only:

```
      ldb    sync[ifp],tmp1        (2)     ; tmp1 ← synchronization counter
      subcc  tmp1,1,tmp1           (1)     ; decrement synchronization counter
      bne,a  continue              (1 or 2) ; if count ≠ 0, execute delay slot + branch to continue
       stb    tmp1,sync[fp]        (3)     ; store back decremented synchronization counter
```

Code for *all* POSTs:

```
      cmp    fp,ifp                (1)     ; is inlet's frame running?
      set    Lthr-Lcbbase,tmp2     (1)     ; tmp2 ← thread pointer
      be     isrunning             (1 or 2) ; if inlet's frame running, branch
      ld     Frcv[ifp],tmp3        (2)     ; tmp3 ← pointer to top of RCV (rcv)
      cmp    tmp3,ifp              (1)     ; is inlet's frame idle?
      std    tmp2,[tmp3]           (3)     ; push thread pointer and update rcv
      st     tmp3,Frcv[ifp]        (3)     ; store back adjusted pointer
      bnz    continue              (0)[1]  ; if frame is not idle, branch to continue
      st     queue,Fqueue[ifp]     (3)     ; store back old ready frame link
      mov    ifp,queue             (1)     ; make inlet's frame next ready frame
      jmp    continue              (0)[1]  ; jump to continue
isrunning:
      sth    tmp2,[lcv]            (3)     ; push thread pointer onto LCV
      sub    lcv,2,lcv             (1)     ; update pointer to top of LCV (lcv)
continue:
```

[1] The cost for these instructions is included in the cost for inlet overhead as indicated in Table 2.1.

B. MAPPING OF ACTIVATION FRAME AND LCV TO MEMORY

B.1 The Activation Frame

Figure B.1 illustrates how the activation frame is mapped on the local memory. The sequence how synchronization counters, local variables, `Fqueue`, `Fcbbase`, and `Frcv` mapped on the memory as described by Figure B.1 might differ from the actual implementation in [6] since we do not have the accurate data available. Conceptually, this is not important anyway. However; we do have the exact data (derived from `SWAP` mapping in A.1.3.) about where the RCV starts (`fp`) and to what direction it extends (< `fp`).

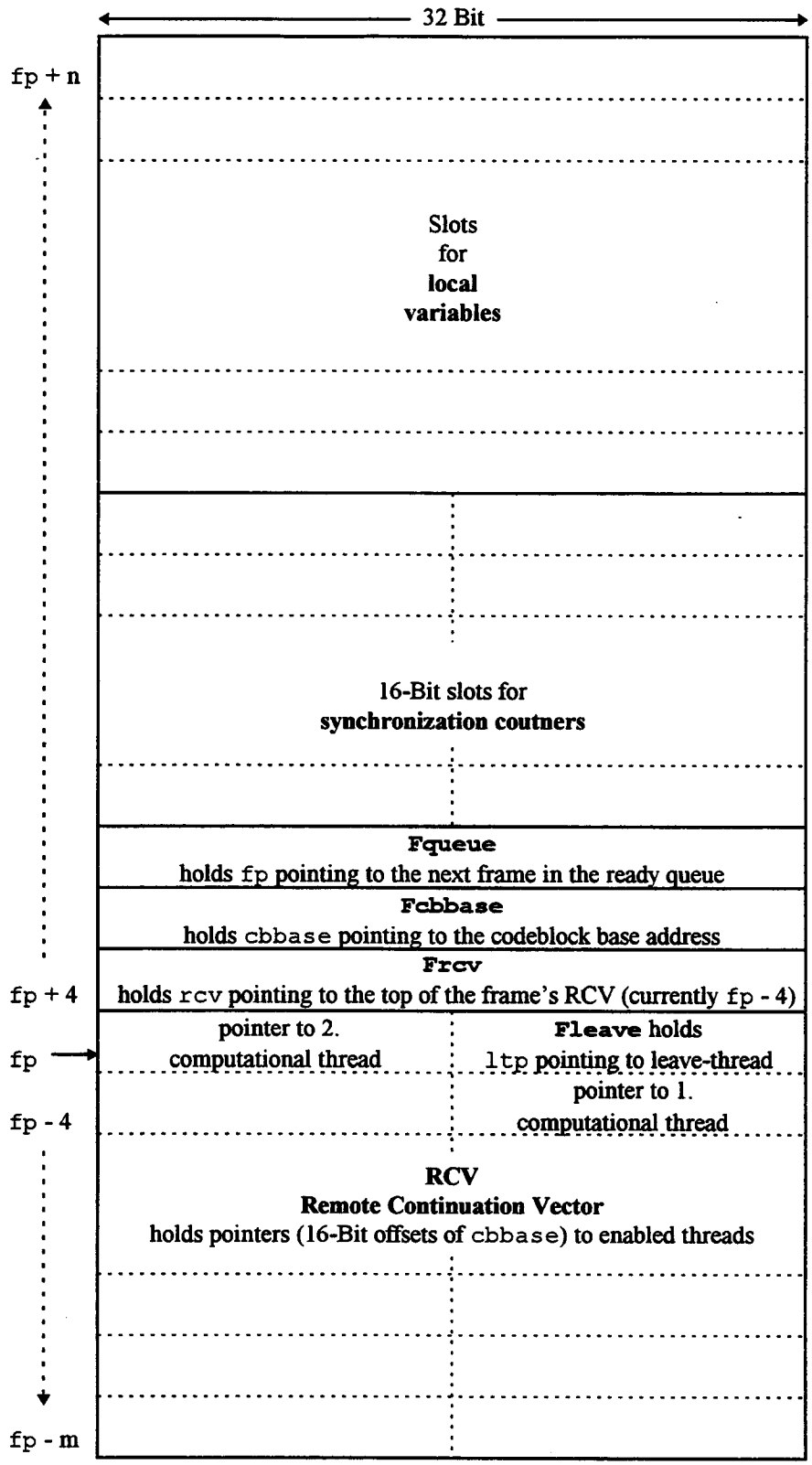

Figure B.1 Representation of the activation frame in the memory

B.2 The LCV

Figure B.2 illustrates a specific example of the mapping of the modified LCV to the memory. The example shows how the LCV would look like after SWAP had copied the RCV (as indicated in Figure B.1) onto the LCV. The pointers relevant to the LCV are depicted as well.

- The pointer to the bottom of the LCV—`lcvend`. It is needed in the Inlet-processor to append enabled threads to the LCV.
- The pointer to the top of the LCV—`lcv`. It resides in both processors. In the Main-processor it is needed to push threads on top of the LCV. In the Inlet-processor it is needed to be compared with `lcvend` in order to determine if the LCV has been emptied.
- The pointer to the base address of the LCV—`lcvbase`. It is needed in both processors to reset `lcv` and `lcvend` (see also A.3.2.).

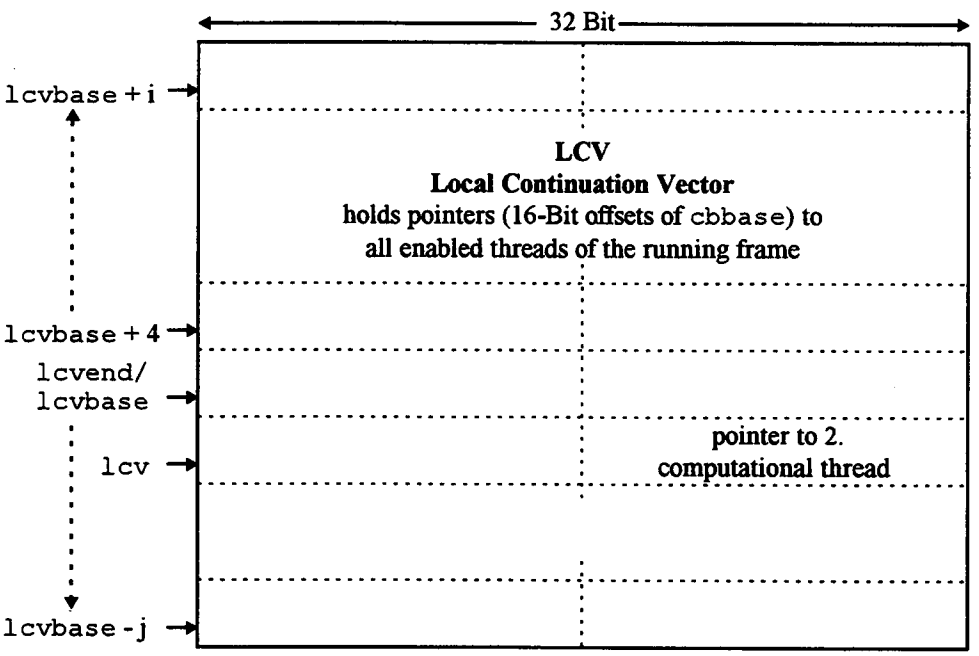

Figure B.2 Representation of the LCV in the memory

Note: First, the leave-thread pointer (`ltp`) is not at the LCV bottom, but in the register `r_ltp`, which is specifically assigned to hold `ltp`. Second, the pointer to the first computational thread has already been popped into `r_ntp` by `cdbp` at the end of SWAP (see A.3.2.).