AN ABSTRACT OF THE THESIS OF

Sridhar V. Kotikalapoodi for the degree of Master of Science in Electrical and Computer Engineering presented on September 7, 1994.

Title: Fine–Grain Parallelism On Sequential Processors

Abstract approved: _____

Dr. Ben Lee

There seems to be a consensus that future Massively Parallel Architectures will consist of a number nodes, or processors, interconnected by high–speed network. Using a von Neumann style of processing within the node of a multiprocessor system has its performance limited by the constraints imposed by the control–flow execution model. Although the conventional control–flow model offers high performance on sequential execution which exhibits good locality, switching between threads and synchronization among threads causes substantial overhead. On the other hand, dataflow architectures support rapid context switching and efficient synchronization but require extensive hardware and do not use high–speed registers.

There have been a number of architectures proposed to combine the instruction-level context switching capability with sequential scheduling. One such architecture is Threaded Abstract Machine (TAM), which supports fine–grain interleaving of multiple threads by an appropriate compilation strategy rather than through elaborate hardware. Experiments on TAM have already shown that it is possible to implement the dataflow execution model on conventional architectures and obtain reasonable performance. These studies also show a basic mismatch between the requirements for fine–grain

parallelism and the underlying architecture and considerable improvement is possible through hardware support.

This thesis presents two design modifications to efficiently support fine–grain parallelism. First, a modification to the instruction set architecture is proposed to reduce the cost involved in scheduling and synchronization. The hardware modifications are kept to a minimum so as to not disturb the functionality of a conventional RISC processor. Second, a separate coprocessor is utilized to handle messages. Atomicity and message handling are handled efficiently, without compromising per–processor performance and system integrity. Clock cycles per TAM instruction is used as a measure to study the effectiveness of these changes.

Fine–Grain Parallelism On Sequential Processors

by

Sridhar V. Kotikalapoodi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 7, 1994

Commencement June, 1995

APPROVED:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented: September 7, 1994

Typed by Kotikalapoodi V. Sridhar for : Kotikalapoodi V. Sridhar

ACKNOWLEDGMENTS

This thesis is dedicated to my parents and my sister Raju, but for whose understanding and support it would not have been possible. I am also very grateful to my brother–in–law for his unconditional help.

Special thanks to my major professor, Dr. Lee, for his inspiring guidance and for constantly challenging me to think in new directions. I owe most of the knowledge that I acquired in this field to his unconditional support and advice. Heart filled thanks to Dr. Lu for being available always for discussions. Also, special thanks to my committee members: Dr. Arthur and Dr. Pattee for making time for this defense amidst their tight schedules.

I extend my gratitude to Sridhar Jasti and Greg for their help in installing the tools for simulations.

Many thanks to all my very special friends – Ravi (Bulky), Satish, Patta, Sudhi, Praveen, Ashu, Bundy, Sameer, Dutta, Manoj R., Jasleen, Rajeev, Sanjeev and Sudha for making my stay at Oregon pleasant and unforgettable.

I would also like to thank the TAM group at UC, Berkeley for providing the TL0 to SPARC translator. Special thanks go to Klaus Schauser and Seth Goldstein for many fruitful interactions. I am also grateful to Computation Structures Group at MIT, including Andy Shaw and Boon Ang.

Table of Contents

1. INTRODUCTION **1**

1. 1. Motivation 2
1. 2. Thesis Organization 5

2. WHY MULTITHREADING **6**

2. 1. Dataflow Architectures 6
2. 2. Shortcomings of dataflow model 8
2. 3. ETS Model 9
2. 4. Hybrid Architectures and Multithreading 11

2. 4. 1. Hybrid Architectures 12
2. 4. 2. P–Risc 12
2. 4. 3. TAM 13

3. THREADED ABSTRACT MACHINE (TAM) **15**

3. 1. TAM Program Structure 15
3. 2. Storage Model 19
3. 3. Execution Model 20
3. 4. Compiling to TAM 22

3. 4. 1. A simple program in TL0 23

4. IMPLEMENTATION OF TAM **27**

4. 1. TL0 on the CM–5 multiprocessor 28

4. 1. 1. Storage model 28
4. 1. 2. Arithmetic and logic instructions 29
4. 1. 3. Sending messages 29
4. 1. 4. Receiving messages 31
4. 1. 5. Thread scheduling 33
4. 1. 6. Frame scheduling 33
4. 1. 7. Heap access 36

4. 2. Measurements 37

4. 2. 1. Control Overhead 38

5. DESIGN FOR EFFICIENT THREAD SCHEDULING **42**

5. 1. The Proposed Method 42

5. 2. Implementation for the SPARC 43

6. MESSAGE HANDLING 47

6. 1. Design using a Coprocessor 48
6. 2. Coprocessor versus Polling for inlets 53
6. 3. Performance 55

7. HARDWARE MODIFICATIONS AND CONCLUSION 60

7. 1. Hardware modifications required to implement the proposed changes 60

7. 1. 1. SPARC processor pipeline 60
7. 1. 2. Internal Instructions 61
7. 1. 3. Pipeline stages for `cdbp` instruction 62
7. 1. 4. Hardware changes needed to implement the `cdbp` instruction 64
7. 1. 5. Hardware required to implement the `ldi` and std instructions 66
7. 1. 6. Implementation of the Double Ended Queue for the LCV 66
7. 1. 7. Hardware to assert the LOCK signal 66

7. 2. Future work and conclusion 66

BIBLIOGRAPHY 70

List of Figures

Figure 2. 1. Organization of a dynamic dataflow machine 8

Figure 2. 2. Illustration of Code–block invocation, Instruction and Frame memory in Monsoon. 11

Figure 3. 1. TAM activation tree and embedded scheduling queue. 16

Figure 3. 2. State transitions taken by frames due to POST and SWAP instructions. 22

Figure 4. 1. Control transfer using FORK. 39

Figure 6. 1. Inlet Processor interface with the system. 49

Figure 6. 2. Concurrent POST and FORK instructions resulting in the erroneous execution. 51

Figure 6. 3. LCV implementation 52

Figure 6. 4. Cycles per TL0 instruction comparison of the CM5, J–machine and the CM5 using modified SPARC chip and inlet processor. 59

Figure 7. 1. SPARC processor's four stage pipeline. 61

Figure 7. 2. Execution of the store instruction. 62

Figure 7. 3. Execution of cdbp instruction when the count is not zero. 63

Figure 7. 4. Execution of cdbp instruction when the count is zero. 65

Figure 7. 5. Decoding the register file for the cdbp instruction. ... 65

Figure 7. 6. Circuit to assert the LOCK signal. 67

List of Tables

Table 3. 1. A brief description of some of the most important TAM concepts 18

Table 4. 1. Access cost to each level of the local storage hierarchy on a SPARC node. 30

Table 4. 2. Reserved special purpose registers. 30

Table 4. 3. Mapping of TL0 arithmetic and logic instructions to the SPARC. 31

Table 4. 4. Cost for sending a message limited to three 32–bit arguments. 32

Table 4. 5. Cost of polling the network and of running an inlet. 32

Table 4. 6. Mapping of TAM POST instructions on to SPARC 34

Table 4. 7. Cost of TL0 frame synchronization and scheduling operations. 35

Table 4. 8. Cost of accessing global data structures with synchronization on a per–element basis. 37

Table 4. 9. Dynamic instruction mix statistics for the benchmark programs. 39

Table 4. 10. Mapping of TAM thread scheduling instructions on to SPARC. 40

Table 4. 11. Cost and frequencies of TL0 thread synchronization and scheduling instructions. 41

Table 5. 1. Scheduling and Synchronization costs for the modified SPARC. 46

Table 6. 1. Dynamic thread and inlet characteristics under TAM. 56

Table 6. 2. Work load distribution between the two processors. 56

Table 6. 3. Overhead cost due to the addition of a coprocessor 57

Table 6. 4. Comparison of a Single processor system with a Inlet coprocessor system. 58

Fine–Grain Parallelism On Sequential Processors

1. INTRODUCTION

In the early days of computing, the issue of constructing a multiple–processor computing system was viewed as little more than an interesting intellectual exercise; after all, it seemed clear that machines could be made to operate faster simply by increasing the speed of the underlying technology. Given this view, it seemed that the style of machine organization for potential multiprocessing was not of overriding importance. The von Neumann organization, because of its sequential nature, was conceptually simple and easy to realize. Hence, it is not surprising that an entire academic community and industry was born with a built–in bias towards sequential computing. While understandable, this assumption about machine organization has inherent limits which, from our present vantage point, sit just beyond the horizon.

Attention has been focused in the recent past on constructing multiprocessor systems – attention derived from a desire for more performance. There seems to be little debate that the general purpose supercomputers of the future will be massively parallel architectures consisting of a number of nodes interconnected a high speed interconnection network [15]. What has not been done sufficiently is a reevaluation of the underlying assumptions. This is painfully clear when one observes that von Neumann style of processing still forms the building block for the majority of multiprocessor projects or proposals. Many variations on the von Neumann theme have been explored (e.g., pipelining, multiple functional units, vector instructions), but not much has been done with the sequential control required for instruction execution.

1. 1. Motivation

Using a von Neumann style of processing within the node of a multiprocessor system has its performance limit imposed by the constraints of the control–flow execution model [2]. One serious problem with distributing work over several von Neumann processors is the implied shared memory. A single processor can mask the time to fetch an item from memory with a variety of techniques such as registers, caches, etc. However, when there are multiple processors in a system, parallel tasks, which are executed on different nodes, may require simultaneous access to a shared memory cell or one task may require the result of another task. Hence, a node has to fetch the contents of the memory on a remote node. The latency of remote memory access typically grows with the machine size.

Classically, caches are used to mask memory latency, and a cache can be added to hold copies of remote locations. However, cache coherence then becomes a significant problem. Further, a processor may still idle when there is a cache miss. Although caches can be used to alleviate the remote memory access latency to some extent, it does not offer anything for the synchronizing load problem. To illustrate this problem, let us consider the case where a process running on a node requires the result from another process running on a remote node. To ensure that the remote load reads the value only after the correct value is written, some form of synchronization is required. Unlike the remote load problem, the latency here is not just an architectural property – it also depends on scheduling, and the time it takes to compute the result, which may be much longer than the transit latency.

Another problem with the von Neumann style of processing is that of programming for parallel execution. Compilers can be used to analyze and transform sequential programs into parallel ones. However, anti–dependence and aliasing detection by compilers has been achieved only on very few programs with simple structures, thus restraining parallelism in most cases.

Due to these limitations, von Neumann model of computation does not readily carry over to multiprocessors. An alternative to the von Neumann model of computation is the dataflow model of computation. The dataflow model of computation can maximally exploit parallelism in a program [11]. In addition, the functional and asynchronous characteristics of the dataflow model of computation overcome many of the problems associated with the control–flow method of exploiting parallelism. First, there is no concept of a shared storage. Instead, operands are communicated as tokens of values rather than addresses of variables. Thus, the dataflow model of computation does not produce side–effects such as the inadvertent modification of a shared variable. Second, since the data is transmitted between instructions in the form of tagged tokens, where the tag carries control information (the context) for the destination instruction, there is no overhead involved in context switching. Hence, context switching can be effectively used to mask the memory latency.

Although, dataflow model of computation offers several advantages such as rapid context switching and side–effect free execution as well as exploits maximum parallelism by executing any operation on any processor, it has some shortcomings which prevent it from being practical alternative to the traditional methods of parallel computing [8]. First, it requires a matching store to hold the state of the overall computation. Given its associative nature, it is impractical to make the matching store extremely large, so deep recursion or extensive parallelism cause the store to fill up and the program to deadlock. Second, the pipeline of the dataflow processor tends to be inefficient. Third, it does not use the high–speed registers.

In light of this discussion, there have been a number of architectures proposed to combine the instruction–level context switching capability with sequential scheduling [11]. Multithreaded architectures retain the advantages of the dataflow model, like fast context switching and cheap synchronization [4]. In addition, by allegating the task of scheduling to compiler, multithreaded architectures based on the dataflow model require little hardware modifications to support the efficient thread scheduling and synchronization. In the context

of multithreading, a thread is a sequence of statically ordered instructions where once the first instruction in the thread is executed, the remaining instructions execute without interruption. As a result, a thread defines the basic unit of work from the dataflow point–of–view that requires synchronization only at the beginning of a thread. Observations of current dataflow projects show that there is a trend towards adopting multithreading as a viable method to build hybrid architectures that combine features of dataflow and control–flow execution models [11].

One such project at UC Berkeley, called Threaded Abstract Machine (TAM), supports fine–grain interleaving of multiple threads by an appropriate compilation strategy rather than through elaborate hardware [7]. Experiments on TAM have already shown that it is possible to implement the dataflow execution model on conventional architectures and obtain reasonable performance [7]. These studies also show a basic mismatch between the requirements for fine–grain parallelism and the underlying architecture and considerable improvement is possible through hardware support.

Based on the aforementioned discussions, this work presents two design modifications required to efficiently support fine–grain parallelism on a conventional RISC architecture. First, a modification to the instruction set architecture is proposed to reduce the cost involved in scheduling and synchronization. The hardware modifications are kept to a minimum so as not to disturb the functionality of a conventional RISC processor. Second, a separate coprocessor is utilized to handle messages. Atomicity and message handling are handled efficiently, without compromising per–processor performance and system integrity. Clock cycles per TAM instruction is used as a measure to study the effectiveness of these changes. Although the discussion is based on the SPARC architecture, the design issues apply to other RISC processors as well.

1. 2. Thesis Organization

Chapter 2 starts with an overview of dataflow architectures. It further discusses the advantages and disadvantages of the dataflow model of execution. Then an overview of hybrid architectures are provided and these discussions unfold the architectural features required of modern parallel machines.

Chapter 3 describes the Threaded abstract machine (TAM), developed at University of California, Berkeley. It describes the program structure, storage model, and execution model in detail. In addition, compilation of high–level parallel language to TAM is explained.

Chapter 4 explains the issues involved in mapping of TAM to traditional processor architectures. In addition, it also presents the dynamic measurements from running benchmark programs on TAM.

Chapter 5 presents our work to reduce the overhead involved in thread scheduling. It also presents the measurements obtained from the proposed change.

Chapter 6 discusses the overhead involved in message handling. It also discusses the issues involved in message handling like atomicity and coherence. It then presents our implementation for efficient message handling and then discusses the results.

Chapter 7 summarizes the hardware modifications required for the proposed changes and concludes with the brief overview of the work.

2. WHY MULTITHREADING

von Neumann style of sequential execution is not very efficient for scalable multiprocessing because of its inability to tolerate increased latencies and to handle greater synchronization requirements [2]. In recent years research has been focused on alternative architectures for scalable multiprocessing. In this chapter, a general description of these architectures is provided. The advantages and shortcomings of these machines are also discussed.

2. 1. Dataflow Architectures

Dataflow machines can directly execute dataflow graphs [11]. Dataflow graph is a directed graph in which vertices or nodes correspond to instructions and the data dependencies which exist between these instructions are represented by edges connecting these nodes. The data values are carried by *tokens*. These tokens travel along the arcs to the destination instructions. To distinguish between the different instances of a node, a *tag* is associated with each token that identifies the context in which a particular token was generated. Thus, typically a tagged–token has processor address, codeblock name, initiation number to identify the instance of the node and the address of the instruction with in the code block appended to the data value. A node can be executed or said to be fired when all its input arcs contain a set of tokens with identical tags.

Dataflow models can be in turn classified as either *static* or *dynamic*. The *static dataflow model* exploits only a limited amount of parallelism and lacks the general support for programming constructs essential for any modern programming environment (e.g., procedure calls and recursion) [11]. On the other hand, *dynamic dataflow model* exploits all the parallelism in the program. For example, a loop can be unfolded dynamically, thus allowing

the execution of the multiple instances of the loop concurrently. For this reason, current dataflow research efforts indicate a trend towards the dynamic dataflow model.

The general organization of the dynamic dataflow machine is shown in Figure 2.1 [11]. Tokens are received by the *Matching Unit*, which is memory containing a pool of tokens waiting for their partners. The basic operation of the Matching Unit is to bring together the tokens with identical tags. If a match exists, the corresponding token is extracted from the Matching Unit and the matched token set is passed on to the Fetch Unit. If no match is found, the token is stored in the Matching Unit to await a partner. In the Fetch Unit, the tags of the token pair uniquely identify an instruction to be fetched from the Program Memory. A typical instruction consists of an operational code, a literal/constant field, and destination fields. The fetched instruction together with the token pair is sent to the Processing Unit. The Processing Unit executes the enabled instructions and produces result tokens to be sent to the Matching Unit via Token Queue.

The simplicity of this model derives from the implicit allocation of storage and scheduling associated with each message arrival. Any operation can execute on any processor, simply by sending the tokens to that processor.

The dataflow model of execution offers many attractive properties for parallel processing. First, the dataflow model of execution is asynchronous, i.e., the execution of an instruction is based on the availability of its operands. Thus, it exposes all the parallelism in the program. Second, data is appended with the address of the destination instruction. Therefore, the synchronization of parallel activities is implicit in the dataflow model. Moreover, since the data carries with it the control information (the context) with it, context switching is fast. This allows efficient exploitation of fine–grain parallelism at instruction level.

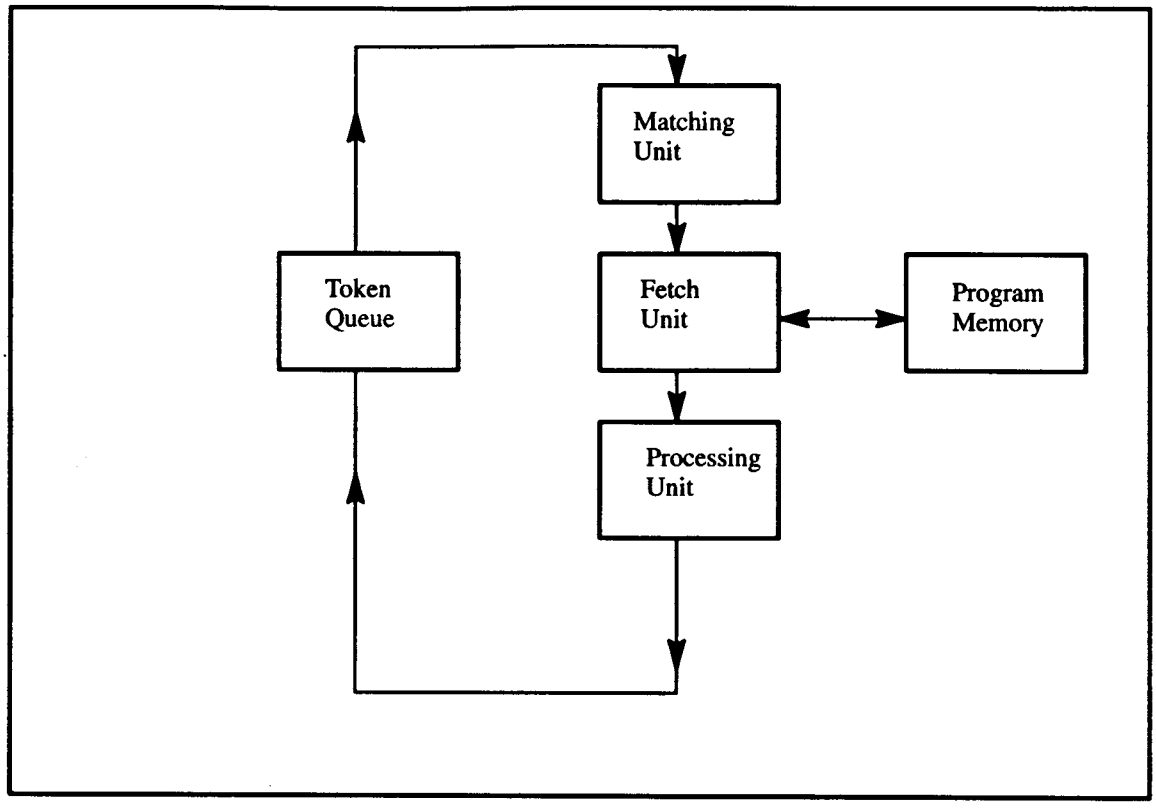


Figure 2. 1. Organization of a dynamic dataflow machine

Although the dataflow model offers several advantages like rapid context switching, cheap synchronization, and exposes all parallelism in a program, it has several shortcomings which prevent it from being practical.

2. 2. Shortcomings of dataflow model

Several shortcoming have been identified with the dataflow model of computing. These shortcomings include:

- Overhead involved in matching tokens is heavy;
- Inefficient resource allocation; and
- The dataflow instruction cycle is inefficient.

Detection of matching tokens belonging to the same instance of an instruction is one of the most important aspects of the dataflow computation model. A failure to find a match implicitly allocates memory within the matching hardware. Thus, when a code–block is mapped to a processor, an unspecified commitment is placed on the matching unit of the processor. Given its associative nature, it is impractical to make the matching store extremely large, so deep recursion or extensive parallelism cause the store fill up and the program to deadlock [5].

Another subtle problem with the dataflow model is, it does not not utilize the high speed registers for computation. So there is no means by which the compiler can organize the program to make efficient use of processor resources [8].

A more general criticism leveled at dataflow model is the inefficiency in its instruction cycle. For example, matching tokens is more complex than simply incrementing a program counter. Similarly, generating and communicating result tokens impose inordinate amount of overhead compared to simply writing the result to a memory location or a register. These inefficiencies in the pure dataflow model tend to degrade performance for programs having a low–degree of parallelism.

2. 3. ETS Model

In the past few years, there have been several proposals which try to overcome the shortcomings of a pure dataflow model. For example, The ETS model embodied in Monsoon is a partial remedy to these problems [16]. Implicit allocation of the matching store is eliminated by explicitly allocating a storage, called an *activation frame*, to hold the local storage for each function invocation. Synchronization bits are associated with each frame location to support a dyadic match.

To illustrate the operations of direct matching in more detail, consider the token matching scheme used in Monsoon. An example of the ETS code–block invocation and its

corresponding Instruction and Frame Memory is shown in Figure 2.2. In a direct matching scheme, any computation is completely described by a pointer to an instruction (IP) and a pointer to an activation frame (FP). The pair of pointers, <FP.IP>, is called a *continuation* and corresponds to tag part of a token. When a token arrives at an actor (e.g., ADD), the IP part of the continuation points to the instruction that contains an offset r as well as displacement(s) for the destination instruction(s). The actual matching is achieved by checking the slot in the Frame Memory pointed to by FP+r. If the slot is empty, the value of the token is written in the slot and its presence bit is set to indicate that the slot is full. If the slot is full, the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The result token(s) generated from the operation is communicated to the destination instruction(s) by updating the IP according the displacement(s) encoded in the instruction (e.g., execution of the ADD operation produces two result tokens <FP.IP+1, 3.55> and <FP.IP+2, $3.55>_L$).

Thus, in the ETS model, the implicit allocation of the Associative Matching Store is eliminated. However, the token buffer remains. When a message arrives, storage is allocated for it in a large queue. Although Monsoon allows short instruction *threads* to be scheduled, using frame slots and a small set of registers to convey the data between instructions in a thread, the arbitrary interleaving of tokens in the queue is unlikely to provide any useful locality. Furthermore, the machine cycle time is limited by the read–modify–write on the frame synchronization bits and the frame access per instruction.

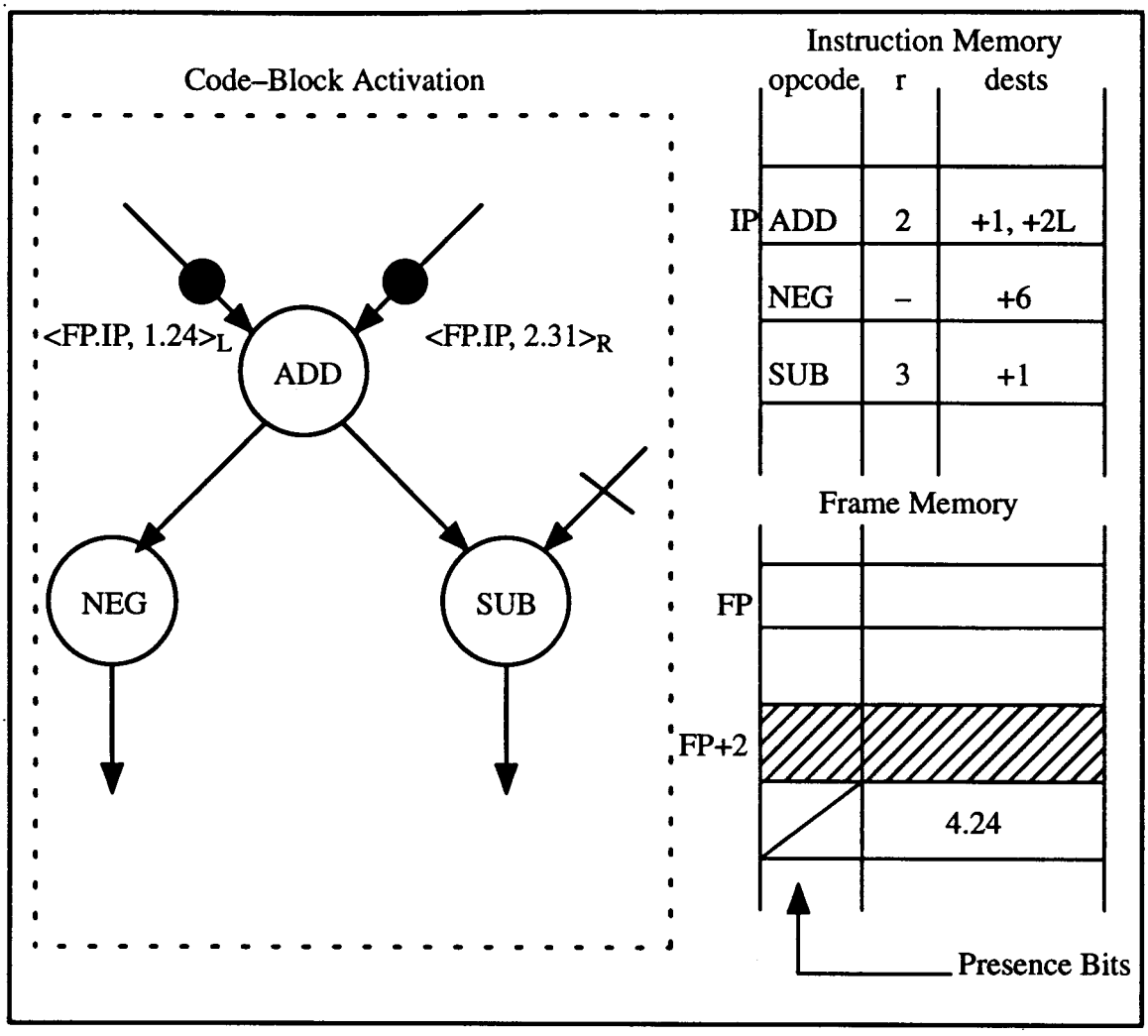


Figure 2. 2. Illustration of Code–block invocation, Instruction and Frame memory in Monsoon

2. 4. Hybrid Architectures and Multithreading

As mentioned in section 2.1, one major advantage of architectures based on the data-flow model of computation is the instruction–level context–switching capability. That is, context–switching can occur on a per–instruction basis since each datum carries context identity with it (i.e., in the form of a tag). This provides the ability to tolerate long and unpredictable memory latencies. On the other hand, the processor resources like registers are not used in the dataflow model. The instruction–level context–switching capability combined

with sequential scheduling, which provide a different perspective on dataflow architectures – *multithreading*, combines the advantages of the both. In the context of multithreading, a thread is a sequence of statically ordered instructions where once the first instruction in the thread is executed, the remaining instructions execute without suspending. As a result, a thread defines the basic unit of work from the dataflow model point–of–view that requires synchronization only at the beginning of a thread.

2. 4. 1. Hybrid Architectures

The Hybrid proposal was the first to observe the interplay between register allocation and thread scheduling [10]. This model provides a machine language of multiple threads operating against an activation frame and registers. However, each frame slot includes presence bits, like the ETS. A thread is suspended upon access to a frame slot marked not–present. The suspension is accomplished by building a queue of instruction pointers into waiting threads. The queue is rooted in the frame slot, so the eventual store of the value enables the threads. Since no registers are saved upon suspension, the compiler is required to evacuate the registers across any potential point of suspension. This elaborate suspension, queuing and scheduling mechanism is part of the basic model and required in any machine that implements it. Moreover, scheduling is outside the programming model, so there is no means by which the compiler can organize the program to make efficient use of processor resources.

2. 4. 2. P–Risc

P–Risc observed that presence–bits can be kept in the frame like local data, rather than as special tags, and that matching could be simulated by toggling the tag bit atomically and suspending on the result [14]. This means that for synchronizing threads, rather than checking the presence bits a count, which is stored in the frame memory like data, is checked to see whether the thread is ready for execution. This eliminates the need for including pres-

ence bits with each slot and results in efficient machine cycle. In addition, P–Risc eliminated the notion of suspension with in a thread. Thus synchronization is required only at the starting of a thread. However, P–Risc failed to retain the distinction between registers and frames of the Hybrid model. Instead the entire frame is viewed as a set of registers. Like the hybrid model, scheduling is outside the execution model. Thus when a thread completes, any enabled thread could execute next, so there is no means by which the compiler can develop a higher level strategy for utilizing processor resources while tolerating latency.

2. 4. 3. TAM

Dataflow research has focused on the obvious costs: scheduling and synchronizing threads. However as seen in the above sections, optimizing scheduling costs while ignoring the effects on the storage hierarchy leads to unrealistic solutions. Instead, the Threaded Abstract Machine (TAM) exposes the scheduling of threads so that the compiler can coordinate scheduling with the usual management of the storage hierarchy [7]. To aid in this coupling, TAM allows groups of related threads to be scheduled together. This reduces the cost of scheduling and permits the compiler to manage storage resources, e.g., registers and local variables, across several threads. Finally, giving priority to related threads tends to improve cache behavior. Overall, the effect is that data can be kept at smaller and faster levels of the storage hierarchy.

Thus TAM supports asynchronous parallelism while tolerating memory latency and overcomes the shortcomings which are inherent in the dataflow model like:

- Matching Overhead: Matching is done by checking a count rather than using associative memory or presence bits. Since the count is stored in the memory like any other data, matching overhead is reduced.

- Inefficient resource allocation: Since the scheduling of threads is storage driven and managed by the compiler, related threads can be scheduled together. Hence, data can be passed in registers across several threads.
- Inefficient Instruction Cycle: The processor pipeline is uniform and does not include the inefficient operations like token matching or checking and updating presence–bits. In addition, TAM can mapped effectively onto a sequential RISC processor.

3. THREADED ABSTRACT MACHINE (TAM)

The Threaded Abstract Machine (TAM), developed at University of California, Berkeley, refines dataflow execution models to address the critical constraints that modern parallel architectures place on the compilation of general–purpose parallel programming languages [7]. TAM defines a self–scheduled machine language of parallel threads, which provides a path from dataflow–graph program representations to conventional control flow. The most important feature of TAM is the way it exposes the interaction between the handling of asynchronous message events, the scheduling of computation, and the utilization of the storage hierarchy.

Since our work is based on TAM, this chapter provides a complete description of TAM and codifies the model in terms of a machine language TL0. Issues in compilation from a high level parallel language to TL0 are discussed in general and specifically in regard to the Id90 language [13].

3. 1. TAM Program Structure

A TAM program consists of a collection of *code–blocks* where each code–block typically represents a loop–body or a function. Each code–block comprises of several *threads* and *inlets*. The *activation frame*, which is analogous to the stack frame for conventional subroutine calls, is the central storage resource. Invoking a code–block involves allocating an activation frame, depositing argument values into the frame and enabling threads for execution within the context of the frame. Initialization also consists of setting the values of *synchronization counters* stored within the frame. The caller does not suspend upon invoking a child code–block so it may have multiple concurrent children. Thus, as shown in Figure 3. 1, the dynamic call structure forms a tree, rather than a stack, represented by a tree of frames.

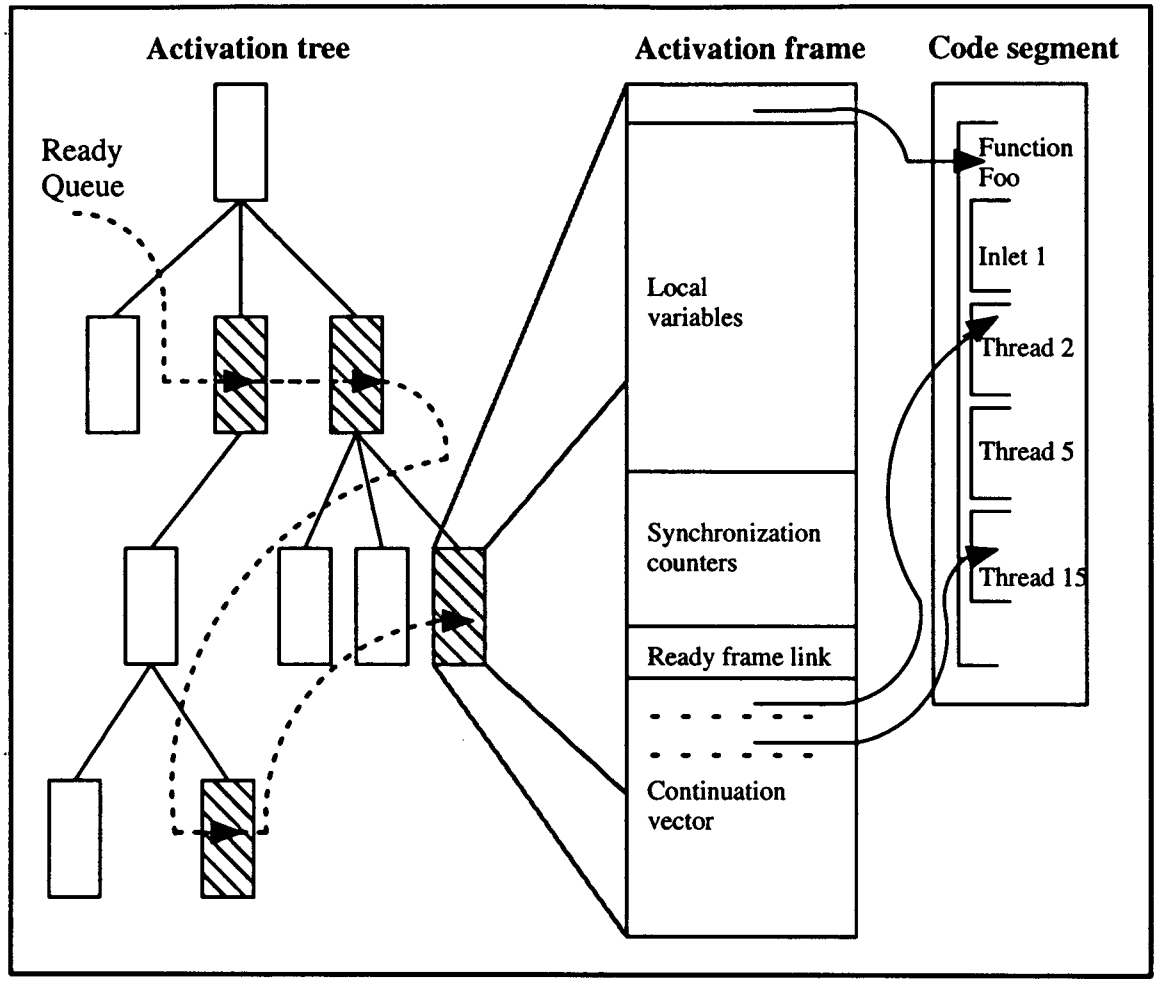


Figure 3. 1. TAM activation tree and embedded scheduling queue

An activation frame may be in one of three states: idle, ready, or running. Every processor has exactly one running (or current, or resident) frame, the frame pointed by the FP register. All local memory accesses that occur either in threads or inlets are relative to the current frame.

To support thread level scheduling, every frame has a *remote continuation vector*, the RCV, and the processor has a *local continuation vector*, the LCV. The LCV holds the enabled[1] threads for the running frame. On the other hand, the RCV contains the enabled

1. Threads are said to be enabled if they are ready for execution.

threads for a ready frame. Every code–block specifies the size of the continuation vector that must be allocated for a new frame. The high–level language compiler is responsible for determining the size by analyzing the structure of the threads in the code–block.

An idle frame is one which has no threads in its RCV. A ready frame has at least one thread in its RCV and is waiting to be scheduled, or, in other words, waiting to become the running frame. There may be zero or more idle or ready frames per processor.

The ready frame queue is maintained using the frame slots of the ready frames. The head of the queue is kept in the QP register. Every other frame in the queue points to the next ready frame. Thus, all the data structures needed for both scheduling levels are maintained in the frames; once a frame is allocated all its scheduling resources have been allocated as well.

A thread is a collection of non–suspending instructions. Two instructions are in the same thread if they can be statically ordered. There are no jumps or branches within a thread, and synchronization occurs only at the beginning of a thread. `FORK` attempts to enable a thread in the current activation. If the thread is an unsynchronizing one, it pushes the pointer to the thread onto the LCV, which contains pointers to the all enabled threads. If the thread requires synchronization, the counter associated with the thread is decremented and if it's zero then the thread is pushed on the LCV; otherwise, the decremented count is stored back. A `SWITCH` instruction forks one of two threads depending on a condition. A `STOP` instruction stops the current thread and causes some other enabled thread to begin execution. This is done by poping a thread from the LCV. When there are no enabled threads or it has executed the current code–block completely, the processor executes a `swap` instruction, which transfers control to a frame from the ready frame queue. A *quantum* is the set of threads executed during a single residency of the frame.

The arguments to the code–block, results from other frames and responses to the global heap accesses are received by *inlets*. Inlets, as shown in Table 3.1, are compiler gene–

Table 3. 1. A brief description of some of the most important TAM concepts

Code Structures	
Codeblock	A collection of threads and inlets that corresponds to a single function (or loop body) in the original program.
Thread	A sequence of instructions with a single entry and single exit point that can be executed without suspension.
Inlet	A sequence of instructions tailored to handle the receipt of a message for a target frame.
Data Structures	
Activation Frame	Similar to an stack frame in a sequential language, it is the unit of storage of each codeblock invocation.
Continuation Vector	The data structure used to store enabled threads.
Ready Queue	The data structure used to keep track of frame that are ready to execute, i.e., frames with enabled threads that are not running.
Synchronization Counters	Counters used to enforce synchronization between threads.
Operations	
Fork	The FORK instruction is used to enable threads for the currently running frame.
Post	The POST instruction is executed in inlets and is used to enable threads for the target frame. It will also, if necessary, enqueue the frame on the ready queue.
Swap	The SWAP instruction schedules a frame from the ready queue.

rated message handlers that copy the arguments into the frame and enable computation depending on the message. The SEND operation delivers a sequence of data values to an inlet relative to the target frame. The inlet specified in the message receives the message through a RECEIVE instruction and stores the data into the specified location and enables a thread by executing a POST instruction. Enabling a thread from inlet is slightly different from enabling one from a thread. An inlet can enable a thread for execution in a different frame where as the one enabled from a thread is with in the current frame and is closely coupled to the current processor state. Inlets may preempt threads, but they may not preempt other inlets.

3. 2. Storage Model

The TAM storage model includes four distinct regions: code storage, frame storage, registers, and heap storage. TAM code storage contains code–blocks representing the compiled form of the program. It appears identical to all processors and is accessible through fast local operations.

Frame storage is assumed to be distributed over processors, but each frame is local to some processor and only accessed from that processor. Work is distributed over processors on a frame invocation basis. Inter–frame communication is potentially interprocessor communication and is realized by sending values to inlets relative to the target frame.

A TAM processor contains data registers of various types and four special address registers: FP, the address of the current frame, IP, the address of the current thread instruction, IFP, the address of the target frame for the current inlet while it is executing, and IIP, the address of the current inlet instruction. A frame is running on a processor when it is referenced by the FP. Instructions can access registers or frame slots, relative to FP.

Heap storage contains objects that are not local to a code–block, including statically and dynamically allocated arrays. Accessing the global heap does not cause the processor

to stall, rather it is treated as a special form of message communication. A request is sent to the memory module containing the accessed location while threads continue to execute. The request specifies the frame and inlet that will handle the response. If the response returns during the issuing quantum, the inlet integrates the message into the on–going computation by depositing the value in a frame or register and enabling a thread. However, if a different frame is active when the response returns, the inlet deposits the value into the inactive frame and posts a thread in that frame without disturbing the register usage of the currently active frame.

Global data structures in TAM provide synchronization on per–element basis to support I–structure and M–structure semantics [3]. If the I–structure element is empty, a read is deferred until the corresponding write takes place. A remote I–structure operation generates a request for a particular heap location and the corresponding response is received by an inlet. Meanwhile, the processor continues with other enabled threads. In TAM, these split–phase transactions are supported by instructions, such as `IFETCH` and `ISTORE`, which are used to read and write to the data structures, respectively.

3. 3. Execution Model

The processor executes instructions within the current thread sequentially until a `STOP` is executed. At that point a thread address is removed from the LCV and loaded into IP, initiating the next thread. When no threads remain in the LCV, `STOP` transfers control to a *leave–thread* specified in the frame. The leave–thread typically loads the next frame pointer into FP, loads the *enter–thread* address from that frame into the LCV and performs a `STOP`. The enter–thread typically copies the threads accumulated in the RCV to the LCV and performs a `STOP`, thereby starting the new quantum.

The TAM scheduling queue is a data structure obtained by linking together frames. The compiler defines the representation of this frame level structure by the code it places in the leave–thread. The compiler can also insert register saves in the leave thread and restores in the enter thread, if register values are carried across quanta.

In translating TAM to a conventional machine, the LCV is simply a stack. The leave–thread address is placed at the bottom of the stack. `FORK` pushes an instruction address; `STOP` pops an address and jumps to it. Code generators will typically combine the last fork in a thread with the stop, producing a simple branch instead of push–pop–jump.

Inlet execution may preempt the current thread when a message arrives. The address of the inlet is loaded into the register IIP and the frame address specified in the message is loaded into the register IFP. If the message is for the current activation (i.e., if IFP = FP) the thread registers can be used in the inlet to deliver the data into registers instead of frame slots. However, if IFP $\neq$ FP, the `POST` enables a thread in the target frame. In addition, if the frame was not ready before the `POST`, the frame has to be linked to the ready frame list.

Invoking a code–block involves first allocating a frame. The caller sends arguments to inlets in the code–block relative to the newly allocated frame. The inlets are executed upon message arrival (possibly interrupting a thread on the processor holding the frame), store the values in the frame, and post threads of the code–block for later execution. The activation thereby becomes ready, meaning that it has threads waiting to be executed, and it is linked into a pool of ready frames. Execution then continues with the interrupted thread. Eventually, the new frame is scheduled when there are no enabled threads in the running frame by executing a `SWAP` instruction. The `SWAP` instruction removes one of the ready frames from the ready list and enables it by making it the current frame and executing the enter–thread as described above. The state transitions taken by the frames is shown in Figure 3. 2.

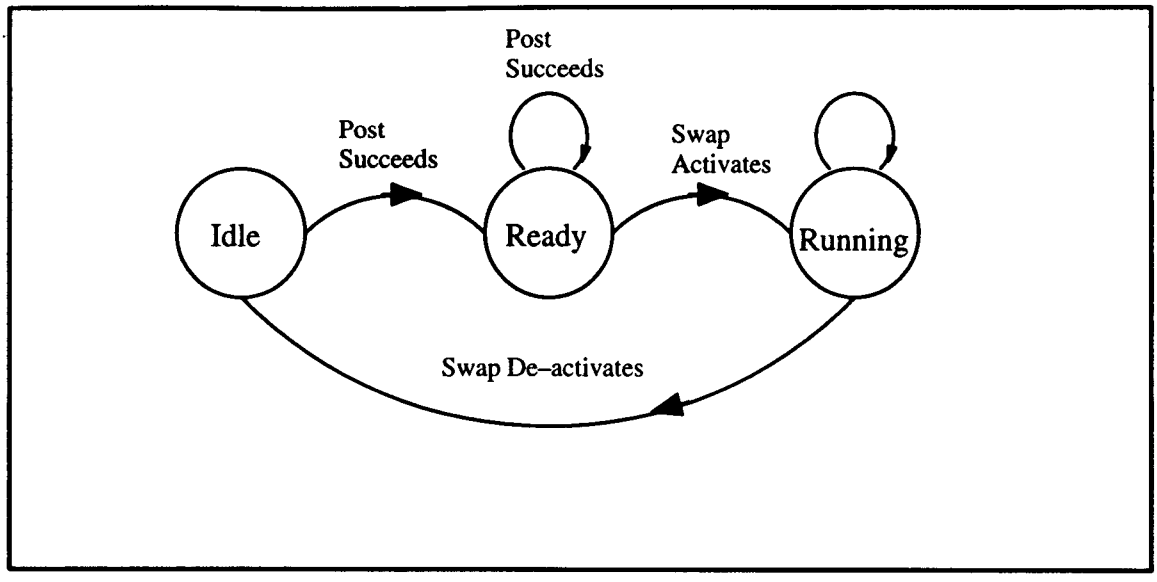


Figure 3. 2. State transitions taken by frames due to POST and SWAP instructions

Depending on its communication pattern, an invocation goes through one or more scheduling quanta. At some point it usually sends return values back to inlets of its caller. The frame is explicitly released when it is no longer required. The means of determining when frames are allocated and released depends on the high–level language; no automatic management is embedded in TAM.

3. 4. Compiling to TAM

The overall goal in compiling to TAM is to produce code that is latency tolerant, yet obtains processor efficiency and locality. TAM exposes parallelism, scheduling, and communication to the compiler and makes each cost explicit. Exposing the costs gives the compiler a clear optimization goal and allows it to map the various constructs of the parallel language to the best suited TAM primitives. On the other hand, TAM places the responsibility for correctly resolving several issues, such as management of frames, ordering of threads, and usage of local storage on the compiler. Although the source language for the compiler

is the dataflow language Id90, the TAM parallel execution model is well suited for implementing other parallel languages. This section discusses the key aspects of the compilation process from a high–level parallel language down to TAM, including the representation of parallelism, communication, synchronization, scheduling, storage management, and the use of the storage hierarchy. These issues are addressed both in general and in the context of Id90.

3.4.1. A simple program in TL0

To illustrate some of the TL0 conventions and present a concrete TL0 program, we consider the following trivial program which computes the Fibonacci numbers. The recursive calls to `fib` are the sources of parallelism. Arguments must be communicated to these parallel calls and the final result requires synchronization of the partial results.

The code for calculating a Fibonacci number recursively,

```
def  fib n = if (n < 2) then 1 else fib (n-1) + fib (n-2)
```

is compiled into the TL0 code, which is shown below.

```
CBLOCK FIB.pc
FRAME_BODY RCV=3 LCV=5                    % frame layout, RCV size is 3 threads, LCV
                                          % size is 5 threads
   islot0.i islot1.i islot2.i             %argument and two results
   pfslot1.pf pfslot2.pf                  %frame pointers of recursive calls
   sslot0.s                               %synch variable for thread 6
   retfp.pf retip.j                       %return frame pointer and inlet
REGISTER                                  %registers used
   breg0.b ireg0.i                        %boolean and integer temps
INLET 0                                   %recv parent frame ptr and return inlet
   RECEIVE retfp.pf retip.j
   FINIT                                  %initialize frame
   SET_ENTER 7.t                          %set enter–activation thread
   SET_LEAVE 8.t                          %set leave–activation thread
   SEND retfp.pf[retip.j+-1.i] <- fp.pf   %send back the address to caller
   STOP
INLET 11                                  %receive argument
   RECEIVE islot0.i
```

```
   POST 0.t "default"
   STOP
INLET 14                                  %receive frame pointer of first recursive call
   RECEIVE pfslot1.pf
   POST 3.t "default"
   STOP
INLET 15                                  %receive result of first call
   RECEIVE islot1.i
   POST 5.t "default"
   STOP
INLET 16                                  %receive frame pointer of second recursive
   RECEIVE pfslot2.pf                     %call
   POST 4.t "default"
   STOP
INLET 17                                  %receive result of second call
   RECEIVE islot2.i
   POST 5.t "default"
   STOP

THREAD 0                                  %compare argument against 2
   LT breg0.b = islot0.i 2.i
   SWITCH breg0.b 1.t 2.t
   STOP
THREAD1                                   %argument is < 2
   MOVE ireg0.i = 1.i                     %result for base case
   FORK 6.t
   STOP
THREAD 2                                  %argument >=2, allocate frames for calls
   MOVE sslot0.s = 2.s                    %initialize synchronization counter
   FALLOC 14.j = FIB.pc "default"         %invoke first child
   FALLOC 16.j = FIB.pc "default"         %invoke second child
   STOP
THREAD 3                                  %got FP of first call, send its arg
   SUB ireg0.i = islot0.i 1.i             %argument for first call
   SEND pfslot1.pf[0.i/FIB.pc] <- ireg0.i %send it
   STOP
THREAD 4                                  %got FP of second call, send its arg
   SUB ireg0.i = islot0.i 2.i             %argument for second call
   SEND pfslot2.pf[0.i/FIB.pc] <- ireg0.i %send it
   STOP
THREAD 5                                  %got results from both calls
   SYNC sslot0.s                          %synchronize
   ADD ireg0.i = islot1.i islot2.i        %add results
   FORK 6.t
   STOP
THREAD 6                                  %done
```

```
   SEND retfp.fp[retip.j] <- ireg0.i      %send result to parent
   FFREE fp.pf "default"                  %deallocate own frame
   SWAP "default"                         %swap to next activation
   STOP
THREAD 7                                  %enter–activation thread
   STOP                                   %no registers to restore
THREAD 8                                  %leave–activation thread
   SWAP "default"                         %swap to next activation
   STOP                                   %no registers to save
```

Consider the execution after the invocation of some frame *f* of the function `fib`. The first thread to be executed is Thread 0 which contains the conditional expression, with a test of the integer argument contained in the frame location `islot1` and a fork of either `Thread 1` or `Thread 2` based on the result of the comparison.

`Thread 2` generates parallelism by allocating two frames for the recursive calls. The FALLOC sends a request to a system inlet that handles frame allocation. FALLOC is a split–phase operation, because the allocation may require sending a request to another processor. The responses to the frame allocations are returned to `inlets 14` and `16`, respectively. Let's assume that the first allocation request completes before `Thread 2` finishes. Then inlet 14 is likely to interrupt Thread 2. Inlet 14 enables Thread 4 for execution. Therefore, at the end of Thread 2, *f* will continue with Thread 4, after which *f* will have no more enabled threads (unless the other allocation has already returned), so a SWAP is performed (via Thread 8) to another ready frame on the local processor (possibly the newly allocated frame). Eventually, Inlet 16 will be triggered to receive a pointer to the remotely allocated frame into the frame `pfslot1`. Inlet 16, posts Thread 3 using the default frame scheduling policy and enables the frame.

Thread 3 computes argument value in a register and sends it to Inlet 11 of the frame for the first recursive call. The argument/result linkage of a parallel call can be viewed as a very general form of split–phase operation; eventually, the result will return to Inlet 15. In the meantime, the argument message triggers Inlet 11 for the callee frame, which receives

the three values into the frame, initializes the frame with an empty RCV, sets the enter and leave threads and posts Thread 0, where our description began. Eventually the callee sends back its result.

The results from the recursive calls trigger Inlets 15 and 17, both of which post Thread 5, a synchronizing thread using `sslot0` as a counter. The second post is successful, so when *f* is run the addition is performed and the result is sent back to the caller in Thread 6. This final thread also releases the frame *f*.

The register usage policy in this example is to have the registers vacant across potential suspension points. However, the result value is carried in a register from either Thread 1 or Thread 5 to Thread 6, since no synchronization point intervenes in either case.

This simple example illustrates the interplay between representation of parallelism, communication, synchronization, scheduling, storage management and the use of the storage hierarchy.

4. IMPLEMENTATION OF TAM

This chapter describes an implementation of TAM on the CM–5 multiprocessor. TAM is codified in a pseudo machine language TL0. TL0 is a machine independent assembly language for TAM and the concrete target for the compilation from a high level parallel language. By dividing up the compilation process into two separate phases, from high level parallel language to TL0 and then from TL0 to native machine code, high level compilation issues can be isolated from the specific hardware support for threaded execution.

A TL0 program is composed of code–blocks. Codeblocks in turn consist of the activation frame layout, registers, and the code for the threads and inlets which execute relative to the frame. Each frame slot and register is statically typed. The TL0 storage hierarchy consists of an unlimited number of machine registers, frame storage and the global heap. TL0 instructions can operate directly on registers or on the activation frame. TL0 has five different instruction categories.

ALU instructions have three–address format and can operate on variables in registers and the local frame.

Network Access is provided by `SEND` and `RECEIVE` instructions. `SEND` is used in threads to send values to an inlet of another frame. `RECEIVE` instruction, which appears in an inlet, stores the message data fields into frame slots or registers.

Thread control is achieved by `FORK` and `SWITCH` instructions, and each thread is terminated by a `STOP` instruction.

Frame scheduling is expressed using `POST` and `SWAP` instructions.

Heap access is provided by `IFETCH` and `ISTORE` instructions. These instructions simply send a message to the memory controller holding the designated location. The response for a fetch is received by an inlet, but there are no explicit acknowledgments of stores.

The remainder of this chapter presents the mapping of the storage model and the implementation of these instruction categories on the CM–5 processor.

4. 1. TL0 on the CM–5 multiprocessor

The CM–5 is a massively parallel MIMD computer based on the SPARC RISC processor chip–set (including FPU, MMU and 64 KByte direct–mapped write–through cache), 8 MBytes of local DRAM memory and a network interface. The nodes are interconnected in two identical disjoint hypertrees.

4. 1. 1. Storage model

Program code is placed on every processor and the activation frames are allocated in local memory which can be kept in the cache. The heap is divided into two regions, one for small arrays which are allocated local to a node and the other for large arrays which are spread across the nodes such that logically consecutive elements are mapped onto different processors.

TL0 registers are mapped onto SPARC registers. Since TL0 does not limit the number of available registers, it is the responsibility of the code generator to spill excess TL0 registers to the activation frame. Since TL0 instructions allow frame relative addressing, operands to instructions residing in the frame must be temporarily loaded into registers. Table 4.1, summarizes the cost of accessing operands at the various levels of the storage hierarchy.

The TL0 registers are implemented on the CM–5 as a flat register file in a single register window. The single register window is divided into three categories: special–function

registers, thread registers and inlet registers. The special–purpose registers are (g0–g7), as shown in Table 4.2, hold important variables and constants used by the TL0 implementation. The TL0 IP and IIP registers are both mapped to the SPARC Program Counter (PC) register. There are sixteen thread registers (i0–i7 and 10–17) which are fully under control of the register allocator. The eight inlet registers (o0–o7) are generally reserved for inlets but may be used by the register allocator between successive network polls to hold thread temporaries.

4. 1. 2. Arithmetic and logic instructions

Most TL0 arithmetic and logic operations map into a single machine instruction. Integer divition, and multiplication are implemented by calling the appropriate library routine. Table 4.3 summarizes the costs of the basic instructions.

4. 1. 3. Sending messages

The TL0 SEND instruction can send a message of arbitrary length to an inlet of another frame. Since the CM–5 limits the message to three 32–bit words of arguments, the code generator will convert a SEND of longer messages into multiple sends.

The cost of SEND is shown in Table 4.4. The cost of a SEND is relatively high because the network interface (NI) is attached to the node MBUS and hence access to the NI requires uncached loads and stores. For this reason, sends to the local node are special–cased in software, even though the CM–5 hardware supports loop–back.

Table 4. 1. Access cost to each level of the local storage hierarchy on a SPARC node

Operand (32–bits) location	access costs	
	instructions	cycles
Register	0	0
Constant	0–2	0–2
Cache	1	2–3
DRAM	1	20

Table 4. 2. Reserved special purpose registers

Register		Function
Zero	(g0)	Hard–wired to 0
LCV	(g1)	Pointer to top of local continuation vector
Self	(g2)	Node ID
FP	(g3)	Frame pointer
Cbbase	(g4)	Pointer to origin of current code–block
Izero	(g5)	Offset to base of heap tags
NI	(g6)	Network Interface base address
Queue	(g7)	Pointer to frame scheduling queue

Table 4. 3. Mapping of TL0 arithmetic and logic instructions to the SPARC

Operation	costs	
	instructions	cycles
Integer arithmetic		
Add, sub, logical	1	1
Integer multiply	19–54	21–56
Divide	15–40	30–100
Floating–point arithmetic	1	5–7

4. 1. 4. Receiving messages

In TL0, when a message is received an inlet is invoked. The first instruction of the inlet is a RECEIVE, which specifies the frame slots where the message data is to be stored. Arrival of a message can be detected either by enabling message interrupts or by polling the network regularly. Dispatching a message interrupt into the user program incurs approximately 140 cycles of overhead and is expensive. The strategy employed in the CM–5 implementation is to explicitly poll the network once in every thread. If the thread contains an instruction which might access the network, for example SEND, then the poll is combined with that instruction. All other threads have an explicit poll inserted at the end of the thread. If a message has arrived, the appropriate inlet is called. Table 4.5 shows the cost of polling the network and the cost of running an inlet.

Table 4. 4. Cost for sending a message limited to three 32–bit arguments

Operation	costs	
	instructions	cycles
Send message to local frame		
Overhead	4	4
Push word	1	1
Send message to remote frame		
Overhead	10	25
Push word	1/2	4

Table 4. 5. Cost of polling the network and of running an inlet

Operation	costs	
	instructions	cycles
Explicit poll	3	9
Poll as part of send	2	2
Message handling		
Inlet overhead	6	13
Receive 32–bit word	$1^1/_2$	6

4. 1. 5. Thread scheduling

In TL0, thread control is realized by the FORK, SWITCH, and STOP instructions. The synchronizing are distinguished from the non synchronizing threads by including a SYNC statement placed at the beginning of synchronizing threads. Although the SYNC declaration is placed at the beginning of the thread, the synchronization test (decrement and test for zero) is performed as part of the FORK instruction. Non–synchronizing FORKs do not require the decrement and test and thus are cheaper than synchronizing ones.

Conditional control flow is implemented in TL0 through compare instructions which set a boolean variable and a SWITCH instruction which forks one of two threads depending on a boolean.

4. 1. 6. Frame scheduling

The compiler uses `default`, `local`, `remote`, `cyclic` policies for frame allocation and `default`, `fifo` and `lifo` frame scheduling policies.

FALLOC instruction allocates the frame for a new activation and passes a number of arguments to its inlet 0. The choice of processor is controlled by the policy attached to the instruction. The FFREE instruction deallocates a frame, possibly the current frame. Typically, this is followed by a SWAP which terminates the current activation.

For the SPARC, the RCV is implemented using 16–bit offsets from the code–block base. The pointer to the top of the RCV is kept in the frame, not in a register. As messages for the activation arrive, inlets are executed and enable threads into the RCV using the POST instruction. The mapping of the various POST instructions onto SPARC are shown in Table 4.6. The cost of a POST (shown in Table 4.7) is generally higher than that of a FORK and depends not only on whether the target thread is synchronizing or not, but also on the state

Table 4. 6. Mapping of TAM POST instructions on to SPARC

Operation	SPARC instructions			Cycle
POST a non sync. thread				
To a running frame	`cmp`	`fp,ifp`	;is inlet frame running	1
	`set`	`Lthr-cbbase,tmp2`	;tmp2←thread offset	1
	`be`	`isrunning`		1
	`isrunning:`			
	`sth`	`tmp2,[lcv]`	;Store the offset	3
	`sub`	`lcv,2,lcv`	;Decrement the pointer	1
To an idle frame	`cmp`	`fp,ifp`	;is inlet frame running	1
	`set`	`Lthr-cbbase,tmp2`	;tmp2←thread offset	1
	`be`	`isrunning`		2
	`ld`	`Frcv[ifp],tmp3`	;tmp3←rcv top	2
	`sth`	`tmp2,[tmp3]`	;push thread addr.	3
	`cmp`	`tmp3,ifp`	;was rcv empty?	1
	`sub`	`tmp3,2,tmp3`	;update rcv	1
	`st`	`tmp3,Frcv[ifp]`	;save back rcv	3
	`bnz`	`continue`	;nz→already enqueued	0
	`st`	`que,Fqueue[ifp]`	;store queue head ptr in frame	3
	`mov`	`ifp,que`	;make ifp head of queue	1
	`jmp`	`continue`		0†
To a ready frame	`cmp`	`fp,ifp`	;is inlet frame running	1
	`set`	`Lthr-cbbase,tmp2`	;tmp2←thread offset	1
	`be`	`isrunning`		2
	`ld`	`Frcv[ifp],tmp3`	;tmp3←rcv top	2
	`sth`	`tmp2,[tmp3]`		3
	`cmp`	`tmp3,ifp`	;was rcv empty?	1
	`sub`	`tmp3,2,tmp3`	;update rcv	1
	`st`	`tmp3,Frcv[ifp]`	;save back rcv	3
	`bnz`	`continue`	;nz→already enqueued	0†
POST to a sync. thread	`ldb`	`sync[ifp],tmp1`	;Load the count into reg. tmp1	2
	`subcc`	`tmp1,1,tmp1`	;Decrement the count	1
	`bne,a`	`continue`		1 or 2
	`stb`	`tmp1,sync[fp]`	;Store the count	3
	`code for a nonsync. thread`			

†POST is always the last instruction in the inlet and thus jump or branch would disappear in favor of a `retl` instruction from the inlet. Thus it has no cost.

Table 4. 7. Cost of TL0 frame synchronization and scheduling operations

Operation	costs	
	no. of instr. and memory accesses(load/store)	cycles
Post a thread from inlet		
Idle frame		
unsynchronizing	12(1 / 3)	18
successful sync.	15(2 / 3)	23
unsuccessful sync.	4(1 / 1)	7
Ready frame		
unsynchronizing	9(1 / 2)	14
successful sync.	12(2 / 2)	19
unsuccessful sync.	4(1 / 1)	7
Running frame		
unsynchronizing	5(0 / 1)	7
successful sync.	8(1 / 1)	12
unsuccessful sync.	4(1 / 1)	7
Swap to next frame		
first 3 threads	14	26
per extra 4 threads	6	12

of the frame. If the frame is idle (i.e., it has no enabled threads in its RCV), then it has to be enqueued onto the ready queue. In addition for both idle and ready frames, the cost of manipulating the pointer to the top of the RCV is higher than for the LCV since it is in the frame, not a register.

If the target thread is for the running frame, then instead of pushing onto RCV, the POST instruction can push the thread onto LCV. Thus, for the cost of a compare between the FP and IFP, the cost of a POST can be brought down to that of a FORK.

If there are no enabled threads, then the leave thread of the activation is executed; it is responsible for switching to the next frame.

4.1.7. Heap access

TL0 provides a special syntax for issuing remote references (e.g., IFETCH and ISTORE). Each instruction specifies the base and offset of the I–structure being accessed. The expansion first calculates the node and address of the element being accessed. Then, the expansion determines if the access is a local access and if so, performs it inline. Otherwise, a request is sent to the node that contains the element. The costs of the different cases are shown in Table 4.8.

Table 4. 8. Cost of accessing global data structures with synchronization on a per–element basis

Operation	costs	
	instructions	cycles
I–structure fetch		
Local, data present	8	11
Local, data not–present	25	58
Remote		
Initiate request	18	38
Service, data present	29	91
Service, data not–present	39	115
I–structure store		
Local, no waiting fetches	9	15
Local, waiting fetches	18	30
Remote		
Initiate request	18	38
Service	13	44
I–structure allocate (N words)	$5+4[N/8]$	$6+7[N/8]$

4. 2. Measurements

In this section measurements obtained from running benchmark programs on a single SPARC processor are presented and analyzed. These programs were written in Id and then compiled to TAM. The TAM code is then translated to SPARC. Six benchmarks ranging from 50 to 1,000 lines are used. QS is a simple quicksort using accumulation lists. GAMTEB is a MonteCarlo neutron transport code which is highly recursive with many conditionals. Paraffins enumerates the distinct isomers of paraffins. Simple is a hydrodynamics and

heat conduction code. Speech determines cepstral coefficients for speech processing. MMT is a simple matrix operation that involves creating two identity matrices, multiplying them and subtracting from a third.

Table 4.9 shows that the control overhead varies from 5% to 48% depending on the nature of the program. Message overhead varies from 0.5% to 11%. Message overhead goes up to 28%, depending on the scheduling, for the same benchmarks if these are run on a multiprocessor configuration with 64 processors [7]!.

4. 2. 1. Control Overhead

As can be seen from Table 4.9, control instructions constitute a significant part of the overhead of supporting fine–grain parallelism. The control overhead is mainly due to three instructions: `FORK`,`SWITCH` and `STOP` instructions. The TAM translator optimizes the `FORK`s and `SWITCH`es by pushing it to the end of the thread and combining it with `STOP` to form simple branches. If the `FORK` is to an immediately following thread, the branch becomes a fall through. A `FORK` or a `SWITCH` that cannot be optimized into a branch will attempt to push a thread onto the LCV before continuing with the execution of the current thread. Figure 4.1 shows the transfer of control using `FORK`. To see the relative cost of supporting TAM thread scheduling instruction on a conventional processor, Table 4.10 shows the mapping of these instructions to SPARC processor.

The cost and dynamic frequencies of `FORK`, `SWITCH` and `STOP` instructions for Gamteb and Paraffins are shown in Table 4.11. The cycle cost for each TAM instruction is obtained by adding the clock cycles for the SPARC instructions to which it is mapped. The cycle costs for each SPARC instruction is given in Table 4.10.

Table 4. 9. Dynamic instruction mix statistics for the benchmark programs

	Gamteb	QS	MMT	Simple	Speech	Paraffins
Arithmetic	6.81%	3.34%	16.82%	12%	16%	4.41%
Operand	39%	33%	74.1%	52%	66%	37%
Messages	8.01%	11%	0.45%	5.17%	0.43%	5.22%
Heap–contr	0.37%	0.6%	0.01%	0.15%	0%	1.30%
Heap–msgs	3.46%	2.7%	3.48%	5.89%	6.14%	5.34%
Control	42%	49.4%	5.12%	25%	11.43%	47%

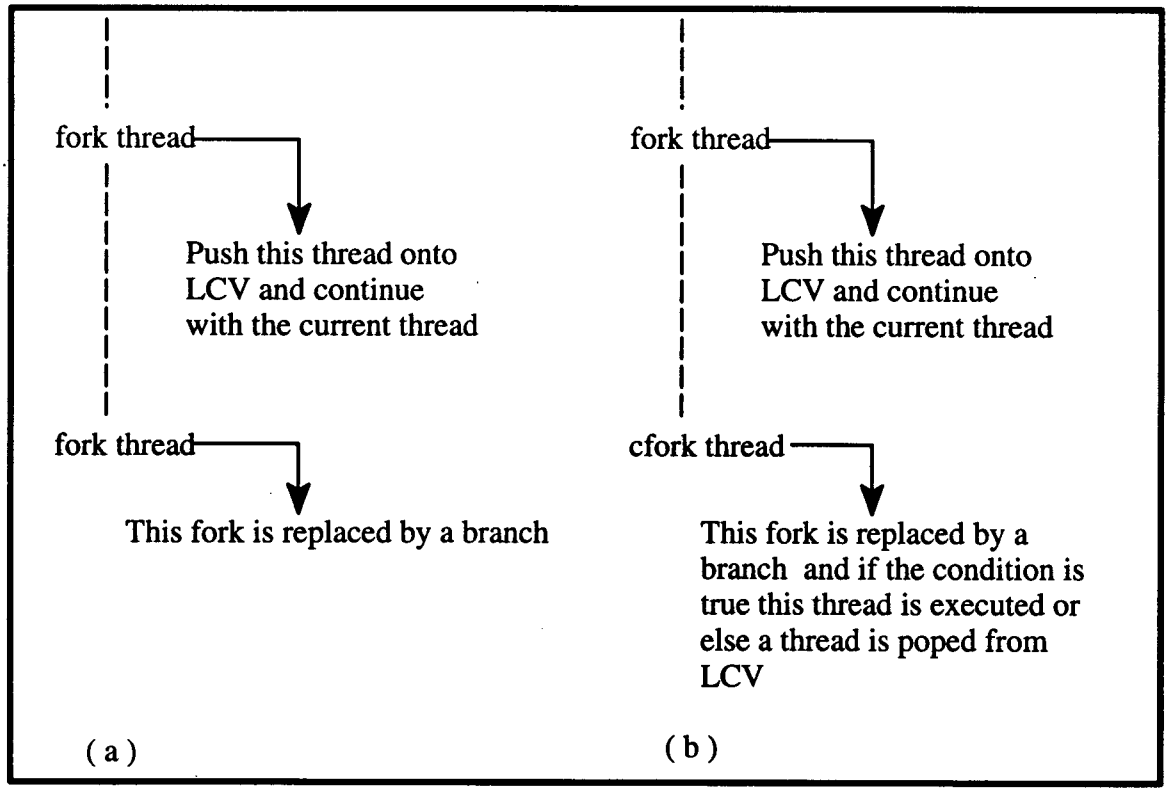


Figure 4. 1. Control transfer using FORK. If the last instruction of the thread is a FORK it is replaced by a branch

Table 4. 10. Mapping of TAM thread scheduling instructions on to SPARC

Operation	SPARC instructions			Cycle
Fork a thread				
Fall through	No additional overhead			0
Branch to thread				
unsync.	ba	thr_addr	;Branch to address	1
succ. sync	ldb	sync[fp],tmp1	;Load the count into reg. tmp1	2
	subcc	tmp1,1,tmp1	;Decrement the count	1
	be	thr_addr	;Branch if count is zero. Make annulling† and put first thread inst. here	1
unsucc. sync	ldb	sync[fp],tmp1	;Load the count into reg. tmp1	2
	subcc	tmp1,1,tmp1	;Decrement the count	1
	be	thr_addr	;Branch on zero.	2
	stb	tmp1,sync[fp]	;Store the count	3
Push thread				
unsync.	set	Lthr-cbbase‡,tmp2	;tmp2←thread offset addr.	1
	sth	tmp2,[lcv]	;Store the offset	3
	sub	lcv,2,lcv	;Decrement the pointer	1
succ. sync	ldb	sync[fp],tmp1	;Load the count	2
	subcc	tmp1,1,tmp1	;Decrement the count	1
	bnz	continue	;Test the count	2
	set	Lthr-cbbase,tmp2	;tmp2←thread offset addr.	1
	sth	tmp2,[lcv]	;Store the offset	3
	sub	lcv,2,lcv	;Decrement the pointer	1
unsucc. sync	ldb	sync[fp],tmp1	;Load the count	2
	subcc	tmp1,1,tmp1	;Decrement the count	1
	bnz	continue	;Test the count	1
	stb	tmp1,sync[fp]	;Use annulling to store the count	3
Switch	fork+2 cycles for branching depending on the condition			
Stop	lduh	[2+lcv],tmp	;Load offset	2
	add	lcv,2,lcv	;Increment the pointer	1
	jmp	[tmp+cbbase]	;Add the offset to the code–block base and go to the thread	2

†SPARC provides a special bit called annul bit for branch instructions. For conditional branch instructions if this bit is 1, then the next instruction in the sequence is executed before transfer of control.

‡`cbbase` points to the base of the code block. `Lthr` is an absolute thread address and `thr_addr` is PC relative. `Lthr –cbbase` gives the offset of the thread from the code–block base.

Table 4. 11. Cost and frequencies of TL0 thread synchronization and scheduling instructions

Type	Cycle Cost.	Paraffins (% of control instr.)	Gamteb (% of control instr.)
FORK a thread			
Fall through	0.0	3.65%	2.64%
Branch to thread			
Unsynchronizing	1.0	2.96%	2.10%
Successful sync.	4.0	6.03%	5.28%
Unsuccessful sync.	8.0	12%	11%
Push thread onto LCV			
Unsynchronizing	5.0	0.1%	0.05%
Successful sync.	10.0	2.71%	5.28%
Unsuccessful sync.	7.0	5.42%	8.11%
SWITCH a thread			
Branch to thread			
Unsynchronizing	2.0	3.96%	1.63%
Successful sync.†	5.5	0.6%	1.85%
Unsuccessful sync.	9.5	1.2%	3.70%
Push thread onto LCV			
Unsynchronizing	6.5	4.81%	2.45%
Successful sync.	12.0	0.2%	1.21%
Unsuccessful sync.	8.5	0.4%	2.42%
STOPs	5.0	20%	24%
Percentage of control instructions		64.04%	71.72%
Percentage of TL0 instructions		30.1%	30.12%
Average Cost		5.44 cycles	6.14 cycles

† The code for `switch` is

```
be    thr1          ; Branch to the address where the code for branch/push the thread is there.
                      (1 cycle if true/ 2 cycles if false)
code[thr2]          ; Code for branch/ pushing thr2
```

Thus it takes 1 extra cycle for the true case than a `fork` and two extra cycles when false. Averaging out gives 1/2 cycles. If the percentage of true cases are more, then one cycle is added to the FORK cost; if the number of false cases are more, two cycles are added.

5. DESIGN FOR EFFICIENT THREAD SCHEDULING

Based on the costs shown in Table 4.11, the main areas for improvement are synchronizing threads and STOPs which typically contribute 30%–40% of the control overhead. This is because unsynchronizing FORKs are optimized into fall–throughs and branches, thus avoiding the cost of poping a thread from the frame. On the other hand, an unsuccessful synchronizing branch requires that a STOP instruction is executed to pop a new thread from the LCV, which requires a total of 13 cycles (8 plus 5). Another source of overhead is SPARC processor's lack of post–decrement/pre–increment capability to efficiently implement pushing/poping of threads.

This chapter presents a method for decreasing the thread scheduling cost and analyzes the simulation results with the proposed change.

5. 1. The Proposed Method

In order to reduce the overhead of implementing unsuccessful synchronizing branches, we propose a method that eliminates the need to access the frame for the LCV, thereby decreasing the thread switching time. This is done by allocating a register `r_lcv` to hold the contents of the top location of the LCV. To make use of this register, a new instruction called *conditional double branch and pop*, `cdbp`, has been added. All synchronizing thread branches are now translated into the instruction

```
cdbp        thr_addr
```

where `thr_addr` points to the thread whose synchronization count is in the register `r_cnt`. The `cdbp` instuction tests the `r_cnt`; if it is zero, jumps to the thread at the location `thr_addr`; otherwise, jumps to the address given by `r_lcv`. If the control transfers to the location given by `r_lcv`, `r_lcv` register is updated by poping the next enabled thread into

it. If the control transfers to the location pointed by `r_lcv`, then the count is stored back in the delay slot.

5. 2. Implementation for the SPARC

The proposed double branch instruction can be incorporated into the existing SPARC instuction set. The general format for branch and jump instructions of SPARC is shown below.

op (31–30)	a (29)	cond (28–25)	op2 (24–21)	disp22 (21–0)

op field is zero for branch instructions and *op2* field decodes the several branches such as conditional, unconditional and floating–point conditional, etc. When the 'a' bit is 1 in a conditional branch instruction, the delay instruction is executed only if the branch is taken.

The condition *op2*=3 is not yet implemented, which can be used for the proposed double branch instruction. Thus, when the condition is true in the `cdbp` instruction, i.e. when `r_cnt` has zero, the 22–bit displacement is added to PC and the control transfers to `thr_addr` and the delay slot is not executed. If the condition is not true, the `r_lcv` is moved to PC and the delay slot is executed before transferring the control to the address given by `r_lcv`. As shown later in Chapter 7, this instruction takes 2 cycles if the count is zero otherwise it takes 3 cycles.

In addition to the `cdbp`, we need to implement two more instructions to make pushing and poping of threads faster.

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

These instructions are similar to other load and store instructions except that additional work of incrementing and decrementing `rs` is done in the execute cycle.

With these modifications, the push thread instructions and branch to thread instructions are mapped as shown below.

Branch to synchronizing thread:

```
ldb    sync[fp],tmp1          ; load the count (2 cycles)
subcc  tmp1,1,tmp1            ; decrement the count (1 cycle)
cdbp   thr_addr               ; branch to thread lthr_addr or thread given by
                                r_lcv (2 cycles if tmp1 is zero; else 3 cycles)
stb    tmp1, sync [fp]        ; use the delay slot to store back the count if it's not zero
                                (3 cycles)
```

Push unsynchronizing thread onto LCV:

```
std    r_lcv,[lcv]            ; Push the thread onto LCV (3 cycles)
set    Lthr-cbbase,r_lcv      ; Store the thread pointer in r_lcv (1 cycle)
```

Push synchronizing thread onto LCV:

```
ldb    sync[fp],tmp1          ; load the count (2 cycles)
subcc  tmp1,1,tmp1            ; decrement the count ( 1 cycle)
bnz,a  continue[2]            ; Test the count (1 cycle/2 cycles for taken/not taken)
stb    tmp1,sync[fp]          ; use annuling to store the count (3 cycles)
std    r_lcv,[lcv]            ; Push the thread onto LCV (3 cycles)
set    Lthr-cbbase,r_lcv      ; Store the thread pointer in r_lcv (1 cycle)
```

And the `STOP` is changed to:

```
mov    r_lcv,tmp1             ; Copy the contents of r_lcv to a temporary register
                                (1 Cycle)
ldi    [lcv+2],r_lcv          ; Copy top of the LCV into r_lcv (2 cycles)
jmp    [cbbase+tmp1]          ; Jump to the thread cbbase+tmp1 (2 cycles)
```

2. "a" indicates that annuling has been used. If the branch is not taken delay slot instruction is annuled.

With these changes, a branch to thread on successful synchronization now requires 5 cycles; however, unsuccessful synchronization requires only 9 cycles compared to 13 cycles (including STOP) without cdbp instruction. The std also speeds up the pushing of unsynchronizing threads onto the LCV from 5 cycles to 4 cycles. In addition, adding this instruction reduces the time to implement pushing thread on successful synchronization from 10 to 9 cycles, while that of unsuccessful synchronization remains unchanged. Table 5.1 summarizes the overall impact of these changes on the cycle cost.

From comparing Table 4.11 and Table 5.1 it can been seen that there is about 1 (19%) cycle reduction in about 30% of the total instructions and the overall execution of the program is 6% faster. This improvement can be achieved by simply changing the control unit and the datapath itself requires no major modifications.

Table 5. 1. Scheduling and Synchronization costs for the modified SPARC

Type	Cycle Cost.	Paraffins (% of control instructions)	Gamteb (% of control instructions)
FORK a thread			
Fall through	0.0	3.65%	2.64%
Branch to thread			
Unsynchronizing	1.0	2.96%	2.10%
Successful sync.	5.0	6.03%	5.28%
Unsuccessful sync.	9.0	12%	11%
Push thread onto LCV			
Unsynchronizing	4.0	0.1%	0.05%
Successful sync.	9.0	2.71%	5.28%
Unsuccessful sync.	7.0	5.42%	8.11%
SWITCH a thread			
Branch to thread			
Unsynchronizing	2.0	3.96%	1.63%
Successful sync.	6.5	0.6%	1.85%
Unsuccessful sync.	10.5	1.2%	3.70%
Push thread onto LCV			
Unsynchronizing	5.5	4.81%	2.45%
Successful sync.	11.0	0.2%	1.21%
Unsuccessful sync.	8.5	0.4%	2.42%
STOPs	5.0	6.8%	9.3%
STOPs eliminated	0.0	13.2%	14.7%
Percentage of control instructions		64.04%	71.72%
Percentage of TL0 instructions		30.1%	30.12%
Average Cost		4.5 cycles	5.19 cycles

6. MESSAGE HANDLING

As discussed in section 4.2, the overhead of handling messages in a multiprocessor environment can be as much as 30% of the total instruction execution. Therefore, in this chapter we discuss the various issues involved in handling messages efficiently. The message overhead involves the cost of passing arguments and return values for the function calls, initializing loop constants and forwarding iteration variables for parallel loops. In TAM, a message is formed and issued with a SEND instruction, which sends a number of data values to an inlet of a potentially remote frame. The message is received by a RECEIVE instruction in the destination inlet which extracts the data from the message and stores it into the frame. In TAM, computation and message handling are done by the same processor.

Sending a message is synchronous with computation, whereas receiving a message is not. Since arguments to be sent are in the registers most of the time, it is better to integrate sending of a message with computation. To decrease the cost of sending, the network interface must be effectively connected to the ALU bus.

The asynchronous message reception model can be implemented using interrupts. On a message arrival, the network interface signals an interrupt causing a trap to the kernel. The kernal forwards the interrupt to the user process by creating a stack frame for the inlet and returning to it; however, this is expensive [21]. The other alternative is to opt for synchronous implementation. Here, the messages are stored into an on–chip queue. The network is polled for a short time. If there is a message, it dispatches to the code indicated by the first word of the message at the head of the queue, i.e., the inlet. The inlet then loads the message data one word at a time into the registers. The advantage of polling is that the compiler decides when to poll for messages. Hence the atomicity is not a problem. There is a tight coupling between the computation and communication. When the message is for the currently running frame, updating the LCV or synchronizing counters becomes easy since they are in the registers. On the other hand, there is overhead involved in polling. It might

not be tolerable for coarse–grained computation. Moreover, there is a waiting time involved for incoming messages before they can be processed.

Alternative choice is to use a coprocessor for receiving messages. The responsibility of the coprocessor is to execute the inlet code. The idea of having a coprocessor to execute inlet code was proposed in the *T project by MIT [15]. However, the important issues of atomicity among the instructions such as FORK, SWITCH, SWAP and POST were assumed to exist and were not dealt properly. The following section presents a design using coprocessor for handling inlets and considers the important topics such as atomicity and coherence between the two processors.

6. 1. Design using a Coprocessor

The important issues involved in the design of the system with a coprocessor are the atomicity among the instructions such as FORK, SWITCH, SWAP and POSTs, and coherency between the main processor and the coprocessor. A simple but effective solution for coherency problem is to use a common data cache for the two processors. The block diagram for the system is shown in Figure 6.1.

The coprocessor executes the inlets. A typical Inlet has three instructions RECEIVE, POST and NEXT. The RECEIVE instruction extracts the message and stores the data into the frame. POST instruction pushes a thread into the LCV or RCV depending on the frame to which the message is sent. NEXT instruction checks whether any message is present. If a message is present, it extracts the frame pointer, inlet instruction pointer and data into the coprocessor's registers. Whenever the coprocessor needs the bus for memory load/store operations, it sends a BHOLD signal to the main processor and the main processor enters a wait state.

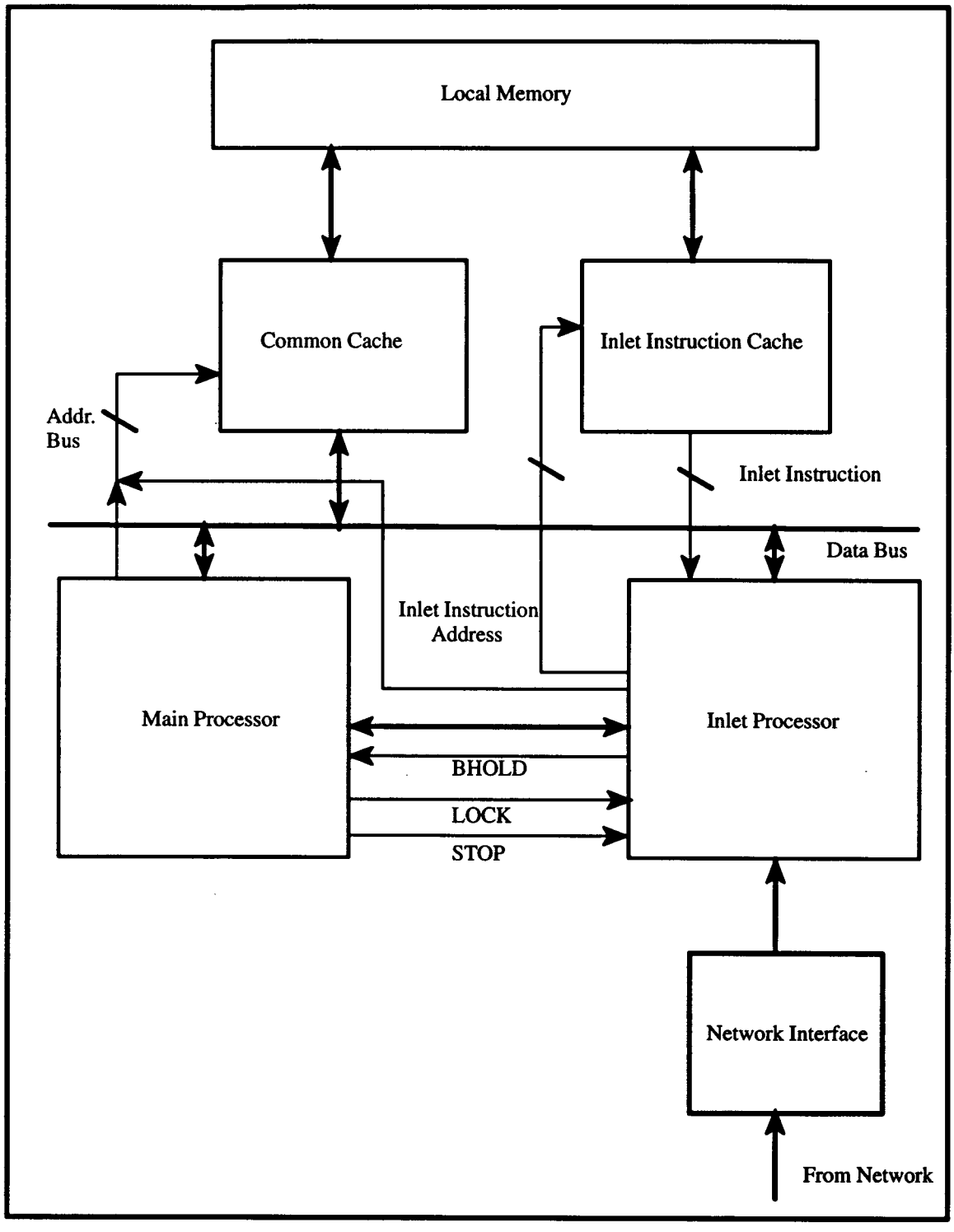


Figure 6. 1. Inlet Processor Interface with the system

The POST instruction checks the pointer to the frame to which the arriving message has to be posted with the frame pointer of the currently executing frame. If they are not equal, the thread has to be pushed onto RCV. Moreover, if the frame is idle, it is added to the ready frame queue. Since the pointer to the top of the RCV is kept in memory, the coprocessor loads and updates this and pushes the thread into the RCV. In this case, since only inlets can post to RCV and threads in the current frame never access RCV pointer, the POST operation need not be atomic with respect to FORK and SWITCH instructions.

If the POST is to the currently running frame, more care has to be taken. If the thread is a synchronizing thread, the POST instruction reads the entry count from the frame location, decrements it and checks whether the thread can be posted. If it cannot be posted, the count is stored back in the frame. A problem occurs when the FORK from the currently running thread accesses the synchronization count from the frame and the POST from the inlet accesses the same location before the updated value of the count from the FORK is stored back. Then each will decrement the count by one and write it back which is the original value decremented by one where as, we need to store the value decremented by two. To avoid this problem, these two operations need to be atomic. Atomicity in this design is achieved by asserting a LOCK signal whenever the main processor is updating a synchronizing counter. The LOCK signal is asserted until the count is stored back or it is zero. This prevents the coprocessor from accessing the memory bus. Although this avoids the problem of coprocessor accessing the count when the main processor is decrementing it, the problem still remains if the count was accessed by the coprocessor before the LOCK signal was asserted and it is stored back after the LOCK signal is deactivated. This problem can be avoided if the coprocessor checks the count before storing it back. If the decremented count in the coprocessor equals the count in the frame location, then the main processor has updated this count. So, in this case count is decremented by the coprocessor again before storing it back.

Another problem is posting a thread to the LCV. Since the LCV pointer and top of the LCV are stored in the processor registers, these values need to updated whenever a thread

is posted to the current frame. For example, consider the case where the inlet processor has just read the LCV pointer from the main processor register to update it and post a thread. Main processor now executes a pop instruction and updates the LCV pointer. Now the coprocessor writes the decremented value into the LCV register resulting in erroneous value (Figure 6.2).

In our design, this problem is avoided by implementing the LCV as a double–ended queue. Inlets, instead of pushing threads onto top of the LCV, append the thread pointer to the bottom of the LCV. Figure 6.3 shows this implementation of the LCV. Initially we have the LCV on top of the leave–thread pointer. When there is a POST from the inlet, it is appended to the bottom of the LCV and the leave–thread pointer is pushed down by a slot. The

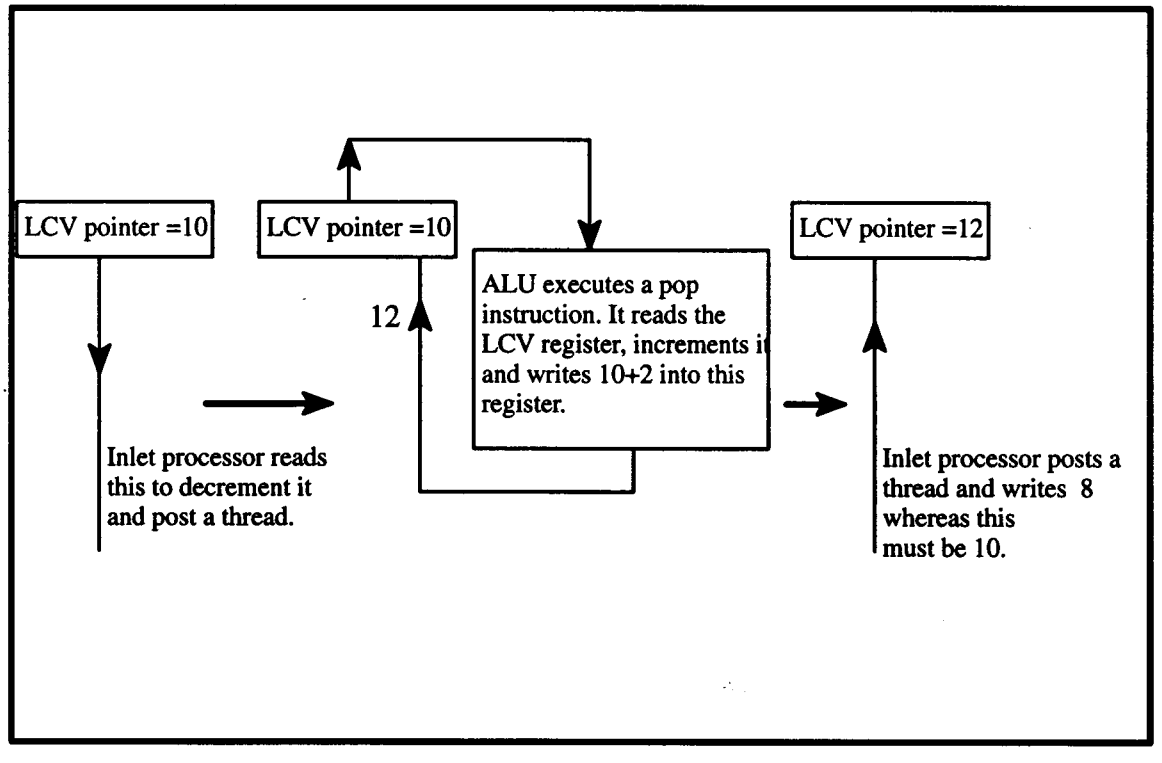


Figure 6. 2. Concurrent POST and FORK instructions resulting in the erroneous execution

following sequence of instructions are required to implement this modification.

```
; reg register points to the thread to be posted
; ltp register points to the leave–thread pointer
        swap[3] [ltp],reg  ; Swaps the thread to be posted and the leave–thread
                           pointer (4 cycles).
        addcc ltp,2,ltp    ; Pull the stack down by a slot (1 cycle).
        st    reg,[ltp]    ; Store the leave–thread pointer in the slot (3cycles).
```

With this modification, posting a thread to the LCV takes 4 more cycles since the simple store is replaced by a swap instruction to swap the leave–thread pointer in the memory and the pointer to the thread to be posted which is in the register. This takes one additional cycle compared to a simple store. Now, the leave–thread pointer, which is in the register, has to be stored back in the memory resulting in the additional 3 cycles. But, since this is executed by the coprocessor, the execution time of the thread is not affected.

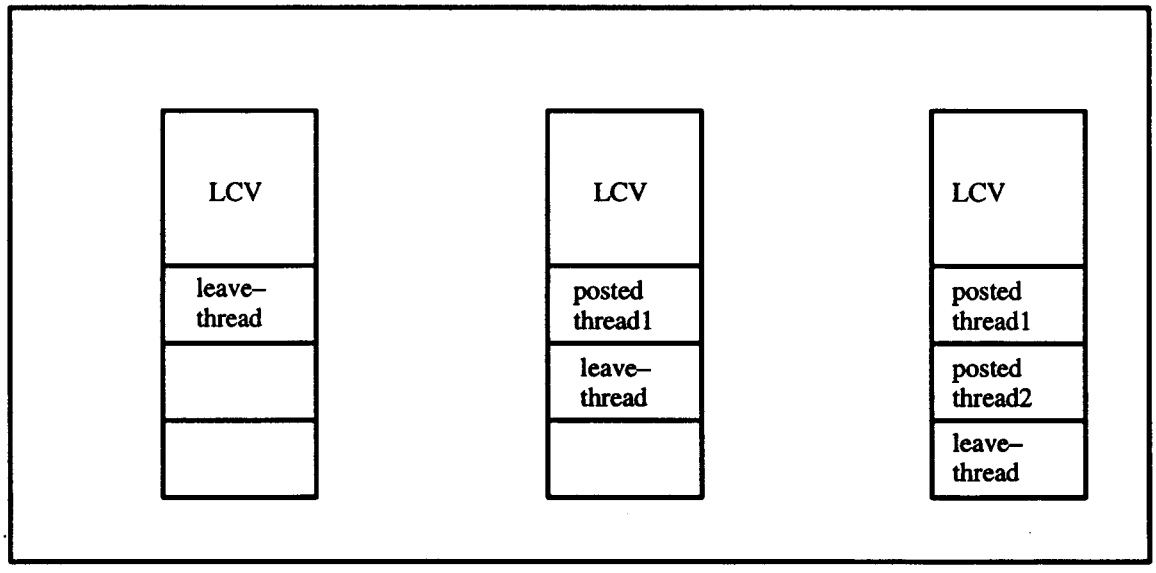


Figure 6. 3. LCV implementation

3. This swap is different from the TL0 SWAP.

The only other instruction that needs to be atomic with respect to a POST is SWAP. SWAP transfers the execution to another enabled frame. The control is transferred to a different frame when there are no ready threads in the current frame[4]. This SWAP occurs in the leave–thread. SWAP needs to be atomic with respect to POST because the coprocessor has a register to hold the current frame pointer. Every POST compares this with the frame pointer from the message. Hence, whenever the currently running frame is changed, the frame pointer register in the coprocessor has to be changed immediately. Atomicity in this case is achieved by stopping the coprocessor. Once SWAP instruction is decoded, the main processor sends a STOP signal to the co–processor. Then the co–processor stops executing the messages until it receives the new value of the frame pointer from the main processor. Moreover, the threads posted to the currently running frame before the SWAP is executed and after the leave–thread pointer is popped into `r_lcv` should be executed before executing the leave–thread. Hence, the starting code of the leave–thread should compare the `r_lcv` with the LCV top; if they are equal leave–thread is executed; otherwise, the thread at the top of the LCV is executed and the `r_lcv` is changed to point to the thread which is just below the top of the LCV.

6. 2. Coprocessor versus Polling for inlets

As discussed earlier, several factors determine whether to use a coprocessor for inlets or to use a single processor that polls for the messages. One important factor is the percentage of time which is spent on executing inlets. Using a coprocessor for inlets pays–off when the relative work load between two processors is about the same, i.e., they both execute about the same number of instructions. The remaining part of this section compares the performance of the two choices.

4. SWAP is also executed whenever the execution of the current frame is completed and this frame can be freed. But in this case there will not be any threads posted to this frame.

Table 6.1 compares the number of inlet instructions with thread instructions for the benchmark programs executed without a coprocessor. The first row shows the average TL0 instructions per thread and the third row shows the TL0 instructions per inlet. There are about 3–5 TL0 instructions per inlet. The fourth row shows the average inlet instructions per thread. This is obtained from multiplying the second and third rows. From this it can be seen that the average inlet instructions are slightly less than the thread instructions. A better way is to compare the actual times spent by the two processors. To do this, we define a new measure, the average *clock cycles per TAM* instruction (CPT), which is obtained by multiplying the instruction frequency of each instruction type with its clock cycles and summing up these products.

Table 6.2 shows the work load on the two processors. Depending on the program the work load varies from 25% to 75%. Although the work load on the inlet processor is less than the main processor, the coprocessor eliminates the overhead due to polling, which contributes about 4% overhead for the benchmark programs [21]. In addition to this, the average CPT for the main processor is reduced considerably as shown later.

Using a coprocessor for handling messages increases the overhead of various operations. The overhead involved in each instruction type is summarized in Table 6.3. Whenever the main processor is updating the entry count, it sends a LOCK signal to the inlet processor. Hence, the inlet processor remains idle for this period. This cost is reflected in the third row of the Table 6.3 for various synchronization operations. In addition, whenever the inlet processor needs the bus it sends a BHOLD signal to the main processor. This cost is reflected in the remaining columns of the third row. This cost is obtained my multiplying the number of memory accesses(shown in Table 4.6) with the cycles(1 cycle for load and 2 cycles for store). The extra cost of posting a thread to the current running frame(because of the double–ended structure of the queue) is also incuded in the Table.

6. 3. Performance

Table 6.4 compares the CPT on a single SPARC processor used for both computation and inlet handling with the CPT for the modified SPARC using a coprocessor. The analysis assumes a perfect cache. The CPT for TAM is taken from the Berkeley TAM group. Comparing rows one and two from Table 6.4, it can be seen that the main processor CPT is reduced by more than 50%. This is achieved with only a slight change in the ISA (adding instructions like cdbp etc.) of the SPARC processor and having a coprocessor for the execution of the inlets.

As seen from Table 6.3, the overhead due to the inlet processor is 1.26 cycles (1.39 cycles) for Paraffins (Gamteb). The percentage overhead is obtained by dividing this by the total cycle time (which is shown in the first row of the Table 6.4). Hence the percentage overhead is 1.26/15.5 = 8% for Paraffins (1.39/13 =11% for Gamteb). On the other hand, the total improvement due to the changed ISA (6%) and elimination of polling (4%) is 10%. Since the overall improvement is almost same as the over head, the combined CPT for the design with two processors should be approximately equal to that of the CPT with the uniprocessor case. This in turn can be verified from the Table 6.4.

Though the combined CPT is almost same for both designs, the design with the inlet processor allows the division of computation and communication between the processors resulting in smaller effective CPT as shown in Figure 6.4.

Figure 5 compares the cycles per TL0 instruction for CM5, J–machine (modified to include a floating point unit and a data cache), CM5 (modified to include an improved network interface) and the current design, i.e., CM5 using the modified SPARC and coprocessor for inlets. It can be seen from this figure that by splitting the work between the main processor and coprocessor there is a significant reduction in the CPT of the main processor.

Table 6. 1. Dynamic thread and inlet characteristics under TAM

Thread Characteristics	QS	GAM-TEB	PAR-AFFINS	Simple	Speech	MMT
Ave TL0 Insts. per Thread	2.6	3.2	3.1	5.3	6.3	17.6
Inlets per Thread	0.3	0.4	0.2	1.1	1.2	1.6
Ave TL0 Insts. per Inlet	4.0	5.1	3.0	3.4	3.0	3.0
Ave Inlet Insts. per Thread	1.2	2.04	0.6	3.74	3.6	4.8

Table 6. 2. Work load distribution between the two processors

Thread Characteristics	QS	GAM-TEB	PAR-AFFINS	Simple	Speech	MMT
Ave TL0 Insts. per Thread	2.6	3.2	3.1	5.3	6.3	17.6
Ave Inlet Insts. per Thread	1.2	2.04	0.6	3.74	3.6	4.8
CPT for Thread †	4.82	6.14	5.66	7.5	7.72	6.74
CPT for Inlet	7.57	7.12	7.46	7.67	7.66	6.01
Ave. clock cycles per Thread	12.53	19.65	17.55	39.75	48.64	118.62
Ave. Inlet clock cycles per thread	9.08	14.52	4.48	28.69	27.58	28.85
Ratio of work load on inlet proc./main processor	0.72	0.74	0.26	0.72	0.57	0.24

† The CPT for thread and inlet are obtained by taking the additional overhead due to the addition of the inlet processor. This overhead costs are shown in Table 6.3. In addition the above table includes the performance improvement due to the changed ISA of the SPARC.

Table 6. 3. Overhead cost due to the addition of a coprocessor

Type	Cycle Cost. Main Proc.	Inlet Proc.	Over-head (cycle)	Paraffins (% of to-tal instr.)	Gamteb (% of to-tal instr.)
FORK a thread					
Branch to thread					
Successful sync.	5	5	5	2.83%	2.22%
Unsuccessful sync.	9	9	9	5.64%	4.62%
Push thread onto LCV					
Successful sync.	9	4	4	1.27%	2.22%
Unsuccessful sync.	7	7	7	2.55%	3.41%
SWITCH a thread					
Branch to thread					
Successful sync.	6.5	5	5	0.28%	0.78%
Unsuccessful sync.	10.5	9	9	0.56%	1.55%
Push thread onto LCV					
Successful sync.	11	4	4	0.01%	0.51%
Unsuccessful sync.	8.5	7	7	0.02%	1%
POST a thread from inlet					
Idle frame					
Unsynchronizing	7	18	7	0.61%	0.54%
Successful sync.	8	23	8	0.12%	0%
Unsuccessful sync.	3	7	3	0.12%	0%
Ready frame					
Unsynchronizing	5	14	5	0.45%	1.57%
Successful sync.	6	19	6	0%	0.1%
Unsuccessful sync.	3	7	3	0.23%	0.19%
Running frame					
Unsynchronizing	5	11	9	0.26%	0.24%
Successful sync.	6	16	10	0%	0.01%
Unsuccessful sync.	3	12	9	0%	0.01%
RECEIVE a message	2	6	2	2.1%	2.79%
SWAP	26	26	26	0.3%	0.23%
Increase in combined CPT of the two processors				1.26 cycles	1.39 cycles

† The main processor cycle cost for posts is obtained from the number of cycles the inlet processor needs the bus. For example, for posting an unsuccessful synchronizing thread to the running frame, inlet processor needs the bus to load the entry count (address is on the bus for 1cycle) and since the thread is unsuccessful, it has to store the count(address is on the bus for 2 cycles). Hence the cost for the main processor is 3 cycles.

Table 6. 4. Comparison of a Single processor system with a Inlet coprocessor system

	QS	GAM-TEB	PAR-AFFINS	Simple	Speech	MMT
CPT for TAM	15	13	15.5	20	20.5	16.5
CPT for the main processor	4.82	6.14	5.66	7.5	7.72	6.74
CPT for the Inlet processor	7.57	7.12	7.46	7.67	7.66	6.01

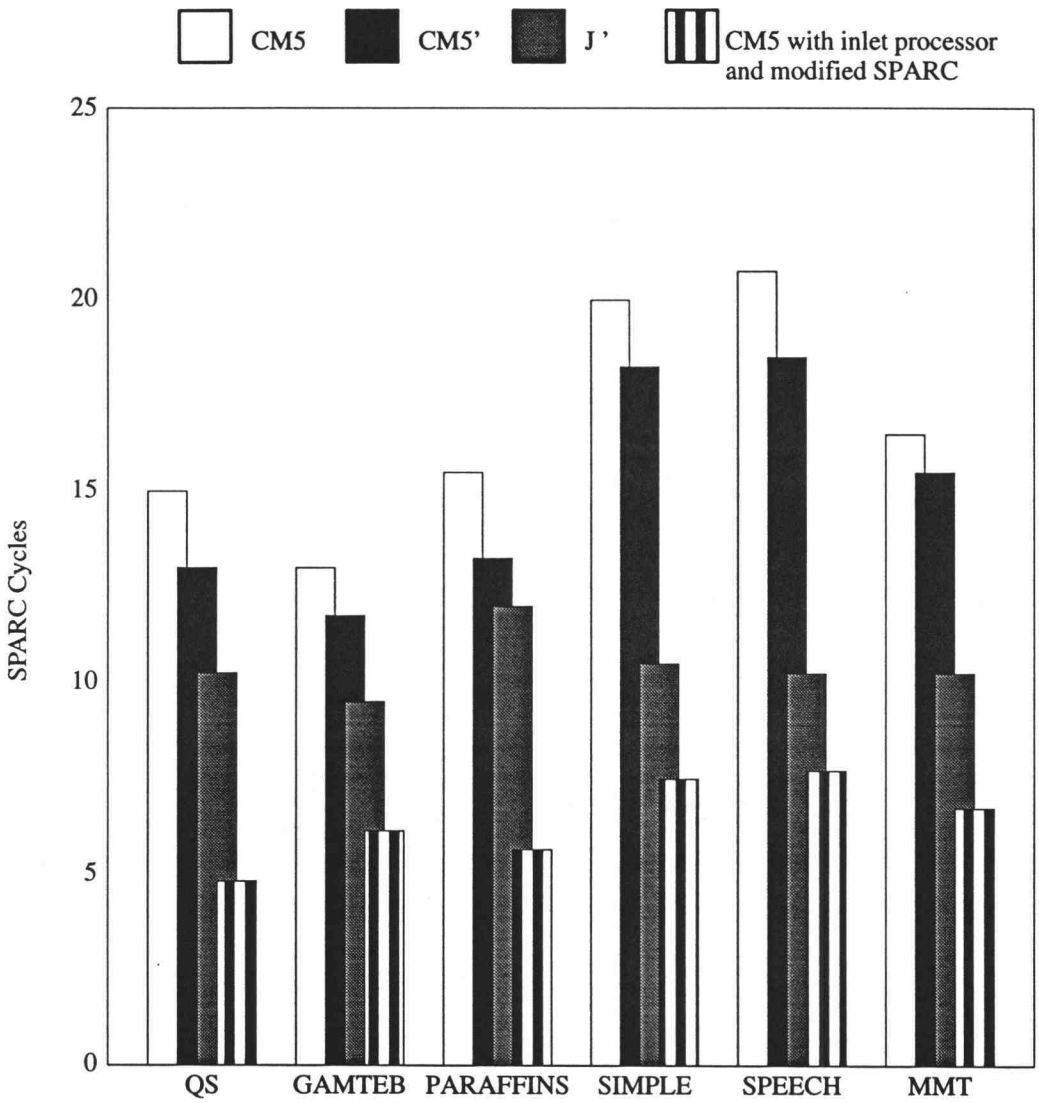


Figure 6. 4. Cycles per TL0 instruction comparison of the CM5, J–machine (modified to include floating point hardware and cache), CM5 (modified to include a better network interface) and the CM5 using modified SPARC chip and inlet processor.

7. HARDWARE MODIFICATIONS AND CONCLUSION

7. 1. Hardware modifications required to implement the proposed changes

7. 1. 1. SPARC processor pipeline

The SPARC processor has a four–stage deep pipeline [12]. Each stage of the processor pipeline performs a subset of operations that are needed to complete the execution of an instruction. A brief description of each pipeline stage follows:

1. Fetch Stage – In this stage of the pipeline, a new instruction addressed by the program counter is fetched.
2. Decode Stage – In this stage, the instruction is decoded and source operands are read from the register file. The source operands read during this stage are passed to both the Execution Unit and the Instruction Fetch Unit for execution of the instruction in later stages. The decode stage of the pipeline is also used to generate the next instruction address (and in the case of branches, the branch target address).
3. Execute Stage – In this stage, the Execute Unit performs arithmetic and logic operations on the operands read during the decode stage. The results of these operations are saved in a temporary result register before they are actually written into the destination register. For loads and stores the effective address for the operands is also calculated in this stage.
4. Write Stage – The write stage marks the end of an instruction execution in the pipeline. In this stage a decision is made whether to write the results into the register file, which means the instruction has completed successfully, or to prohibit any changes in the state of the processor. The write stage will abort if an exception is raised during the execution of that instruction. Figure 7.1. shows the working of the four–stage pipeline.

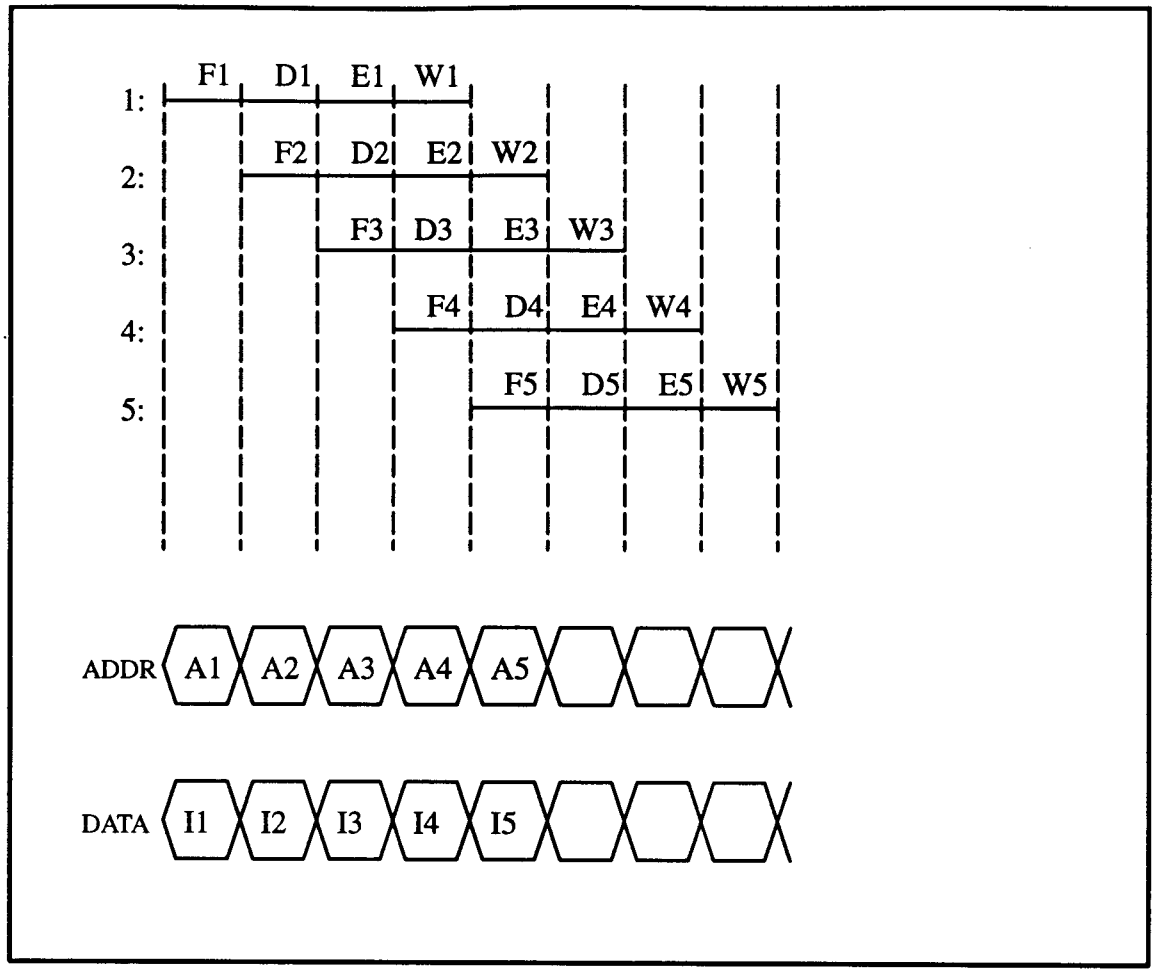
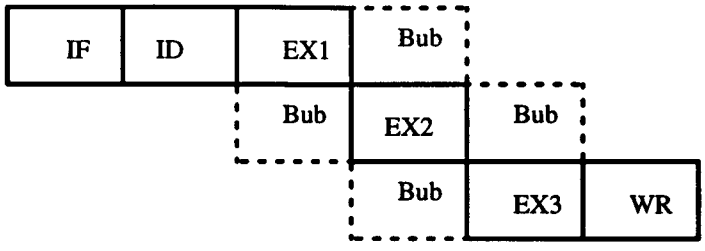


Figure 7. 1. SPARC processor's four stage pipeline

7. 1. 2. Internal Instructions

The state machine and the controls of the SPARC processor are designed so that each multiple–cycle instruction behaves like several consecutive single–cycle instructions. This is accomplished using *internal instructions* which are generated automatically by the Fetch Unit and are injected into the processor's pipeline as they are needed. Load and store instructions are examples of instructions that need more than one cycle to complete. Figure 7.2 shows a single–word store instruction which takes two extra cycles in the pipeline to complete.



EX1 Stage: Effective Address is calculated

EX2 Stage: Address is kept on the bus for checking write access

EX3 Stage: Data is kept on the bus

Remaining stages of the internal instructions are filled by bubbles as shown

Figure 7. 2. Execution of the `store` instruction

7. 1. 3. Pipeline stages for `cdbp` instruction

As seen in Chapter 5, the `cdbp thr_addr` instructions checks the count; if the count is zero control transfers to the thread pointed by `thr_addr`; else the control transfers to the thread pointed by `r_lcv` and `r_lcv` has to be updated.

The operations of a `cdbp` instruction in the various pipeline stages are as follows.

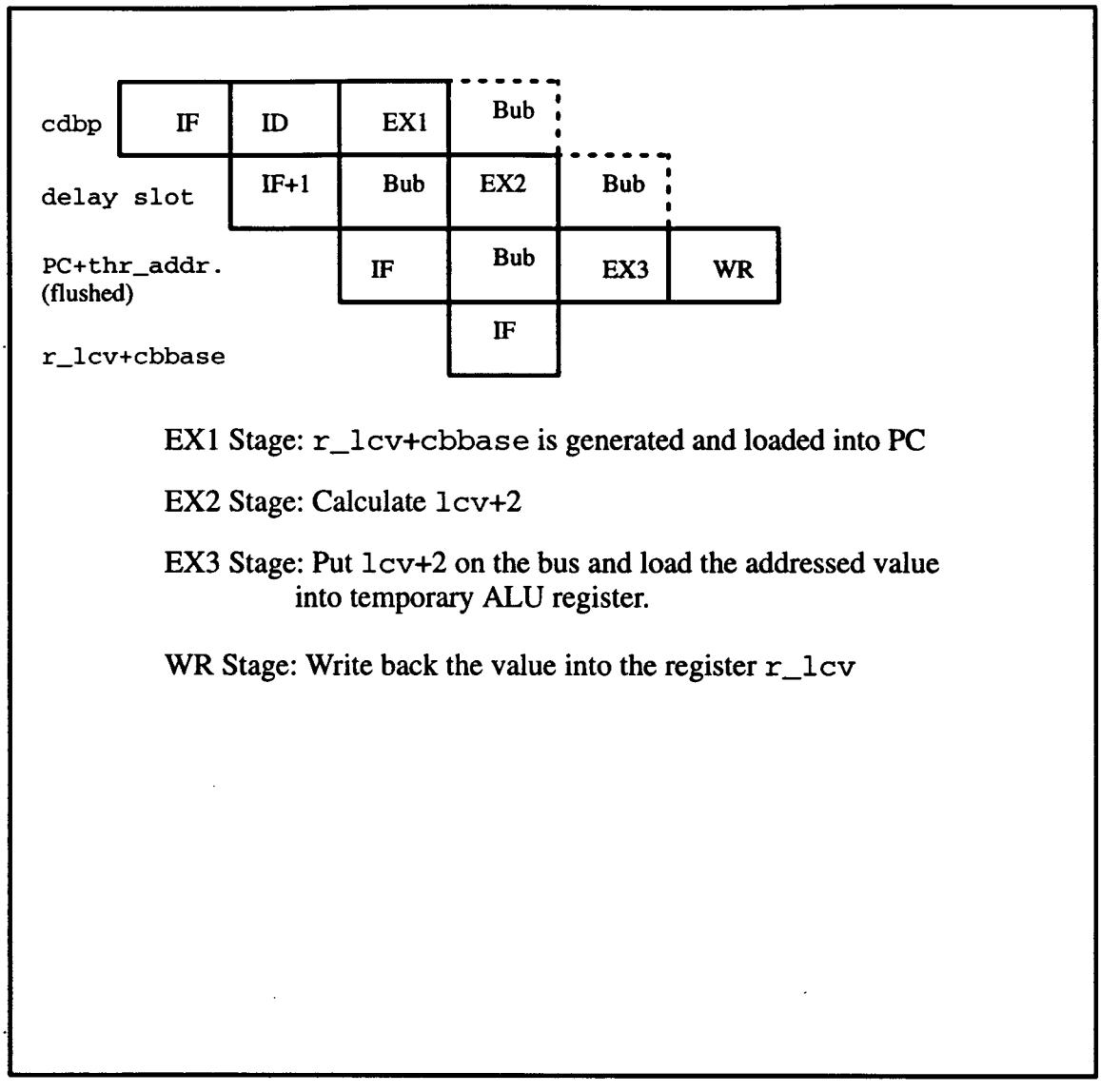


Figure 7. 3. Execution of cdbp instruction when the count is not zero

i) When the count is not zero:

a)Fetch Stage: Fetch the cdbp instruction.

b)Decode Stage: Decode the instruction and compute PC+thr_addr. Operands r_lcv and cbbase (which are implied) are read from the register file.

c)Execute Stage(EX1): By this time the condition code from the previous instruction is avail– able. Thus, if the condition is not true (r_cnt is not equal to 0), the address r_lcv+cbbase is generated and loaded into PC. If the condition is not true, nothing has

to be done in this stage and the following stages are skipped.

d)Execute Stage(EX2): Increment the stack pointer. `lcv←lcv+2`.

e)Execute Stage(EX3): Put the address `lcv+2` on the bus and load the value into the temporary register.

f)Write Back Stage(WR): Write back the value into the register `r_lcv`.

j)Fetch Stage(F2): In this stage the delay slot instruction is fetched.

k)Fetch stage(F3): Here the instruction at address `r_lcv+cbbase` is fetched.

Thus, when the count is not zero, the `cdbp` instruction, as seen from Figure 7.3, takes 3 execute cycles before the next instruction can enter the execute stage.

ii) When the count is zero:

The execution of this is shown in Figure 7.4. The operation in the pipelineis as follows.

Fetch Stage: Fetch the `cdbp` instruction.

Decode Stage(D1): Decode the instruction and compute PC+`thr_addr`. Operands `r_lcv` and `cbbase` (which are implied) are read from the register file.

Thus as seen from the Figure 7.4, `cdbp` instruction takes 2 cycles before the target instruction can enter the execute stage.

7. 1. 4. Hardware changes needed to implement the `cdbp` instruction

As seen from the above discussion, the `cdbp` instruction can be supported in the four–stage pipeline of the SPARC processor. The only change that needs to be done is to redesign the control section to give the necessary control signals when this instruction is decoded. In SPARC processor, the register file is decoded from the operand bits of the instruction. However, in the `cdbp` instruction the operands `r_lcv` and `lcv` are implied and are not specified in the instruction. Hence, the control unit has to decode the register file in this case. The circuit for this is shown in Figure 7.5. Similar circuit is required to decode cbbase.

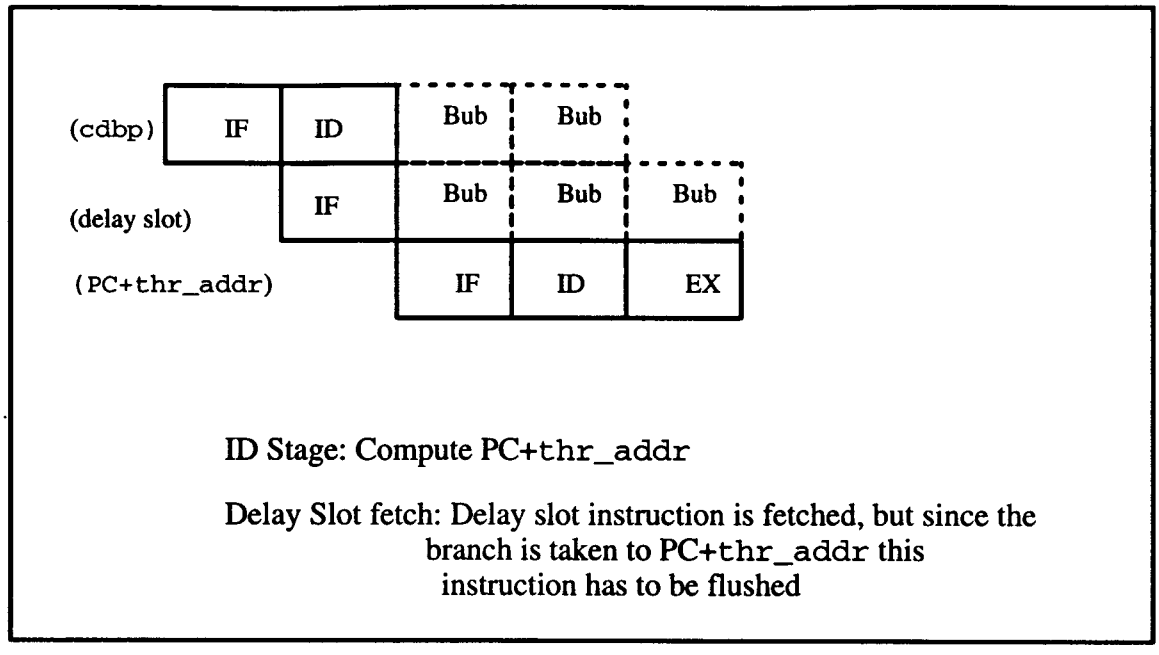


Figure 7. 4. Execution of cdbp instruction when the count is zero

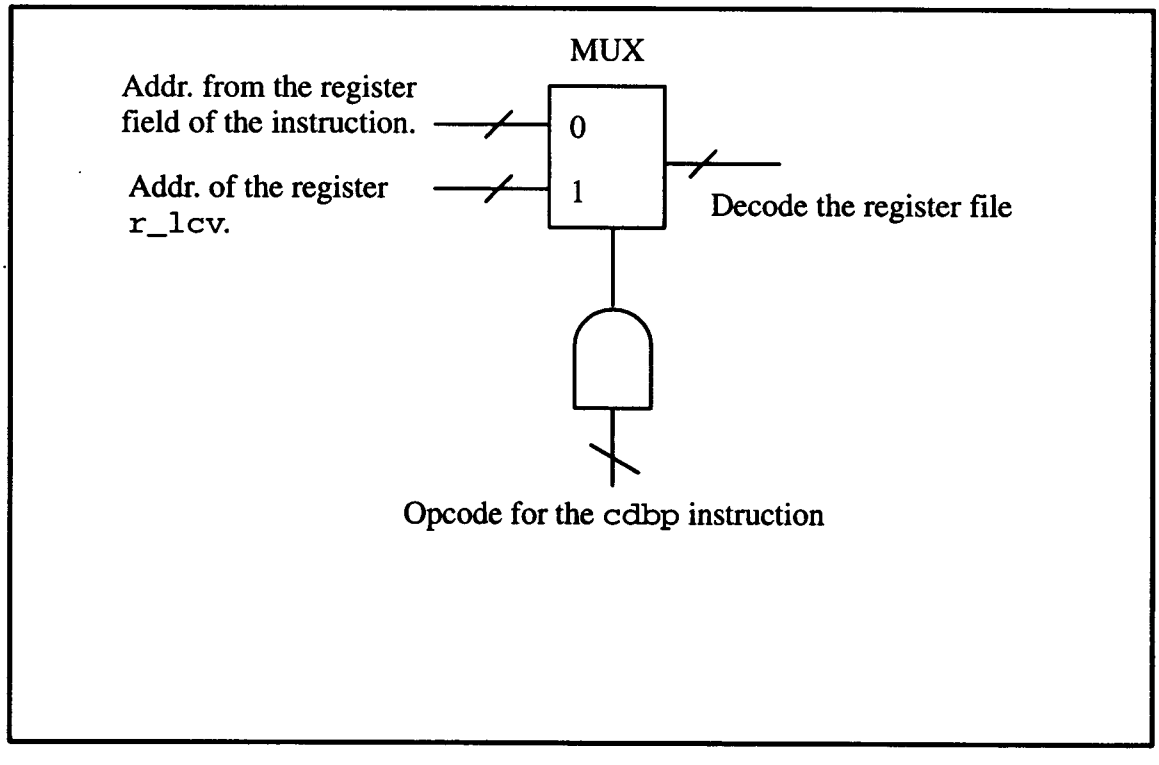


Figure 7. 5. Decoding the register file for the cdbp instruction

7. 1. 5. Hardware required to implement the `ldi` and `std` instructions

The `ldi` and `std` instructions are similar to other load and store instructions except that the register has to be incremented or decremented by 2. This can be easily done in the execute stage of the pipeline if the control unit is designed to give the corresponding control signals to the execution unit.

7. 1. 6. Implementation of the Double Ended Queue for the LCV

As seen in Chapter 6, the LCV in the modified design is implemented as a double ended queue. Since the queue grows both sides, care must be taken to see that the available memory is not exceeded. As the compiler has an estimate of how big can the LCV be, this is not a serious problem.

7. 1. 7. Hardware to assert the LOCK signal

As seen in section 6.1, the main processor has to assert the LOCK signal whenever it is updating the synchronization count. This has to be asserted until the count is stored back or the count is zero. Since the load and decrement instructions are used to manipulate the count, these are used in the circuit. The circuit for this is shown in Figure 7.6.

7. 2. Future work and conclusion

The results from the Figure 6.4 prove that a conventional RISC processor can support fine–grain parallelism with minimal hardware changes. As seen in chapter 5, a 6% increase in the execution speed is achieved by adding the instructions `cdbp`, `ldi` and `std`. These can be incorporated in the ISA of the SPARC processor by redesigning the control unit.

As seen earlier, using a coprocessor for inlets results in a much lower CPT. But, while designing a system with a coprocessor for inlets, issues like atomicity and coherency cannot

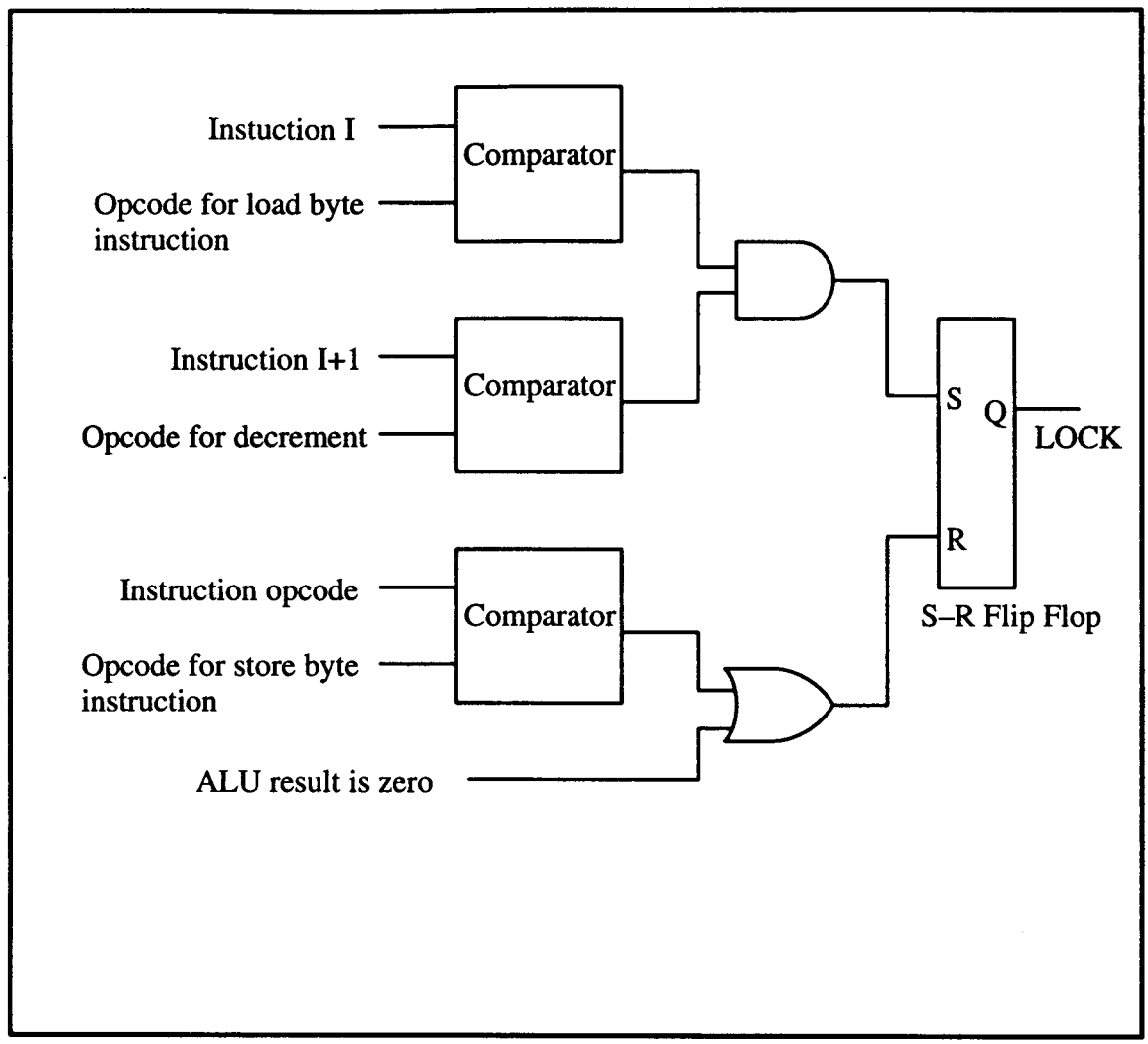


Figure 7. 6. Circuit to assert the LOCK signal

be ignored. Using a coprocessor for inlets slightly increases the overhead of various dynamic scheduling operations. For example, whenever the BHOLD signal is asserted, the main processor has to wait for accessing the frame memory. Similarly, if the POST from the coprocessor is to the currently running frame and the LOCK signal is asserted by main processor while the coprocessor is updating the count, the coprocessor has to check the count from the frame location before the decremented count can be stored back. Also as seen in section 6.1, changing the simple stack structure of the LCV resulted in 4 additional cycles for posting a thread to the currently running frame. In addition, whenever the main processor is updating the

entry count, the inlet processor is idle. But since the two processors are executing simultaneously, this CPT is shared between the processors and the overall execution is faster as shown in the Figure 6.4. In addition, the overhead due to polling, which is about 4%, is reduced.

In our design, the LCV is implemented as a double ended queue. Here, the threads posted from the inlets are appended at the bottom of the LCV. Another attractive design alternative is to post the threads from inlets to the currently running frame into the RCV of the frame rather than the LCV. In this design, the processor executes the current frame until there are no enabled threads; then the leave–threads is executed. The leave–thread, in this design, has to check the size of the RCV of the current frame before swapping to a different frame. If the RCV size is not zero, then the RCV is copied to the LCV and the current frame is executed; otherwise, execution is transferred to a different frame.

The attractive features of posting a thread to RCV rather than LCV include the implementation of the LCV as a simple stack. The additional cost involved in posting a thread (like swapping the leave–thread pointer with the thread to be posted) are removed. On the other hand, the leave–thread has to do the extra work of coping the RCV to LCV if the RCV size is not zero.

In either of these design choices for posting a thread to the current frame, the execution order of the threads from the original implementation (i.e. having a single processor with polling) is changed. In the modified design, the threads posted from the inlets are executed only when there are no threads enabled by FORKs. This modified scheduling might slightly effect the execution time of the program.

In our design, cache coherency problem is eliminated by using a common cache for both the main processor and the coprocessor. While this eliminates the coherency traffic, as seen from chapter 6, this choice leaves one of the processors waiting for the bus while the other one is using it. Another design choice is to have separate caches for the main processor

and the coprocessor. In this case again, the issues of atomicity and coherency have to be addressed carefully.

The coprocessor in our design is assumed to be similar to the main processor. Since the coprocessor mainly executes only RECEIVE, POST, and NEXT instructions, it can be much simpler and can be optimized for these instructions.

In summary, the results are encouraging and prove that fine–grain parallelism can be supported on the traditional von Neumann processors with slight hardware modifications. These also show that issues like atomicity and message handling are of utmost importance and should be considered carefully.

BIBLIOGRAPHY

[1] Arvind, D. E. Culler, R. A. Iannucci, V.kothail, K. Pingali, and R. E. Thomas, "The tagged token dataflow architecture," Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, August 1983.

[2] Arvind, and R. A. Ianucci, "Two Fundamental Issues in Multiprocessing," Proceedings of DFVLR – Conference 1987 on Parallel Processing in Science and Engineering, Bonn–Bad Godesberg, W. Germany, Springer–Verlag LNCS 295, June 25–29 1987.

[3] Arvind, R. S. Nikhil, and K. K. Pingali, "I–structures: Data Structures for Parallel Computing," Proc. of the Graph Reduction Workshop, Sante Fe, NM. October 1986.

[4] D. E. Culler, "Multithreading: Fundamental Limits, Potential Gains, and Alternatives," Proc. of the Supercomputing '91, Workshop on Multithreading, 1992.

[5] D. E. Culler and Arvind, "Resource Requirements of Dataflow Programs," Proc. of the 15th Annual Int. Symp. on Comp. Arch., pages 141–150, Hawaii, May 1988.

[6] D. E. Culler and G. M. Papadopoulos, "The Explicit Token Store," Journal of Parallel and Distributed Computing, pages 289–308, January 1990.

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

[8] D. E. Culler, K. E. Schauser, and T. von Eicken, "Two Fundamental Limits on DataFlow Multiprocessing," Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL. North–Holland, January 1993.

[9] V. Grafe and J. Hoch, "The Epsilon–2 Multiprocessor System," Journal of Parallel and Distributed Computing, 10, 1990, pp. 131–140.

[10] R. A. Iannucci, "Toward a Dataflow/von Neumann Hybrid Architecture," Proc. 15th Int. Symp. on Comp. Arch., pages 131–140, Hawaii, May 1988.

[11] B. Lee, A. R. Hurson, "Dataflow Architectures and Multithreading," IEEE computer, August 1994.

[12] M. Namjoo and A. Agrawal, "Implementing SPARC: A High Performance 32–bit RISC Microprocessor," Sun Microsystems Technical Publications, Sun Microsystems, Inc., 1987.

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

[14] R. S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?," Proc. of the 16th Annual Int. Symp. on Comp. Arch., Jerusalem, Israel, May 1989.

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

[16] G. M. Papadopoulos, and D. E. Culler, "Monsoon: an Explicit Token–Store Architecture," Proc. of the 17th Annual Int. Symp. on Comp. Arch., Seattle, Washington, May 1990.

[17] S. Sakai *et al.*, "An Architecture of a Dataflow Single Chip Processor," Proc. 16th Annual Int'l. Symposium on Computer Architecture, 1989, pp. 46–53.

[18] M. Sato *et al.*, "Thread–Based Programming for EM–4 Hybrid Dataflow Machine," Proc. 19th Annual Int'l Symposium on Computer Architecture, 1992, pp. 146–155.

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

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

[21] E. Spertus, S. C. Goldstein, K. E. Schauser, D. Culler, and T. von Eicken, and W. J. Dally, "Evaluation of Mechanisms for Fine–grained Parallel Programs in the J–Machine and the CM–5," Proc. of the 20th Int'l Symposium On Computer Architecture, San Diego, CA, May 1993.