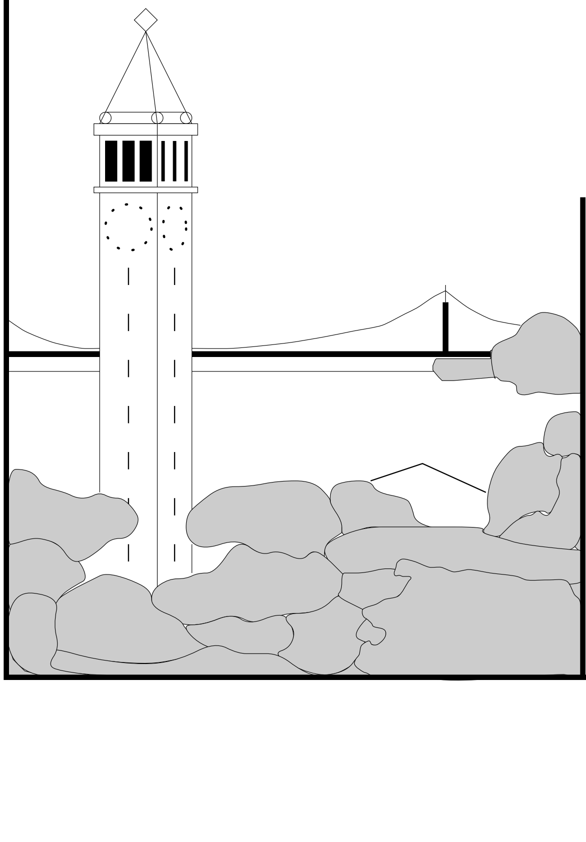
The Implementation of a Threaded Abstract Machine

Seth Copen Goldstein



Report No. UCB/CSD 94-818

May 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Implementation of a Threaded Abstract Machine

Seth Copen Goldstein

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley

May, 1994

Seth Copen Goldstein is supported by an AT&T Graduate Fellowship. This work was supported by a Presidential Faculty Fellowship (CCR-9253705). Computational support was provided by the NSF Infrastructure Grant number CDA-8722788 and the LLNL Advanced Computer Laboratory.

Abstract

This report describes TL0 and a translator for TL0, which targets both commodity workstations and commercial multiprocessors. TL0 is a portable parallel assembly language based on the Threaded Abstract Machine (TAM). TAM defines an execution model of self-scheduled threads which meets the demands that modern parallel architectures place on the compilation of general-purpose parallel programming languages. The most important features of TAM are its utilization of the storage hierarchy and the way it exposes asynchronous message events and the scheduling of computation.

This report defines TL0 in two ways: by defining the operational semantics of TL0 and by describing TLC, the target of the TL0-to-C translator. We discuss the constraints imposed on the design of the system by asynchronous messages handling. We then present a detailed analysis of the implementation of dynamic scheduling and how it is influenced by the two-level scheduling hierarchy. In addition to describing TL0, we explain how to eliminate many of the sources of overhead introduced by using C as an assembler.

The translator described is currently being used as the back end for an Id90 compiler which produces the fastest Id90 executables in the world. By defining TL0 and describing the translator, this report should allow others both to build high-level language compilers that use TL0 as a target and to port TL0 to other parallel machines.

Acknowledgments

The work described in this thesis is the result of a collaboration of the Berkeley TAM group. Working with this group has been exciting, rewarding, and, above all, fun. I want to thank everyone I worked with including Professor David Culler, Andrea Dusseau, Thorsten von Eicken, Lok Tin Liu, Steve Lumetta, Steve Luna, Richard Martin, and Klaus Eric Schauser.

I want to thank David Culler, my advisor, for constantly challenging me to think in new directions. He is a wonderful person to work with.

I am very grateful to my second reader, Kathrine Yelick, for getting me see the same old things from a different viewpoint. I want to thank her for helping me to look to the future.

This masters would never have been completed without the work done by Klaus Eric Schauser on the Id90 compiler and Thorsten von Eicken's active message layer. In addition to their technical expertise, my old officemates have always been there for me. In particular, Klaus Eric has been a constant source of inspiration. Working with him has been one of the high points of graduate school. I am sure that he is destined to be one of the best professors ever! We will miss him.

I want to thank my new officemates, Andrea Dusseau and Steve Lumetta, for not only putting up with me during the writing of this masters, but for their excellent advice. I am grateful to Chu-Cheow Lim for his careful reading. And I want to thank Nathan Tawil, final arbiter of taste, for his generosity of time, skilled editing, and support.

I want to thank my wife, Deborah Weisser, for her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Threaded Abstract Machine, TAM, grew out of work on dataflow execution models. It refines the dataflow execution model to meet the demands that modern parallel architectures place on the compilation of general-purpose parallel programming languages. It extends previous work by paying particular attention to the utilization of the storage hierarchy in the context of dynamic scheduling and asynchronous message handling. TAM defines a scheduling hierarchy that reflects the underlying storage hierarchy and allows the compiler to control the dynamic scheduling of computation.

TAM is easily understood independently of its dataflow heritage as a general framework for self-scheduling parallel threads. Threads enable other threads and generate messages, which are received asynchronously from the communication network and in turn enable threads. The key issues addressed by TAM are how resources are shared among threads, how scheduling is structured to make efficient use of resources, and how asynchronous messages are incorporated into the mainline computation.

TAM originated as a bridge between traditional dataflow and control-flow execution models. We have utilized TAM primarily as an intermediate target in compiling the high-level dataflow language Id90 to conventional parallel machines, including the Thinking Machines CM-5. The target of this compilation step is TL0, Thread Language Zero, a concrete realization of TAM. TL0 defines a portable parallel assembly language based on the TAM model. In addition to its support for functional programming languages, TL0 appears to be a natural target for other parallel programming languages that use multithreading. In particular, it seems especially suitable as a target for the compilation of parallel object-oriented languages.

Compiling down to TL0 involves compiling a high-level language into control flow among a collection of threads. A separate translator then translates from TL0 to the target machine, optimizing for specific characteristics of the instruction set. Thus, TAM decouples development of novel parallel languages and innovation in parallel machine architecture, while providing a meaningful interface between the two. Hardware design alternatives can be evaluated by how they accelerate TL0 primitives, weighted by the frequency of use of these primitives in programs.

In this report we give an exact specification of TL0 and how it is implemented on both commodity uniprocessors and commercial multiprocessors. This specification serves more than one purpose. It establishes the requirements for high-level language compilers that use TL0 as a target. The description of the TL0-to-C translation process should also provide a template for those wishing to port TL0 to other machines.

1.1 Goals of TL0

TL0 is intended to serve four related, but functionally distinct, research efforts. First, it is intended to serve as as an Id90 execution vehicle. Second, it is meant to provide a framework in which to study the compilation of languages that require dynamic synchronization and non-trivial communication. Third, it is aimed at allowing us to study the architectural mechanisms that are needed to support parallel programming. Finally, it is designed to facilitate the study of the dynamic behavior of large control-parallel applications.

TL0 provides an execution vehicle for Id90, an implicitly parallel functional programming language [10]. Its functional nature, synchronizing data structures, and non-strict execution model hold the promise of considerable parallelism, but also provide a serious challenge to the compiler writer. The compilation process starts in the MIT front end [15], which converts the Id90 program into program graphs. These are converted by the Berkeley compiler into dual graphs, on which the program is partitioned into threads [13]. The partitioned graph is then converted into TL0 code.

The last step in the compilation process is to translate the TL0 code into an executable that can be run on a real machine. This report describes the TLC translator, which converts the TL0 code into TLC, a C-based representation of TL0. The TLC code, which consists solely of C, is then compiled by a C compiler into executables.

The Berkeley Id90 compiler generates executables for both sequential and parallel machines. It supports both Sparc- and Mips-based workstations. The parallel machines it supports are the Sequent Symmetry, the nCube-2, and the CM-5. Aside from the special-purpose Monsoon machine, it provides the only realistic execution vehicle for Id90 programs.

TL0 fulfills its second goal, supporting compiler research, in two ways. First, by dividing the compilation process into two parts (from Id90 to TL0, and then from TL0 to the machine), we have been able to separate the high-level compilation aspects from the low-level details. Second, TL0 provides a means of evaluating compilation strategies. By analyzing the dynamic TL0 instruction frequencies, instead of a particular machine's instruction set, we are able to focus on the direct mechanisms that need to be optimized.

The third goal, studying parallel architectures, is met by investigating how the TL0 primitives are mapped onto machines. Current machines can be evaluated for how well they support the TL0 primitives. Indeed, once a set of benchmarks and traces for these benchmarks have been recorded, future designs can be evaluated by examining how the TL0 primitives would map to the machine.

The fourth goal, that of studying the dynamic behavior of large parallel programs, is supported by creating instrumented executables. The instrumented executables can provide us with summary statistics for individual executions of the program as well as traces that describe the behavior of the program at specified time steps. These traces, and the "movies" they allow us to make of the program's behavior help us to study the interaction of many of the variables that determine the overall performance of parallel programs on parallel machines.

1.2 Overview

In addition to describing the semantics of TL0 and its implementation on sequential processors and multiprocessors, this report highlights the main ideas of TL0. TL0 makes explicit the TAM's two-level scheduling hierarchy. By exposing scheduling operations it puts them under direct compiler control. TL0 also allows the compiler to explicitly manage all the resources needed to execute the program. While standard machine models allow the compilers to manage only the registers and memory, TL0 also places the network under compiler control. TL0's split-phase operations tolerate latency from two sources. First, they tolerate long-latency communication by allowing computation to overlap with communication. Second, they support the potential algorithmic latency inherent in the use of non-strict, synchronizing data structures, by permitting frames to suspend. Finally, TL0 addresses the incorporation of asynchronous messages into the rest of the program.

One of the main design goals of TL0 was to produce a portable parallel assembly language. The task of defining TL0 with enough detail that it could be used as a compiler target, while still keeping it portable enough to be used on different machines, involved many trade-offs. For example, TL0 includes I-structure operations, but does not define a tagged memory architecture. TL0 does not specify how inlets are activated, i.e., it does not specify if the network is polled, or if it interrupts the processor. Yet it does require certain scheduling operations to be atomic. No restriction is placed on whether operands are in registers or in memory. Indeed, there is no limit on the number of registers available to the compiler. However, each register and accessed memory location must be typed. This allows a translator to map each reference to the correct register of the target machine.

While many low-level details are left to the final translation process, TL0 does define a clear execution model. This is seen most obviously in the restrictions placed on inlets. Inlets are small routines used to handle the reception of messages from the network. To avoid the need for "interpreting" the incoming message, each inlet is matched by the compiler to a send instruction. In order to avoid deadlock, inlets must initiate only instructions that have already had resources allocated for them. This simple statement has multiple ramifications. Inlets cannot contain conditional loops. They may not send an arbitrary number of messages. Indeed, they must send messages at a lower priority than the messages sent by threads.

The design decisions left unspecified in TL0 are made concrete in TLC, the target of the translator defined in this report. How heap memory is laid out, how synchronizing I-structure operations are implemented, how inlets are activated, the size of each data type, etc. must all be defined in TLC. The decreasing abstractness from TAM to TL0 and finally to TLC involved design choices that are explained in the following chapters.

The rest of this chapter outlines the basic concepts of TAM. In Chapter 2 we define the semantics of TL0. We begin by outlining the execution model derived from TAM, and then continue by describing the operational semantics of the TL0 instructions. Chapter 3 continues the lowering process by describing TLC. The C control structures created to implement the machine model are followed by descriptions of the translation of each of the TL0 instructions. The chapter ends by describing the sequential and parallel implementations. Chapter 4 summarizes the results of some previously published studies that show the efficiency of the the tools described in this thesis.

1.3 The Threaded Abstract Machine

This section describes the TAM execution model. TAM differs in philosophy from traditional dataflow models in that it exposes communication, synchronization, and scheduling so that a high-level language compiler can attempt to optimize for important special cases. The compiler can produce efficient message-handling code that is closely integrated with the scheduling of computation. The integration presents some subtle trade-offs, since efficient message handling and efficient computation place opposing demands on the use of the storage hierarchy. TAM retains an explicit storage hierarchy to allow the compiler to mediate these demands on a case-by-case basis.

A TAM program is a collection of *codeblocks*, where each codeblock consists of several *threads* and *inlets*.

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 a stack frame in a sequential language, it is the unit of storage for each codeblock invocation.
	Continuation Vector	The data structure used to store enabled threads.
	Ready Queue	The data structure used to keep track of frames 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.
	FALLOC	The FALLOC instruction is used to invoke a codeblock, i.e. to perform a function call.

Table 1.1: *A brief description of some of the most important TAM concepts.*

Typically a codeblock represents a function or loop body in the high-level language program. Threads correspond roughly to basic blocks. The *activation frame* is the central storage resource and the critical concept for understanding TAM. As suggested by Figure 1.1, the frame provides storage for the local variables, much like a conventional stack frame. It also provides the resources required for synchronization and scheduling of threads, as explained below. Invoking a codeblock involves allocating a frame, depositing arguments into it, and initiating execution of the codeblock relative to the newly allocated frame. The caller does not suspend upon invoking a child codeblock, so the caller may have multiple concurrent children. Thus, the dynamic call structure forms a tree, rather than a stack, represented by a tree of frames. This tree is distributed over the processors, with frames as the basic unit of parallelism between processors. Finer-grain parallelism is not wasted; it is used to maintain high processor utilization. Thread parallelism within a frame is used to tolerate communication latency and instruction parallelism within a thread is used to tolerate processor pipeline latency.

Initiating execution of a codeblock means enabling threads of computation, where each thread is a sequence of instructions that cannot suspend. Each of the arguments to a codeblock potentially enables a distinct thread. A processor may host many ready frames (the activation tree is usually much larger than the number of processors), each with several enabled threads. The TAM scheduling queue is a two-level structure comprising a collection of frames, each of which contains one or more addresses of enabled threads in a region of the frame called the *continuation vector*. The compiler is permitted to specialize the frame-level

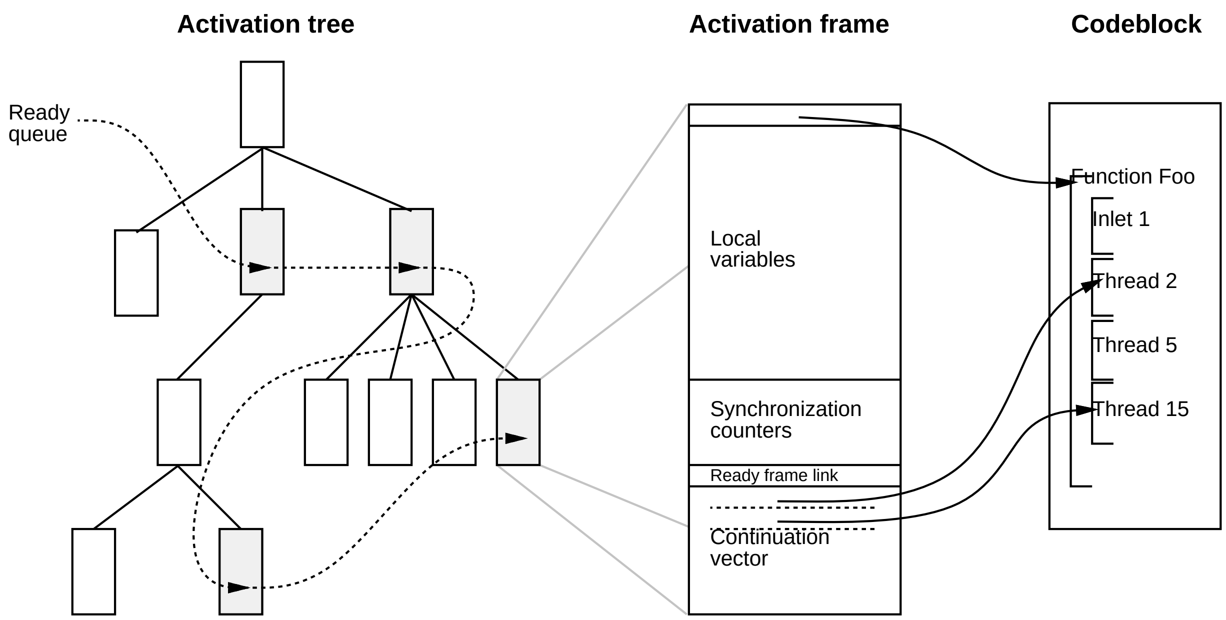


Figure 1.1: TAM *activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame, in addition to holding all local variables, contains counters used to synchronize threads and inlets, and provides space for the* continuation vector*—the addresses of all currently enabled threads of the activation. On each processor, all frames holding enabled threads are linked into a ready queue. Maintaining the scheduling queue within the activation keeps costs low: enabling a thread consists simply of pushing its instruction address into the continuation vector and, sometimes, of linking the frame into the ready queue. Scheduling the next thread within the same activation is simply a pop-jump.*

structure, but typically the data structure used is a linked list of frames per processor. When a frame is scheduled, threads are executed from its continuation vector until none remain. The last thread schedules the next ready frame. Thus, the time during which a frame is resident (i.e., the interval between one scheduling and descheduling of the frame) defines a unit of scheduling, called a *quantum*, consisting of the consecutive execution of several threads. This scheduling policy enhances locality by concentrating on a single frame as long as possible.

Instructions in a thread include the usual computational operations on registers and local variables in the current frame. The basic control flow operation is to enable, or FORK, another thread to execute in the current quantum. The SWITCH operation conditionally forks one of two threads. Threads are also enabled as a result of message arrivals. Each codeblock contains a collection of inlets, which are compiler-generated message handlers for the remote accesses and call/return linkage. For example, arguments to a codeblock invocation are sent to inlets of the codeblock with the newly allocated frame as the context. Inlets typically deposit message data into the designated frame and *post* threads into its continuation vector. Precedence between threads, i.e. data dependencies and control dependencies, are enforced using synchronization counters within the frame. A *synchronizing thread* has an associated *entry count* which is decremented by forks and posts of the thread. The thread is enabled when the count reaches zero.

TAM threads are self-scheduled; there is no implicit dispatch loop in the model. Thus, the compiler can control the scheduling by how it chooses to generate forks, posts, and entry counts. There is also no implicit saving and restoring of state, so the compiler manages storage in conjunction with the thread scheduling that it specifies. Since threads do not suspend, values that are local to a thread can be kept in registers. In addition, whenever the compiler can determine that a collection of threads always executes in a single quantum, it can allocate values accessed by these threads to registers. Observed quanta are usually much larger than what static analysis could determine, because several messages arrive for a frame before it is actually scheduled. Since the frame switch is performed by special compiler-generated threads, it is possible to take advantage of this dynamic behavior by allocating values to registers based on expected quantum sizes and saving them if an unexpected frame switch occurs.

Accessing the global heap may be a long-latency operation, and, rather than causing the processor to stall, is treated as a special form of message. 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 ongoing 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. The global heap supports synchronization on an element-by-element basis, as with I-structures [1]. Thus, there are two sources of latency in global accesses: A hardware communication latency occurs if the accessed element is remote to the issuing processor and, regardless of placement, a synchronization latency occurs if the accessed element is not present, causing the request to be deferred.

Compared to dataflow execution models, TAM simplifies the resource management required to support arbitrary logical parallelism. A single storage resource—the activation frame—that is naturally managed by the compiler as part of the call/return linkage represents the parallel control state of the program, including local variables, synchronization counters, and the scheduling queue. TAM embodies a simple two-level scheduling hierarchy that reflects the underlying storage hierarchy of the machine. Parallelism is exploited at several levels to minimize idle cycles while maximizing the effectiveness of processor registers and cache storage.

Chapter 2

TL0: An Implementation of TAM

This chapter describes TL0, a concrete machine description and assembly language for the Threaded Abstract Machine TAM. Our goal in defining TL0 is to expose the essential features of conventional parallel machines, allowing us to compile Id90 and other high-level languages into efficient executables on machines such as the CM-5.

TL0 is designed to be a portable target for high-level parallel language compilers. Thus we specify only those architectural features necessary for such a compiler. For instance, TL0 does not restrict the number of registers, nor does it specify widths of data types, or even the low-level details of the network. Instead, it focuses on those aspects of parallel machines that make compiling for them so hard—for instance, the scheduling of threads and the interaction of the network with the main line of computation.

Section 2.1 describes the underlying machine defined by TL0, including the storage hierarchy, register set, and execution model. The semantics of the heap and its operations are defined in Section 2.2. Section 2.3 summarizes the components of the machine state and presents the ideas discussed in previous sections in a single coherent model. Section 2.4 spells out the operational semantics of TL0 assembly language.

2.1 The Structure of The Machine

We begin our definition of the TL0 abstract machine by describing the underlying storage model. Then we present the execution model by defining frames, threads, inlets, and codeblocks. Finally, we describe how our assumptions about the interconnection network affect the model.

2.1.1 The Storage Model

TL0 implements the TAM storage model with one level of code storage and three levels of data storage. The code storage contains all the codeblocks that define the program. All accesses to code storage are fast local read-only operations.

The three levels of data storage are registers, frame store, and heap. Registers are the fastest level of the storage hierarchy. For values that must be kept between quanta, the slower levels of the memory hierarchy must be used.

The frame store contains the activation frames created during execution of the program. Each activation frame corresponds to an invocation of a codeblock. An activation frame (often referred to as a *frame*) corresponds to a stack frame in a sequential language. The frame holds the local variables, continuation vector, thread entry counters, and system information needed for the lifetime of the activation. While the frame store is potentially distributed among the processors, each frame is local to a single processor. Therefore, all accesses made to a frame are local memory accesses.

The inlets and threads of the codeblock for which the frame was allocated use the frame for all their local data. Indeed, when a codeblock is invoked and a frame allocated, all the storage resources required to run that frame are satisfied. The frame that an inlet may modify is the inlet's *target frame*. The frame that a thread may modify is the *current frame* or the *running frame*.

The heap contains objects that are not local to any particular codeblock invocation. It includes both statically and dynamically allocated arrays. The heap may be distributed among the processors, and individual heap objects, unlike frames, may also be distributed over the processors. All accesses to the heap are split-phase operations.[1] In addition to data values, memory locations in the heap may have associated tags that provide per-element synchronization required for I-structures and M-structures.

2.1.2 Registers

The TL0 machine defines six dedicated registers: FP, TFP, PC, TPC, QP, LCVPTR.

fp (frame pointer) holds a pointer to the current frame.

tfp (thread frame pointer) is used only when an inlet is executed. It holds a pointer to the frame that was interrupted by the message[2] that activated the current inlet. It is restored to the FP register when the inlet completes.

pc (program counter) is the current program counter used for executing both inlet and thread instructions.

tpc (thread program counter) is used to hold the value of the program counter that was interrupted by the execution of the current inlet. It is restored to the PC when the inlet that is currently executing finishes.

qp (ready queue pointer) points to the set of ready frames.

lcvptr (local continuation vector) points to the local continuation vector for the current frame.

In addition to these six dedicated registers, TL0 allows the compiler to define an unlimited number *codeblock registers*. The lifetime of the codeblock registers is a single quantum, or in the case of registers used in inlets, the lifetime of the inlet. In keeping with the concept that TL0 is a portable assembly language, we do not restrict the number of registers available to the compiler. However, in order to support such portability, all registers are strongly typed. The TL0 translator maps the codeblock registers onto real registers in a host processor.

The differentiation between inlet registers and thread registers is deliberately not enforced in the model. Our intent is that both inlets and threads have access to the same register set, which is partitioned by the

[1] Split-phase operations consist of two parts, a request and a reply. The first phase initiates the operation, allowing the processor to continue execution. The second phase completes the operation by returning the requested results. This allows latency to be tolerated by masking it with useful computation.

[2] In this paper we use "message" loosely. When a "message" is "sent" to a frame on the same processor, no actual messages are sent; rather, the inlet is called directly. Thus, we use "call an inlet" interchangeably with "send a message to an inlet."

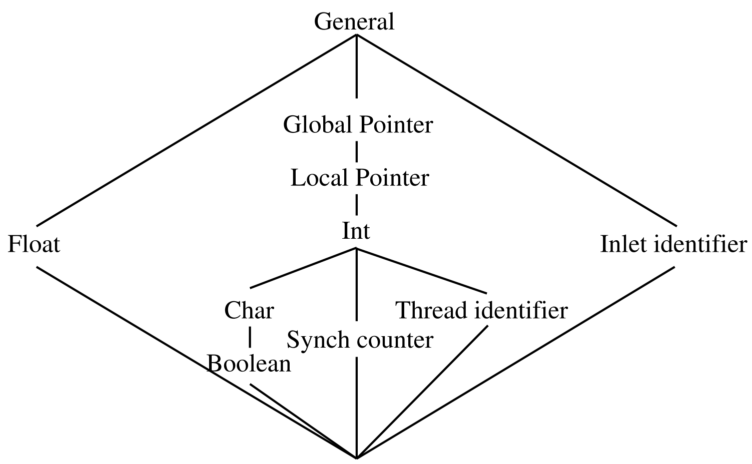


Figure 2.1: *Relative data type sizes assumed by* TL0. *Data types lower down in the lattice can fit inside datatypes higher up in the lattice.*

compiler. In this respect TL0 differs greatly from other multithreaded machines which provide different contexts for different priority levels (e.g., threads and inlets).

2.1.3 Data types

All registers and memory locations in TL0 are strongly typed, yet the sizes of the types are left undetermined. This fits with TL0's role as a portable assembly language. It allows a compiler to specify what goes into a memory location or register without constraining the underlying hardware on which TL0 can be implemented. On the other hand, it limits the ability of the compiler to reuse registers and frame slots across types. Such low level optimization is left to the TL0 translator.

While TL0 does not specify the size of any data types, it does require that certain relations hold among them. For example, char must be no larger than int, and a general must be large enough to hold a value of any data type. The constraints on relative size of datatypes are summarized in the lattice in Figure 2.1.

In addition to the usual arithmetic types (*boolean*, *char*, *int*, *float*) and the pointers (*codeblock pointer*, *frame pointer*, *structure pointer*), TL0 includes four special types. A *synch counter* type is used for the memory locations that record thread entry counts. All variables of this type must be allocated in frame slots. The types *thread identifier* and *inlet identifier* are used to specify threads and inlets, respectively, within a codeblock. These identifiers are small integers. Finally, memory locations or registers of type *general* can hold any value. Locations are not tagged, and therefore consumers and producers of values stored in generals must agree on the type of the data.

TL0 uses two forms of continuation: thread continuations and inlet continuations. A thread continuation is a thread identifier and its associated frame. Since thread identifiers are always either specified relative to the current frame (i.e., as the operand of a FORK) or stored in the LCV or RCV[3], the frame is always implicit and the identifier itself denotes a complete continuation. Unlike thread continuations, inlet continuations are passed between frames and must explicitly specify the frame part of the continuation.

[3]The *local continuation vector* (LCV) and the *remote continuation vector* (RCV) are described in Section 2.1.4.

2.1.4 The Execution Model

The TL0 execution model is designed around the storage hierarchy described above. It attempts to allow full utilization of registers and local memory, while supporting long-latency split-phase operations to the heap. The two-level scheduling hierarchy is directly supported by the TL0 instruction set so no hidden scheduling mechanisms are needed. TL0 is designed to run on parallel processors each with its own thread of control.

The Activation Frame

Central to the execution model is the concept of the activation frame. In addition to its duty as the data repository for a function activation, it is used for both frame and thread scheduling. New frames are allocated in response to codeblock invocations. This is analogous to function calls on a sequential machine. In order to support parallelism and to hide latency, frame allocation (initiated by the FALLOC instruction) is a non-blocking operation. As a result, computation in TL0 will unfold as a tree, instead of a stack (as on uniprocessors).

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 to by the FP register.[4] All local memory accesses (e.g. to non-heap variables) that occur in either threads or inlets are relative to the current frame, which is pointed to by the FP register.

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 RCV holds the enabled threads for a ready frame. The LCV contains the enabled threads for the running frame. Every codeblock 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 codeblock.[5]

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.

The only implicit frame scheduling policy set forth by the execution model is that once a frame becomes resident it will remain resident as long as there are enabled threads in its continuation vector. Every time a frame is scheduled the first thread run is the *enter thread*. The enter thread, specified with the SET_ENTER instruction, can be changed during the life of the frame. After the enter thread is executed, the enabled threads are executed. When there are no more enabled threads the *leave thread* is executed. The leave thread, which is set with the SET_LEAVE instruction, is responsible for saving any necessary state for the current activation, and must also schedule one of the ready frames.

The scheduling of frames is handled by the SWAP instruction. It removes one of the ready frames from the ready list and enables it by making it the current frame and executing its enter thread. The time during which a frame remains resident, from the time the enter thread is run until the time the SWAP is executed, is

[4] The model can be easily extended to allow multiple active frames on a single processor.

[5] This is possible because a thread may appear only once in the continuation vector.

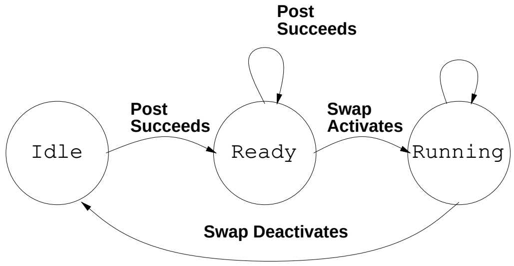


Figure 2.2: *State transitions taken by frames due to post and swap instructions.*

called a *quantum*.

Notice that no implicit state is saved when the frame is descheduled. If any registers are to be saved (or restored) they must be explicitly saved (or restored) by the leave (or enter) thread. On the other hand, registers remain valid not only within a single thread, but across all the threads that are scheduled in the same quantum.

Threads

While a frame provides the storage for a particular activation, the work is carried out by the threads of its associated codeblock. A thread is a sequence of instructions that executes from beginning to end without suspension or branching. It most clearly resembles a basic block in that it has only one entry point and always executes to the end. All of the threads that belong to a single codeblock reference the same frame layout and the same set of TL0 registers. Each thread in a codeblock is identified by a small unique integer.

All control transfer is thread-based and is realized by the FORK, SWITCH, and POST instructions.[6] FORK describes basic control flow; it attempts to enable a thread in the current frame. Conditional control flow is specified with SWITCH, which conditionally forks to one of two threads.

Threads can be either unsynchronizing or synchronizing. If an unsynchronizing thread is forked, it is always enabled. If a thread is synchronizing, it has an associated synchronization counter in the frame. A FORK of a synchronizing thread decrements the counter and, if the counter becomes zero, enables the thread.

When a FORK succeeds, the thread address is put onto the local continuation vector (LCV), the structure that holds the list of threads that are ready to run for the current frame. At the end of every thread is a STOP instruction, which removes an enabled thread from the LCV and transfers control to it.

In addition, threads can be enabled from inlets by a POST instruction. Whereas FORK enables a thread from another thread in the currently running frame, POST instructions occur in inlets and enable a thread for the target frame of the inlet. The target frame may be the currently running frame, an idle frame, or a ready frame. A POST of a synchronizing thread requires decrementing the synchronization counter, just as a FORK does. However, if a post succeeds, the enabled thread address is put onto the remote continuation vector (RCV) of the target frame. Furthermore, if the frame was idle and a thread was enabled by the POST, then the frame becomes ready and is moved to the ready queue. (See Figure 2.2.)

The distinction between the RCV and LCV is not incidental. It is at the heart of why TL0 is an effective target machine for compiling implicitly parallel languages. Both the threads in the RCV and any control information

[6]The remaining control instruction, SWAP, schedules a frame and its enter thread as described above.

used to manipulate the RCV are kept in the frame to which they are related. The cost of manipulating the RCV is thus higher than that of manipulating the LCV, which is managed by the LCVPTR register. Further, the LCV itself can be kept in a special processor cache—since there is only one per processor, not one per activation. Because these two objects are distinct, thread- and inlet-level control instructions do not have to synchronize in order to enable more work for a frame. In addition, since FORKs are much more common than POSTs, the overall cost of control is lowered.

Inlets

While threads perform the useful computation that the frame was allocated to accomplish, inlets provide an interface between the frame and the rest of the computation. Indeed all interframe communication in TL0 is performed by sending messages to inlets.

Messages are initiated with the SEND instruction, which packages its operands, of arbitrary type and number, into a message and sends the message packet to the inlet and frame specified in the instruction. The destination of a message, specified by the inlet/frame pair, is essentially a continuation. Messages are non-blocking, so execution continues to the next instruction. When a message is received, the FP register is copied into the TFP register and the destination frame in the message is loaded into the FP. In addition, the is stored in the TPC and then loaded from the message. The inlet specified in the message is then executed.

An inlet is a sequence of instructions intended to handle a specific message. In other words, an inlet is designed to handle a message of a fixed number of arguments with fixed types. It is matched to a particular SEND instruction. Typically, an inlet will receive the data of the message, store it into the destination frame, and post threads relative to that frame. If the inlet expects any data it will have a RECEIVE instruction as its first instruction. This instruction contains a list of all the data elements the inlet expects to receive. It stores each data element into the frame. The last instruction of every inlet is the STOP instruction.

During execution of the inlet, the inlet may access the inlet registers, the TFP register, and any frame slots in the destination frame. Further, if the FP and TFP are the same, the inlet may also access the thread registers. An inlet may preempt the currently running thread.[7] It may not, however, preempt another inlet. Generally, the inlet's target frame and the current thread frame can be different, but they may be the same.

In order to support code sharing and conditionals, inlets may FORK or SWITCH to other inlets. Inlets that are the target of such a fork may be synchronizing, but may not receive messages. If an inlet forks another inlet, the FORK or SWITCH must be the penultimate instruction. Inlets may only fork other inlets, not threads; they may post to threads. The POST instruction attempts to enable a thread for the frame associated with the message, whether the frame is currently running, idle, or ready. If the thread is enabled and the frame was idle, the frame is placed on a queue of ready frames.

Although inlets may contain arbitrary code, subject to the constraints mentioned here and below, their *raison d'être* is to service the network and, thus, they should complete quickly. They may not contain any operations which have unknown completion times. For instance, operations that would require a memory allocation in inlets are not permitted. Also, inlets are limited to replying with only a single message, as explained in Section 2.1.5.

[7] Alternatively, inlets may execute on a separate network processor with access to the frame store.

Atomicity Requirements

Inlets and threads cooperate in determining the flow of computation, but inlets may preempt threads to handle incoming messages. The two priority levels share the resources of the machine. The thread synchronization counters are used by both FORKs in threads and POSTs in inlets, so thread FORKs must be atomic with respect to inlet execution. The remote continuation vector of the frame is manipulated by both POSTs and SWAPs, as is the global ready frame queue, so SWAP must also be atomic. Finally, the network is shared between SEND instructions in threads and REPLY[8] instructions in inlets, so SEND must be atomic.

No position is taken on whether the above requirements are met by using specific atomic instructions (e.g., atomic decrement), by enabling/disabling message interrupts, or through judicious placement of network poll instructions.

2.1.5 The Network

TL0 is intended to provide an efficient abstract machine for compiling to parallel machines. In this respect, it makes certain assumptions about the interprocessor communication network. First and foremost, it assumes that the network is accessible to the user program, and, in particular, that the user program may—with no interference from the operating system—send messages directly to and extract messages directly from the network.

The only assumption made about the network fabric in TL0 is that it is reliable. In particular, TL0 is designed to accommodate networks that deliver messages out of order.

Messages are inserted into the network with the SEND instruction.[9] It constructs a message packet that consists of the address to be executed at the destination, the frame pointer, and the other arguments in the instruction. Once the message is constructed, SEND injects it into the network. Messages inserted into the network will eventually arrive at their destination, invoking the inlet specified at the head of the message.

We do not take a position on the maximum message length, leaving the packaging of messages up to the TL0 translator. Though TL0 does not restrict the length of any message, it is assumed that sending small messages is possible and inexpensive.

Although the conceptual network as described above is very simple, it is instructive to look behind the scenes in order to explain the constraints we place on inlets. In order for a SEND to complete it may have to make "room" in the network by processing incoming messages. Since an incoming message will invoke an inlet, it is possible for the inlet itself to try to send a message; this could obviously lead to deadlock. This situation is prevented by dividing the network into two conceptual networks[10] (a send network and a reply network) and by allowing inlets to send only one message, which must be a *reply*.

The send network is used only for messages initiated by SEND instructions in threads. Inlets that receive these messages may themselves send only a single message on the reply network. If a send is blocked it will process incoming messages on both networks. However, if a reply is blocked it will only drain the reply network. This procedure guarantees that a processor will eventually be able to inject messages into the network.

The above division of the network is actually just a special case of the more general principle that inlets

[8]TL0 does not actually define a REPLY instruction. Instead, all SEND instructions occurring in inlets use the reply network.

[9]The TL0 SEND instruction coupled with inlets are the embryonic ideas behind active messages [16].

[10]Or a network with two priorities.

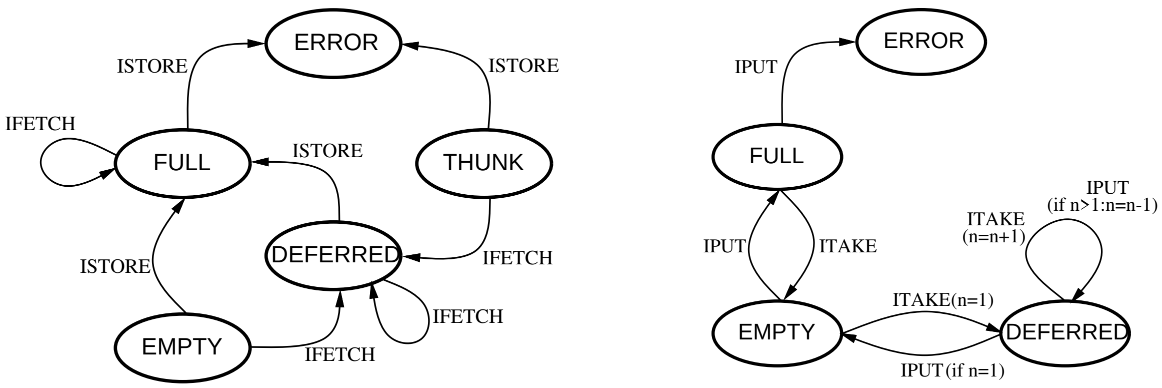


Figure 2.3: *State transitions for I-structures (on the left) and M-structures (on the right).*

may not start any operation for which resources have not yet been committed. A reply message will only be sent if the request message has allocated resources for it to complete; i.e., the reply will be sent to an already allocated frame.

2.2 The Heap

One of the key motivations for TL0's inlet/thread dichotomy is to support split-phase operations. In TL0, the parallel call is an example of a split-phase operation. The call is initiated with an FALLOC and ends when an inlet is invoked by the callee. This structure is also followed for all non-local data access operations, e.g. for heap operations.

There are two kinds of heap objects: packed and unpacked. The unpacked heap objects, I/M-structures [1], provide per-element synchronization. The packed heap objects, A-structures, provide synchronization on a per-object basis.

2.2.1 I-Structures and M-Structures

Every location of an I/M-structure consists of a data element and a synchronization tag. The data element is large enough to hold any data type (i.e., it is a general). The synchronization tags can take on one of four values: `empty`, `full`, `deferred`, and `thunk`.

When an I-structure is allocated, every tag is initialized to the `empty` value. Values are fetched from an I-structure with an IFETCH instruction and stored into an I-structure with the ISTORE instruction. As can be seen in Figure 2.3, the tags are used to implement write-once semantics for I-structures. When a data value is stored into an element, its associated tag is set to `full` and can never be altered. Every reader of an element always gets the same value—even if the read is issued before the write has completed. In the case where reads happen before writes, the read is deferred. If an element goes into the deferred state, then each request issued to it is enqueued on a deferred list. When the value is finally stored into the element, every deferred request is satisfied. TL0 does not require any particular arrangement for the deferred list.

M-structures work essentially the same way as I-structures. The elements of this structure act like semaphores. Upon allocation all the elements are initialized to `empty`. For each ITAKE instruction that

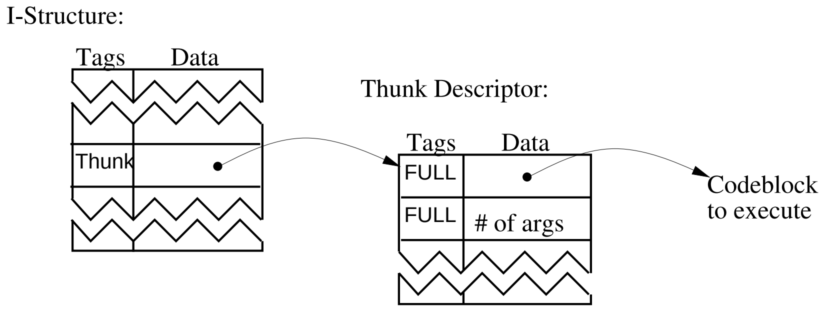


Figure 2.4: *Schematic representation of a thunk. The I-structure element which is a thunk points to a thunk descriptor. The descriptor points to a codeblock to be invoked and a (potentially empty) list of arguments to be supplied to the codeblock.*

completes, there is exactly one IPUT instruction executed. If multiple ITAKE instructions have been executed then they will defer in the same way that IFETCHs defer. However, when an IPUT is executed, only one of the deferred requests will be serviced. Note that it is an error for an IPUT to be issued to an element that already has data.

2.2.2 Thunks

I-structures have an additional tag, `thunk`, which indicates that an unevaluated closure is stored in the element. If the element stored in an I-structure is a thunk, then when an IFETCH is issued to that element, the tag value becomes `deferred` (with the original request enqueued on the deferred list) and a frame is allocated to carry out the computation specified by the thunk. The computation will result in a value being stored back into the element that had the thunk. The value stored for a thunk is a pointer to another I-structure, which contains the closure to be executed. (See Figure 2.4.)

In the current version of TL0, only compile-time thunks are created. These thunks point to a closure of zero arguments—in other words, to a codeblock which expects no arguments. When the thunk is initiated, a frame is allocated for the codeblock specified. The frame terminates by storing a value into the slot that previously had the thunk. Thus, the codeblock knows where to store the result.

2.2.3 A-Structures

A-structures are heap objects which have only a single synchronization tag for the entire object. These structures are used primarily to represent write-once printable strings. When a new A-structure is created the synchronization counter is initialized to the size of the object. As each AWRITE is executed the counter is decremented by one. Any AREAD executed to any element of the structure before the entire structure is filled in will be deferred. Figure 2.5 shows the state transitions for A-structures.

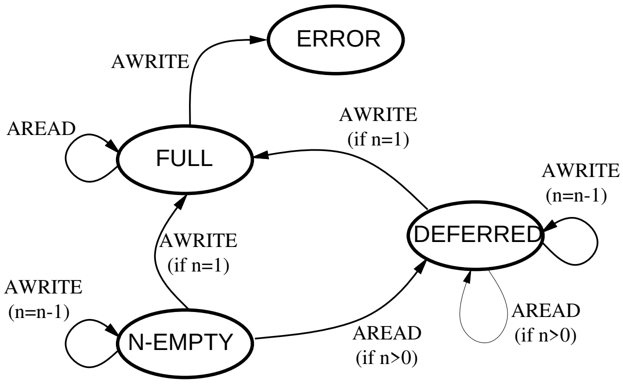


Figure 2.5: *State transitions for A-structures.* n *is initialized to the size of the structure.*

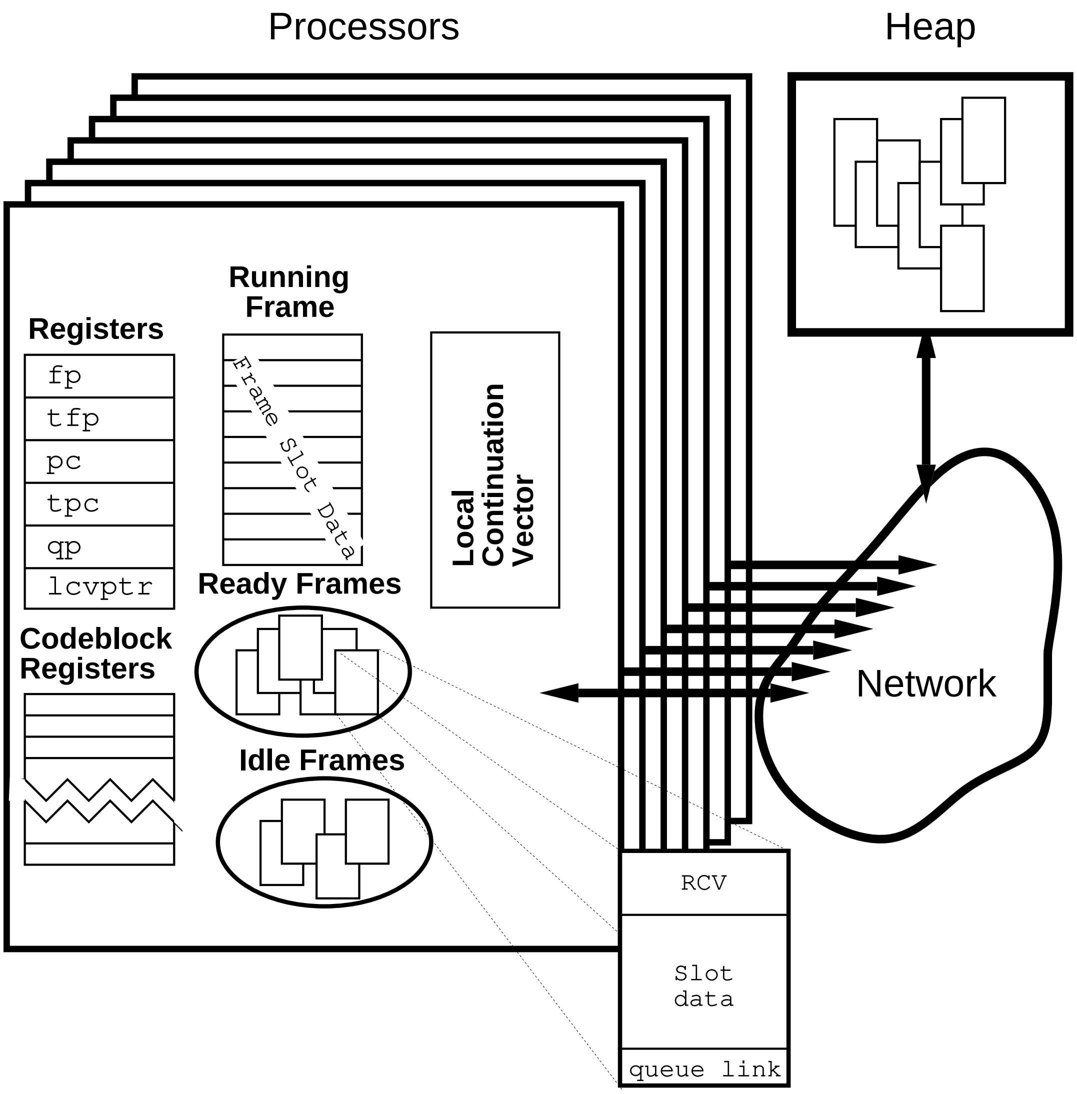


Figure 2.6: *A pictorial representation of the state of the machine.*

2.3 Machine State

In summary, the state of the TL0 machine can be described by the state of each processor, the contents of the heap, and the state of the network. (See Figure 2.6.) Each processor is described by the contents of the TL0 registers (FP, TFP, PC, TPC, QP, LCVPTR, and the currently active codeblock registers), the current LCV, the set of idle frames, the set of ready frames and their respective contents and remote continuation vectors. The heap is described by the tags and values associated with the slots of every I-structure and M-structure and the synchronization counter and data associated with every A-structure. The network is described by the undelivered messages that it contains.

The machine characterized above is easy to describe, but misses some of the original intent of TL0. It implies a single processor that executes both threads and inlets, or two similar processors, one executing threads and the other inlets. (In the second processor the thread-level execution would essentially involve a polling loop.) A different definition of the TL0 machine would include an inlet frame pointer register (IFP) and an inlet program counter register (IPC) instead of the TFP and TPC registers defined in Section 2.1.2.

If we remove the TFP and TPC registers and define IFP and IPC registers, then we can imagine a system with two different processors, one dedicated to computation and one to communication. With this model, arriving messages do not interrupt computation; instead the IPC is loaded with the handler address at the head of the message and executes the inlet concurrently with the threads. In this case, almost all the instructions are the same, except that in defining instructions that run at inlet level we replace FP with IFP, TFP with FP, and PC with IPC. In particular, when a message is received, the target frame is loaded into the IFP and the address of the inlet specified in the message is loaded into the IPC. When the inlet STOP is executed, control returns to the PC.

2.4 The TL0 Language

A TL0 program is a collection of codeblocks and global structure definitions. A codeblock represents the compilation of a sizable chunk of the high-level language program text, typically a function or loop body. A codeblock contains a declaration section, an inlet section, and a thread section. The declaration section describes external references, named constants, activation frame storage layout, and register usage within inlets and threads. The inlet section specifies the interface through which the codeblock receives messages, including arguments, return values, and heap-access responses. The thread section represents the computational body of the codeblock. Each inlet and thread is a simple sequence of instructions.

2.4.1 A Simple Parallel 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 source of parallelism. Arguments must be communicated to these parallel calls and the final result requires synchronization of the two 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 shown in Figure 2.7.

```
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 my address to caller
    STOP
INLET 11                                  % receive argument
    RECEIVE islot0.i
    POST 0.t "default
    STOP
INLET 14                                  % receive frame pointer of first
    RECEIVE pfslot1.pf                    % recursive call
    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
    RECEIVE pfslot2.pf                    % recursive 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
THREAD 1                                  % argument is <2
    MOVE ireg0.i = 1.i                    % result for base case
    FORK 6.t
    STOP
THREAD 2                                  % arg >=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[11.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[11.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.pf[retip.j] <- ireg0.i     % send result to parent
    FFREE fp.pf "default                  % de-allocate 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...
```

Figure 2.7: *TL0 code for function* `fib`.

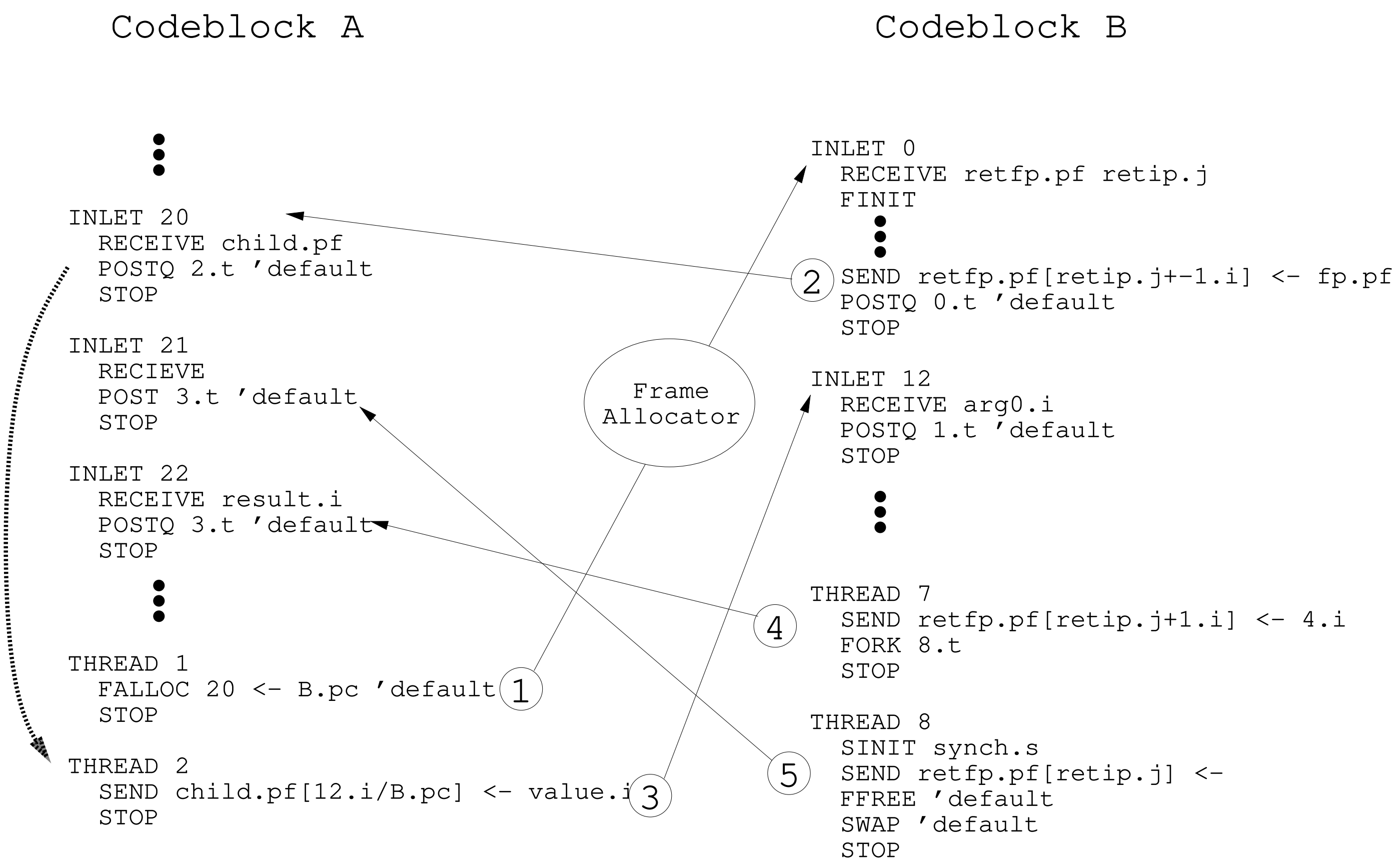


Figure 2.8: *Example code with flow of control for a call and return drawn in. It shows a call from a frame of codeblock A to a frame of codeblock B. The sequence of messages involved in the call linkage are numbered 1–5. Note that the frame allocator increments the return inlet number before invoking Inlet 0. The shaded line shows the integration of Inlet 20 into the mainline computation.*

Let us begin our execution scenario 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 frame location islot1 and a fork of either Thread 1 or Thread 2 based on the result of the comparison. (TL0 frame slots and registers are statically typed and referenced symbolically. The actual size of each type is implementation-dependent.)

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 3 for execution. Therefore, at the end of Thread 2, f will continue with Thread 3, 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 frame slot pfslot1. Inlet 16 posts Thread 4 using the default frame scheduling policy and enables the frame.

Thread 3 computes an 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 use of the storage hierarchy.

2.4.2 The Call Interface

Inlets are the external interface to a codeblock. Inlet 0 is always the call inlet that receives the return continuation. It may also receive the arguments of the invocation. Any extra arguments are passed to the consecutive inlets starting at inlet 11.

Results are returned to the parent at the inlet specified by the return continuation, which is the inlet immediately following the one that receives the child frame pointer specified in the FALLOC instruction. If there are multiple results they are returned to the consecutive inlets after the primary return inlet.

Figure 2.8 shows an example call and return. It begins with the FALLOC in Thread 1 of codeblock A. The frame allocator increments the return inlet and sends a message to inlet 0 of codeblock B, which initializes the new frame and replies to the caller by sending a message to the specified `retip.j` inlet minus 1, which in this case is Inlet 20. Inlet 20 posts Thread 2, which sends the first argument to its child, specified by the frame slot `child.pf`. Eventually, the child sends back its result to the Inlet 22. It then forks Thread 8, which is synchronizing. Thread 8 will fire when the computation is completed in the child frame. It will then send back a signal to the parent indicating that is done.

The above example describes the call interface using the default, or non-strict, frame allocation policy. There are two other polices: the strict policy, used to invoke strict functions, and the kloop policy, used to invoke k-bounded loop [6] frames. The strict protocol allocates the required amount of memory and returns a pointer to the newly allocated and as yet uninitialized memory. It is up to the caller to initialize the frame by sending a message to inlet 0. The kloop protocol allocates a set of uninitialized frames and sends a message back to the parent when the allocation has been completed.

In all three cases, the child frame receives as arguments the address of its parent and an inlet address (relative to the parent's codeblock) to which it will return results.

data type	TL0 abbrev
Boolean	.b
Character	.c
Integer	.i
Float	.f
Synchronization counter	.s
Inlet descriptor	.j
Thread descriptor	.t
Frame pointer	.pf
Codeblock pointer	.pc
Structure pointer	.ps
General	.g

Table 2.1: *Type suffixes.*

2.4.3 The Instruction Set

In this section we define the instruction set architecture of TL0. We define the operational semantics of most instructions by their state transitions. Some are defined by reducing them to simpler instructions. As TL0 does not take a position on many low-level implementation details, the pseudo-language used to define the instructions often intentionally uses abstract operations. For instance, how an enabled thread is placed on the LCV, the structure of an LCV, etc. are all ignored. Instead, the LCV is viewed abstractly as a set and the enqueue is described using set union.

2.4.4 Typeface Conventions

In the following pages we define the TL0 instruction using pseudo-code. The typeface conventions are as follows.

TL0 instructions are in	SMALL CAPS.
Operands are in	*italics*.
Dedicated registers are in	`courier`.
Pseudo-language keywords are in	**bold**.
Library routines are in	roman.
System structure field names are in	`courier`.
Continuations are enclosed in	⟨brackets⟩.

2.4.5 Data Types

As explained previously, since TL0 is meant to be a compiler target, it has the unusual feature for an assembly language that every argument is typed. The types are denoted with suffixes that must be appended to every variable, register, or constant. The valid suffixes for the types are summarized in Table 2.1.

Instruction	operation	data types
MOVE dst = src1	$dst \leftarrow src1$	*any*
ABS dst = src1	$dst \leftarrow \lvert src1 \rvert$	i,f
ADD dst = src1 src2	$dst \leftarrow src1 + src2$	i,f
SUB dst = src1 src2	$dst \leftarrow src1 - src2$	i,f
MUL dst = src1 src2	$dst \leftarrow src1 \times src2$	i,f
DIV dst = src1 src2	$dst \leftarrow src1 \div src2$	f
	$dst \leftarrow trunc(src1 \div src2)$	i
MOD dst = src1 src2	$dst \leftarrow src1 - \lfloor src1 \div src2 \rfloor \times src2$	i
MAX dst = src1 src2	$dst \leftarrow max(src1, src2)$	i,f
MIN dst = src1 src2	$dst \leftarrow min(src1, src2)$	i,f
NOT dst = src1	$dst \leftarrow \neg src1$	b
AND dst = src1 src2	$dst \leftarrow src1 \wedge src2$	b
OR dst = src1 src2	$dst \leftarrow src1 \vee src2$	b
XOR dst = src1 src2	$dst \leftarrow src1 \oplus src2$	b
BITNOT dst = src1	$dst \leftarrow \neg src1$	i
BITAND dst = src1 src2	$dst \leftarrow src1 \wedge src2$	i
BITOR dst = src1 src2	$dst \leftarrow src1 \vee src2$	i
BITXOR dst = src1 src2	$dst \leftarrow src1 \oplus src2$	i

Table 2.2: *The* TL0 *arithmetic and logical operators.*

Instruction	operation	data types
EQ dst = src1 src2	$dst \leftarrow src1 = src2$	b = *any any*
NE dst = src1 src2	$dst \leftarrow src1 \neq src2$	b = *any any*
LT dst = src1 src2	$dst \leftarrow src1 < src2$	b = i,f,c i,f,c
LE dst = src1 src2	$dst \leftarrow src1 \leq src2$	b = i,f,c i,f,c
GT dst = src1 src2	$dst \leftarrow src1 > src2$	b = i,f,c i,f,c
GE dst = src1 src2	$dst \leftarrow src1 \geq src2$	b = i,f,c i,f,c
ISPTR dst = src1	$dst \leftarrow src1 \leq 256$	b = ps

Table 2.3: *The* TL0 *comparison operators.*

In addition, a variable may be cast to or from a general by the addition of a second suffix. If a variable is of type general (i.e. if its first suffix is `.g`) then it may be accessed by appending a second suffix of a basic type. For example, `var.g.i` accesses the variable `var`, which is a general, as an integer. If a variable is a basic type, it may be cast as a general by appending the general suffix, `.g`.

2.4.6 The Instruction Set

ALU Instructions

The ALU instructions include all the instructions that perform arithmetic or logical operations on one or two operands. These include arithmetic, logical, comparison, and type conversion operations. The unary instructions have two operands which specify a source and a destination. The binary operations have three operands which specify two sources and a destination. The operands must obey certain type rules.

In general, the arithmetic operations can be performed on integers and floating point values, the logical

Instruction	conversion	caveats
ITOF dst = src	int to float	—
ITOC dst = src	int to char	undefined if not in 0..255
ITOP dst = src	int to pointer	undefined if not in 0..255 result is a small int in a pointer
FTOI dst = src	float to int	undefined if not in $-2^{31}..2^{31-1}$
CTOI dst = src	char to int	—
PTOI dst = src	pointer to int	undefined if not a small int in a pointer

Table 2.4: *The* TL0 *conversion operators.*

operations on integer and boolean operands. The comparison operations can operate on any type and produce only boolean results. The type conversion operations are used to move between types with the exception that casting to or from generals can be done on any operand as described above in Section 2.4.5.

The format of a unary instruction is: `OP dest = src`. The semantics of this instruction are to perform `OP` on the `src` and store the result in `dest`, and then to increment the PC register. The format of a binary ALU instruction is : `OP dest = src1 src2`. The semantics of this instruction are to perform `src1 OP src2` and store the result in `dest`, and then to increment the pc register.

With unary and binary operations, the destination operand can be one of the source operands, unless type restrictions disallow this. Operands may be either frame slots or registers. Table 2.2 specifies all the arithmetic, logical, and move operators. Table 2.3 shows all the comparison operators. Finally, Table 2.4 describes all of the conversion operators. All of the instructions do the obvious thing with the possible exception of ISPTR. The ISPTR instruction allows us to avoid the use of boxed enumerated types. The instruction considers all addresses less than absolute `0xff` as small integers.

Control Instructions

The control instructions are used to transfer or change the flow of control in the program. The thread control instructions—FORK, SWITCH, and CASE—all enable the transfer of control, which actually takes place later. Similarly, the POST instruction, executed only in inlets, enables a thread and its associated frame for later execution. In other words, the only instructions occurring in a thread that affect the program counter are the STOP and SWAP instructions. All the other thread control instructions enable threads or frames to which control will later be transferred by the above two instructions.

On the other hand, the instructions that affect the flow of control for inlets take effect immediately. For instance, the arrival of a message immediately transfers control to the specified inlet. The FORK instruction used in an inlet to fork another inlet also causes control to be transferred immediately to the destination inlet. Note that the destination of an inlet fork instruction is restricted to an inlet in the same codeblock.

In the code descriptions $inlet$.`synch` and thr.`synch` denote the synchronization slot in the frame that corresponds to the inlet or thread specified. Unless otherwise specified, the PC is incremented after each instruction. The FORK instruction as it appears in a TL0 program is really shorthand for one of four instructions, determined by the context in which it appears (i.e., either in a thread or in an inlet) and whether the destination thread (or inlet) is synchronizing or non-synchronizing. Where FORK appears in the operation column it is shorthand for the correct FORK instruction to be included there.

The thread control instructions are defined in Table 2.5. Basic control flow is implemented with the FORK instruction, which places a ready thread on the LCV. The structure of the LCV, and how the thread is inserted

Instruction	operation
FORK thr	$\mathtt{lcv} \leftarrow \mathtt{lcv} \cup \{ thr \}$
FORK $sthr$	$sthr.\mathtt{synch} \leftarrow sthr.\mathtt{synch} - 1$ **if** $sthr.\mathtt{synch} = 0$ **then** $\mathtt{lcv} \leftarrow \mathtt{lcv} \cup \{ thr \}$ **endif**
SWITCH $cond\ thr_1\ thr_2$	**if** $cond$ **then** FORK thr_1 **else** FORK thr_2 **endif**
CASE $index\ thr_0\ thr_1\ \ldots\ thr_n$	**if** $index \geq 0$ **and** $index \leq n$ **then** FORK thr_{index} **else** ERROR
SET_ENTER thr	$\mathtt{fp.enter} \leftarrow thr$
SET_LEAVE thr	$\mathtt{fp.leave} \leftarrow thr$
STOP	**if** $\mathtt{lcv} \neq \emptyset$ **then** **forsome** $t \in \mathtt{lcv}$: $\mathtt{pc} \leftarrow t$ $\mathtt{lcv} \leftarrow \mathtt{lcv} - \{ t \}$ **else** $\mathtt{pc} \leftarrow \mathtt{fp.leave}$ **endif**

Table 2.5: *Thread-level control instructions.* thr *indicates an unsynchronizing thread,* sthr *a synchronizing thread,* thr_i *either a synchronizing or an unsynchronizing thread.*

Instruction	operation
FORK $inlet$	$\texttt{pc} \leftarrow inlet$
FORK $sinlet$	$sinlet.\texttt{synch} \leftarrow sinlet.\texttt{synch} - 1$ **if** $sinlet.\texttt{synch} = 0$ **then** $\texttt{pc} \leftarrow sinlet$ **else** $\texttt{pc} \leftarrow \texttt{pc} + 1$ **endif**
STOP	$\texttt{pc} \leftarrow \texttt{tpc}$ $\texttt{fp} \leftarrow \texttt{tfp}$

Table 2.6: *Inlet-level control instructions.* inlet *indicates an unsynchronizing inlet while* sinlet *represents a synchronizing one.*

into the structure, are left undetermined by TL0. In the definition for FORK this is reflected in the use of the union operation. It is also reflected by the **forsome** operator in the definition of STOP, which picks any enabled thread from the LCV and transfers control to it. Conditional control is implemented by the SWITCH and CASE instructions. The SWITCH instruction performs one of two forks based on a boolean condition, while the CASE instruction uses an integer index between 1 and n to fork one of n threads. Both of these instructions are defined in terms of FORK.

The SET_ENTER (or SET_LEAVE) instruction sets the `enter` (or `leave`) slot in the frame. The `enter` slot determines the thread that is started when the codeblock is scheduled. (See the SWAP instruction in Table 2.7). The `leave` slot specifies the thread that executes when the quantum ends. This is shown in the definition of the STOP instruction, also in Table 2.5.

The inlet control instructions (see Table 2.6) differ from the thread control instructions in that they each directly alter the flow of control. The inlet versions of the fork instruction are mainly used as jump instructions for code reuse. The inlet STOP is like a return-from-interrupt instruction, which in TL0 is performed by simply restoring the PC and FP from the TPC and TFP, respectively.

The last group of control instructions deal with the scheduling of frames. (See Table 2.7.) The POST instruction, executed only in inlets, enables a thread and if necessary enables the target frame. POST performs the same operation as a thread FORK—i.e., it enables a thread—but it places the enabled thread on the RCV. In addition, if the POST enables the destination thread, it will ensure that the frame is either currently running or placed in the ready queue. The structure of the RCV and the ready queue are left unspecified in TL0, except that all the memory needed for these data structures is contained in the frame itself.

The SWAP instruction switches the current frame to the next ready frame. The manner in which a frame is made ready (by POST) or made the current frame (by SWAP) is dictated by the *policy* argument to the instruction.

While TL0 could explicitly encode the actions that are described in the policy arguments, using a policy argument in the instruction enhances the ease with which we can run experiments in scheduling and the like. The policy is thus a shorthand that tells the translator and runtime system what the instruction should do. For every instruction that has a policy argument a default policy is defined. The default policy encodes the most basic approach to implementing the instruction. In Table 2.8 the two post policies and the swap policy currently defined for TL0 are described. There is always at least one frame on the ready queue—the

Instruction	operation
POST *thr policy*	**if** $\texttt{fp.rcv} = \emptyset$ **then** enable `fp` according to *policy* **endif** $\texttt{fp.rcv} \leftarrow \texttt{fp.rcv} \cup \{ \mathit{thr} \}$
POST *sthr policy*	$\mathit{sthr}.\texttt{synch} \leftarrow \mathit{sthr}.\texttt{synch} - 1$ **if** $\mathit{sthr}.\texttt{synch} = 0$ **then** **if** $\texttt{fp.rcv} = \emptyset$ **then** enable `fp` according to *policy* **endif** $\texttt{fp.rcv} \leftarrow \texttt{fp.rcv} \cup \{ \mathit{thr} \}$ **endif**
SWAP *policy*	`fp` ← a ready frame according to *policy* $\texttt{pc} \leftarrow \texttt{fp.enter}$

Table 2.7: *Frame-level control instructions.*

Instruction	policy	operation
POST	*default*	$\texttt{fp}.\mathit{queue} \leftarrow \texttt{qp}$ $\texttt{qp} \leftarrow \texttt{fp}$
POST	*fifo*	$\texttt{fp}.\mathit{queue} \leftarrow \emptyset$ $\texttt{qp}.\mathit{end}.\mathit{queue} \leftarrow \texttt{fp}$ $\texttt{qp}.\mathit{end} \leftarrow \texttt{fp}$
SWAP	*default*	$\texttt{fp} \leftarrow \texttt{qp}$ $\texttt{qp} \leftarrow \texttt{fp}.\mathit{queue}$

Table 2.8: *Example policies for* POST *and* SWAP.

Instruction	operation
SEND $cont$ <-$data_1$ $data_2$ ... $data_n$	$\texttt{network} \leftarrow \texttt{network} \cup \{(cont, data_1, data_2, \ldots data_n)\}$
SEND $fptr[inlet + offset]$<-$data_1$ ... $data_n$	$\texttt{network} \leftarrow \texttt{network} \cup \{(\langle fptr, inlet + offset \rangle, data_1, , \ldots data_n)\}$
SEND $fptr[inlet/cb + offset]$<-$data_1$ ... $data_n$	$\texttt{network} \leftarrow \texttt{network} \cup \{(\langle fptr, cb.inlet{+}offset \rangle, data_1, \ldots data_n)\}$
RECIEVE $data_1$ $data_2$ ... $data_n$	**if** $\texttt{network} \neq \emptyset$ **then** **forsome** $message \in \texttt{network}$ $\texttt{network} \leftarrow \texttt{network} - \{ message \}$ $\texttt{tpc} \leftarrow \texttt{pc}$ $\texttt{tfp} \leftarrow \texttt{fp}$ $\texttt{pc} \leftarrow message.inlet$ $\texttt{fp} \leftarrow message.\texttt{fp}$ **forall** $(1 \leq i \leq n)$ $\texttt{fp}.data_i \leftarrow message.data_i$ **endif**

Table 2.9: *Message primitives in* TL0.

system-idle frame. The system-idle frame executes a nop and swaps to the next ready frame, which, if no more work has accumulated on the processor, just executes itself again.

Messages Instructions

In TL0, all messages are sent with the SEND instruction as shown in Table 2.9. If the SEND is executed in a thread, it will send the message on the send network. If executed in an inlet, it will send the message on the reply network. (See Section 2.1.5.)

The destination of the message is specified by the continuation in the instruction. In this case, the continuation is an inlet address and a frame pointer. There are several different ways to denote inlet addresses. To specify an inlet value, both an inlet number and the codeblock to which it is relative are required. If the codeblock is not specified explicitly, then the codeblock can be found by referencing the codeblock field of the frame pointer. In either case, a signed integer offset may be added to the inlet descriptor. The offset can change only the relative inlet number for a particular codeblock; it cannot change the codeblock. The more explicit *inlet*/*cb* specification facilitates compiler optimizations by indicating the codeblock of the destination frame at compile time. It is an error if the codeblock as specified at compile time does not agree with the codeblock of the destination frame; in other words, *cb* must equal *fptr.cb*.

The RECEIVE instruction is always the first instruction of the inlet. It is essentially responsible for starting the inlet and saving the data in the message into the target frame.

Instruction	operation
CTRAP $slot\ =\ name\ arg_1\ arg_2\ \ldots\ arg_n$	$slot\ \leftarrow name\ (arg_1\ , arg_2\ , \ldots\ , arg_n\ )$
CTRAP $name\ arg_1\ arg_2\ \ldots\ arg_n$	$name\ (arg_1\ , arg_2\ , \ldots\ , arg_n\ )$
STRAP $cont\ =\ name\ arg_1\ arg_2\ \ldots\ arg_n$	$name\ (cont, arg_1\ , arg_2\ , \ldots\ , arg_n\ )$

Table 2.10: *Foreign Language Interface.*

Foreign Language Interface Instructions

The three instructions defined in Table 2.10 allow TL0 programs to call C functions. The two CTRAP instructions are blocking, strict calls to C routines. The STRAP initiates a split-phase call to a local C function. The function returns any results to the continuation supplied as its first argument. The intended use for the STRAP instruction is to invoke library functions that perform long-latency operations. It also allows user-written C functions to return multiple results.

Heap Access Instructions

The heap instructions presented here are not essential to the TL0 model. Rather they should be viewed as macros that define a standard library implementing the I/M/A-structure semantics presented in Section 2.2. Indeed, all the instructions defined in the rest of this section could be implemented with the previously defined TL0 instructions. In order to facilitate optimizations in the TL0 back end, and for ease of use, definition, and experimentation, we have chosen to include these as part of the basic language.

Each of the heap operations has two parts: the request operation, executed in a thread of the TL0 program, and a service operation, which is part of the TL0 library that implements the specific semantics of the operation.

There are two kinds of data heap objects that can be manipulated by TL0: packed and unpacked arrays. The standard object is an unpacked array of tagged words, each of which can hold an element of any type. The tags do not encode type information, but indicate whether the word contains data (`full`), does not contain data (`empty`), has any deferred readers (`deferred`), or contains an unevaluated thunk (`thunk`). The packed arrays do not have per-element synchronization. Rather, the entire object has a single counter which indicates whether or not all the elements have been stored.

The heap instructions can be divided into *maintenance* instructions, which act on the heap or entire heap objects, and *accessor* instructions, which come in read/write pairs and are used to access elements of the objects. All heap instructions are split-phase. The request phase sends a message to the appropriate destination. In the case of the maintenance instructions, the destination will be a system frame that handles the heap. Accessor instructions use the object itself as the destination of the message.

Heap Maintenance Operations

The request phases of the maintenance instructions are defined in Table 2.11. Each sends a message to the *heapAllocator* system frame. The instruction and policy together determine the inlet that will execute. In each case, TL0 does not determine whether the action carried out by the message occurs only at inlet level

Instruction	operation
IALLOC $inlet = [size]policy$	SEND $heapAllocator[policy] \leftarrow \langle \texttt{fp}, inlet \rangle, size$ where *policy* $\in \{ ialloc - default\}$
IFREE $ptr\ policy$	SEND $heapAllocator[policy] \leftarrow ptr$ where *policy* $\in \{ ifree - default\}$
ICLEAR ptr	SEND $heapAllocator[iclear] \leftarrow ptr$
AALLOC $inlet = [size]policy$	SEND $heapAllocator[policy] \leftarrow \langle \texttt{fp}, inlet \rangle, size$ where *policy* $\in \{ aalloc - default\}$
AFREE $ptr\ policy$	SEND $heapAllocator[policy] \leftarrow ptr$ where *policy* $\in \{ afree - default \}$

Table 2.11: *Heap instructions that manipulate entire objects.*

Inlet	Arguments	Definition
IALLOC.DEFAULT	$cont, size$	$ptr \leftarrow$ inew $(size)$ **forall** $w \in ptr : w.tag \leftarrow \texttt{empty}$ SEND $cont \leftarrow ptr$
IFREE.DEFAULT	ptr	ifree (ptr)
ICLEAR	ptr	**forall** $w \in ptr : w.tag \leftarrow \texttt{empty}$
AALLOC.DEFAULT	$cont, size$	$ptr \leftarrow$ anew $(size)$ $ptr.counter \leftarrow size$ SEND $cont \leftarrow ptr$
AFREE.DEFAULT	ptr	afree (ptr)

Table 2.12: *Definitions of the operations performed by the heapAllocator system frame.*

or if resources need to be acquired at thread level as well. Whether at inlet level or thread level, the sum of the actions performed at the server end is defined in Table 2.12.

Both IALLOC and AALLOC return new heap objects which have been initialized to indicate that all the elements are empty—in the former case by initializing each element to the empty tag, in the latter by setting up a synchronization counter. The inew() and anew() routines are raw memory allocators that allocate the proper sized I-structure and A-structure respectively.

The free instructions, IFREE and AFREE, return the objects to the heap so that they may be reused. Since TL0 does not provide any implicit garbage collection, the compiler must use these instructions in order to free up memory, using ifree() and afree().

The high-level language, or compiler, must ensure that it is safe to execute these instructions. For example, there must be no outstanding messages to structures that are freed. In the case of activation frames, the compiler can guarantee this by keeping track of outstanding messages. Heap objects present a tricky problem as none of the instructions that write to heap objects (e.g. ISTORE) send back an acknowledgment that the operation has completed.

Instruction	operation
IFETCH $cont\ =\ ptr\ [index\ ]$	SEND $ptr[\text{IFETCH}] \leftarrow cont, index$
ISTORE $ptr\ [index\ ] =\ src$	SEND $ptr[\text{ISTORE}] \leftarrow index, src$
ITAKE $cont\ =\ ptr\ [index\ ]$	SEND $ptr[\text{ITAKE}] \leftarrow cont, index$
IPUT $ptr\ [index\ ] =\ src$	SEND $ptr[\text{IPUT}] \leftarrow index, src$
IEXAMINE $cont\ =\ ptr\ [index\ ]$	SEND $ptr[\text{IEXAMINE}] \leftarrow cont, index$
IREAD $cont\ =\ ptr\ [index\ ]$	SEND $ptr[\text{IREAD}] \leftarrow cont, index$
IWRITE $ptr\ [index\ ] =\ src$	SEND $ptr[\text{IWRITE}] \leftarrow index, src$
AREAD $cont\ =\ ptr\ [index\ ]$	SEND $ptr[\text{AREAD}] \leftarrow cont, index$
AWRITE $ptr\ [index\ ] =\ src$	SEND $ptr[\text{AWRITE}] \leftarrow index, src$

Table 2.13: *Definitions of the request halves of the heap element accessor functions.*

The ICLEAR instruction is used to reinitialize an object so that it appears completely empty. For performance reasons, the ICLEAR instruction does not send back an acknowledgment, which is probably a flaw.

Heap Accessor Operations

All of the heap accessor functions are defined as SEND instructions. (See Table 2.13.) They each send a message to a special inlet of the I-structure being accessed. The inlet will perform the action dictated by the semantics of the particular operation.

The I-structure instructions, IFETCH and ISTORE, are described in Table 2.14. The enqueue() routine sets the tag to deferred and enqueues the continuation on the deferred list.

```
enqueue(base, type, cont, index)
{
        base[index].tags ← DEFERRED
        base[index].deferred ← base[index].deferred ∪ { cont }
}
```

If the I-structure has a tag value that is not handled by the routine, then an ERROR is signaled. In TL0, ERROR aborts the program.

One possible implementation of I-structures of size n is a size n array of element structures, each of which has three fields—tags, data, and deferred. The *tags* field holds the tags for the elements, the *data* field, the data, and the *deferred* field the deferred list (if any).

The IFETCH instruction is unique in that it can handle the `thunk` tag. If an element with this tag is fetched, the system frame *thunker* will be called, replacing the I-structure element with a deferred entry. The thunker frame will allocate a new activation frame for the codeblock defined by the old contents of the element, passing in as arguments the I-structure base and index that now contain the deferred request. When the computation is done, it will execute an ISTORE, which will return the result to the original requester.

M-structure operations are described in Table 2.15. The non-synchronizing heap access instructions are

Inlet	Arguments	Definition
IFETCH	$ptr, cont, index$	**case** $ptr[index]$.`tags` `full` : REPLY $cont \leftarrow ptr[index]$.`data` `empty` : `deferred` : enqueue(ptr, IFETCH, $cont$, $index$) `thunk` : $desc \leftarrow ptr[index]$.`data` enqueue($ptr$, IFETCH, $cont$, $index$) SEND $thunker[enqThunk] \leftarrow ptr, index, desc$ **otherwise** : ERROR **endcase**
ISTORE	$ptr, index, src$	**case** $ptr[index].tag\ of$ `empty` : $ptr[index]$.`data` $\leftarrow src$ $ptr[index].tag \leftarrow$ `full` `deferred` : $ptr[index]$.`data` $\leftarrow src$ $ptr[index].tag \leftarrow$ `full` **forall** $d \in ptr[index]$.`deferred` REPLY $d.cont \leftarrow src$ **endfor** $ptr[index]$.`deferred` $\leftarrow \emptyset$ **otherwise** : ERROR **endcase**

Table 2.14: *The definition of the service routines for I-structure operations.*

Inlet	Arguments	Definition
ITAKE	$ptr\ cont\ index$	**case** $ptr[index]$.`tags` `full` : $ptr[index]$.`tags` $\leftarrow$ `empty` REPLY $cont \leftarrow ptr[index]$.`data` `empty` : `deferred` : enqueue(ptr, ITAKE, $cont$, $index$) **otherwise** : ERROR **endcase**
IPUT	$ptr\ index\ src$	**case** $ptr[index]$.`tags` `empty` : $ptr[index]$.`data` $\leftarrow src$ $ptr[index]$.`tags` $\leftarrow$ `full` `deferred` : **forsome** $d \in ptr[index]$.`deferred` : REPLY $d.cont \leftarrow src$ $ptr[index]$.`deferred` $\leftarrow$ $ptr[index]$.`deferred` $- d$ **if** ($ptr[index]$.`deferred` $= \emptyset$) **then** ptr.`tags` $\leftarrow$ `empty` **endif** **otherwise** : ERROR **endcase**

Table 2.15: *M-Structure operations.*

Inlet	Arguments	Definition
IEXAMINE	$ptr\ cont\ index$	REPLY $cont \leftarrow ptr[index]$.data
IREAD	$ptr\ cont\ index$	**if** $ptr[index].tag$ = full REPLY $cont \leftarrow ptr[index]$.data **else** ERROR **endif**
IWRITE	$ptr\ index\ src$	**if** $ptr[index]$.tags = empty **then** $ptr[index]$.data $\leftarrow src$ $ptr[index]$.tags $\leftarrow$ full **else** ERROR **endif**

Table 2.16: *Non-synchronizing heap data access operations for unpacked objects.*

described in Table 2.16.

Table 2.17 describes the read/write operations that can be performed on the packed heap: AREAD and AWRITE. If a read is executed before *all* of the writes have been completed, the read will be deferred. When the last write has been completed then all the reads that have been deferred to any element in the object will be completed.

Frame Allocation

Like heap allocation, frame allocation is split-phase. New activation frames are created in response to the FALLOC instruction. It sends a message, as shown in Table 2.18, to an inlet of the system *frameAllocator* frame, which actually performs the activation. The inlet is determined by the policy argument of the FALLOC instruction.

As can be seen in Table 2.19, both the strict and the default policies allocate an activation frame for the codeblock requested. However, the default policy immediately sends a message to inlet 0 of the new frame, while the strict policy sends the address of the new frame back to the caller. It is a TL0 convention that frames allocated with the default policy will contain a reply in inlet 0 which will return the address of the new frame to the caller. Inlet 0 for frames allocated with the strict policy does not contain the send.

Inlet	Arguments	Definition
AWRITE	$ptr\ cont\ index$	**if** $ptr.counter > 0$ **then** $ptr[index]$.`data` $\leftarrow src$ $ptr.counter \leftarrow ptr.counter - 1$ **if** $ptr.counter = 0$ **and** ptr.`deferred` $\neq \emptyset$ **then** **forall** $d \in ptr$.`deferred` REPLY $d.cont \leftarrow ptr[d.index]$.`data` **endfor** **endif** **else** ERROR **endif**
AREAD	$ptr\ index\ src$	**if** $ptr.counter = 0$ **then** REPLY $cont \leftarrow ptr[index]$.`data` **else** enqueue($ptr$, AREAD, $cont$, $index$) **endif**

Table 2.17: *A-Structure operations.*

Instruction	operation
FALLOC $cont = cblk\ policy$	SEND $frameAllocator[alloc.policy] \leftarrow cont, cblk$ where *policy* $\in \{ default, strict \}$
FINIT	`fp`.$cb \leftarrow codeblock\ descriptor$ `fp.rcv` $\leftarrow \emptyset$
FFREE $ptr\ policy$	SEND $frameAllocator[free.policy] \leftarrow ptr$ where *policy* $\in \{ default \}$

Table 2.18: *Frame Allocation and De-allocation routines.*

Inlet	Arguments	Definition
ALLOC.DEFAULT	$fptr, cont, cblk$	$ptr \leftarrow fnew(cblk.size)$ SEND $ptr[cblk/0] \leftarrow cont$
ALLOC.STRICT	$fptr, cont, cblk$	$ptr \leftarrow fnew(cblk.size)$ SEND $cont \leftarrow ptr$
FREE.DEFAULT	$fptr, ptr$	$ffree(ptr, ptr.cb.size)$

Table 2.19: *The frame allocation policies as determined by the inlets defined for the* frameAllocator.

Instruction	operation
KALLOC $inlet = cblk\ rows$	SEND $frameAllocator[kalloc] \leftarrow \mathtt{fp}, inlet, cblk, rows$
KFREE ptr	SEND $frameAllocator[kfree] \leftarrow ptr$
KLINK $next, prev, offset, rows, thr$	**if** $offset \neq rows$ **then** $next \leftarrow nextFrame()$ **else** $next \leftarrow firstFrame()$ **endif** **if** $offset \neq 0$ **then** $prev \leftarrow prevFrame()$ **else** $next \leftarrow lastFrame()$ **endif** **if** $(\mathtt{fp} \neq lastFrame())$ $POST\ thr$ **endif**

Table 2.20: *Instructions that support the use of K-bounded loops.*

Chapter 3

TLC: An Implementation of TL0

This chapter describes a portable implementation of TL0 that can run on both commodity serial and parallel machines. This portability is achieved through a two-level compilation process. First, the TL0 code is translated into TLC; then, TLC is compiled by the host C compiler using a set of machine-specific macros and libraries that implement the lowest-level functionality for the particular processor. In this way, the machine-dependent features of TL0 have been abstracted out of the generated code in the form of macros and routines. In porting TL0 to a new machine only the macros and routines need be altered. The cost of this portability is some decreased efficiency for the instructions most unique to TL0. However, in addition to offering portability, this compilation process facilitates experimentation with different implementation policies and ideas. At the end of this chapter a more efficient, but less portable, system is described using GCC-specific optimizations

The first two sections in this chapter describe TLC without describing the macros, typedefs, etc. used to implement TLC on a particular machine. Section 3.1 begins by describing the general strategy used in the translation process and then proceeds to describe the translation of the storage model and the execution model as defined in Chapter 2. Section 3.2 describes the translations for each of the instructions, except the heap instructions which are described in Section 3.3. The implementations of the TLC macros for control are described in Section 3.4.

The rest of the chapter outlines the different implementations of the TLC macros presented in preceding sections. The control macros are described in Section 3.4, the uniprocessor implementations in Section 3.5, and the parallel processor implementations in Section 3.6. Finally, the GCC implementation is described in Section 3.7.

3.1 The Structure of TLC

3.1.1 General Strategy

The general scheme strikes a compromise between the control-flow model of TL0 and the stack-based structured control flow of C. The idea is to group the threads of a codeblock in a single C function, to transform inlets into functions, and to translate TL0 instructions into C statements.

There is no natural mapping, in terms of the execution model, from threads to anything larger than a group of statements. Thus, an artificial mapping must be performed which warps the natural TL0 flow of control.

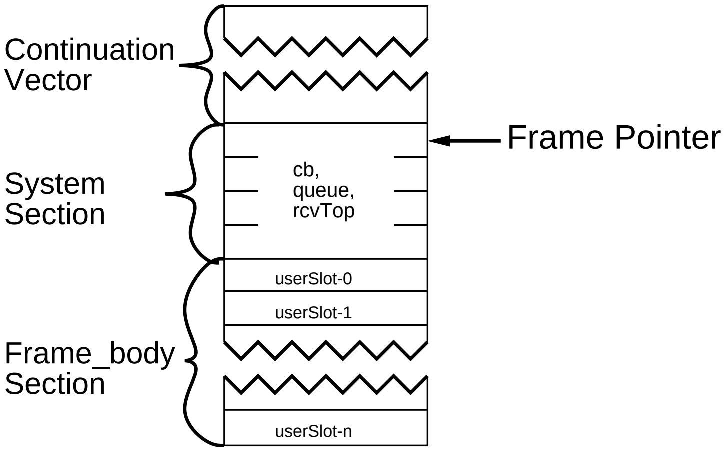


Figure 3.1: *A schematic view of a frame definition. Independent of the implementation, every system section must have at least the fields* `cb`, `Queue`, *and* `rcvTop` *defined.*

This mapping is necessary for two reasons. First, many C compilers cannot handle a `switch` statement with more than a few hundred `case` statements. Second, it allows for separate compilation. To keep the resulting cost as low as possible the threads of a codeblock are collected together into a `switch` statement. Each arm of the `switch` is a separate thread. The switch is then embedded in a function, called the thread-function, which contains code that helps to schedule the codeblock. Thus, instead of the last thread invoking the next codeblock, the thread-function returns to a master loop which schedules the next ready codeblock. This adds some call/return overhead, but limits it as much as possible. Since inlets behave like interrupt routines, it is natural to map each inlet into a separate function. Control is transferred to the inlet, it executes, and then it returns to the location of the caller. Finally, each TL0 instruction is translated into a single C statement.

3.1.2 The Activation Frame

At the heart of the TL0 translation process is the activation frame. In addition to its duty as the data repository for a function activation, it is used for both frame and thread scheduling. Every frame has three sections: system, frame-body, and continuation vector. (See Figure 3.1.) The system section defines all the variables needed by the runtime system. It occupies the first few words immediately following the frame pointer. The frame-body section contains the local data repository for a function activation. The continuation vector, which grows downward in memory (away from the frame pointer), contains pointers to the enabled threads for the frame.

Every codeblock defines an activation frame layout in the `FRAME_BODY` section of the TL0 file. This is translated into a TLC structure definition and a series of constants in the codeblock descriptor. The structure definition defines both system variables common to every activation frame and the frame slots defined in the `FRAME_BODY` section of the codeblock definition.

During execution of the program, the frame pointer, represented by the variable `fp`, is passed into the codeblock's inlets or thread function. It is given the type of the frame structure created for the codeblock. In this way, individual frame slots accessed in the functions take the form of field dereferences, i.e., FP-relative addressing. Finally, since every frame begins with the same system section, the runtime routines can access the system fields by casting the `fp` to the generic type `framePtr`. (See Section 3.1.8)

The System Section

The system variables start at offset 0 and are identical for every frame definition. They define the state needed by the runtime system to schedule and manage frames. The system section is implemented by the TLC macro `SYSTEM_SECTION`.

The system variables are used for frame scheduling, interframe communication, debugging, and statistics collection. While the exact layout and contents of the system section depend on the particular implementation of TLC, every implementation defines at least three system slots: `cb`, `Queue`, and `rcvTop`. The `cb` field is a pointer to the codeblock description that defines the codeblock for the frame. This is used to locate the threads and inlets for the frame. The `Queue` slot is used to link the frame onto the appropriate ready or running queue. The last essential system slot is `rcvTop`, which points to the top of the RCV.

Currently, only one enter and one leave thread are defined for each codeblock. If, however, multiple enter or leave threads were defined in a codeblock, then the current enter (and leave) thread would be recorded in the system frame.

Depending on the implementation, the system frame also contains either an RCV “sentinel” thread or a flag indicating whether the frame is currently ready or idle. (See Section 3.4.)

In addition to the above system variables, which are needed to activate and deactivate the frame, debugging and statistical information are often included. For example, a unique frame id and counters to keep track of CV size, etc. can be included.

The Frame Body

After the system variables every frame slot defined in the `FRAME_BODY` section of the codeblock definition is translated into a field of the structure definition. In inlets and threads, variables defined in the frame activation are accessed as structure members.

The Continuation Vector

A continuation vector capable of containing the number of continuations specified in the `FRAME_BODY` statement is also part of each activation frame, but is not part of the structure definition. Instead, its size is stored in the codeblock descriptor where it can be accessed at frame creation time. The size of the entire structure is also stored in the codeblock descriptor.

The continuation vector, used to hold the threads in a frame that are ready to execute, is implemented as a stack. For reasons of efficiency there are two continuation vectors, the RCV and the LCV. The RCV is used to hold threads that are enabled by the POST instruction. The LCV is used to hold threads that are enabled by FORK or SWITCH instructions.

The main difference between these two continuation vectors is that the top of the RCV must be addressed by a pointer that is kept in the frame for which the threads are enabled. The top of the RCV is the frame slot `rcvTop`. (See above.) The LCV can, instead, be addressed by a local variable `lcvTop`, which represents the LCVPTR register of TAM.

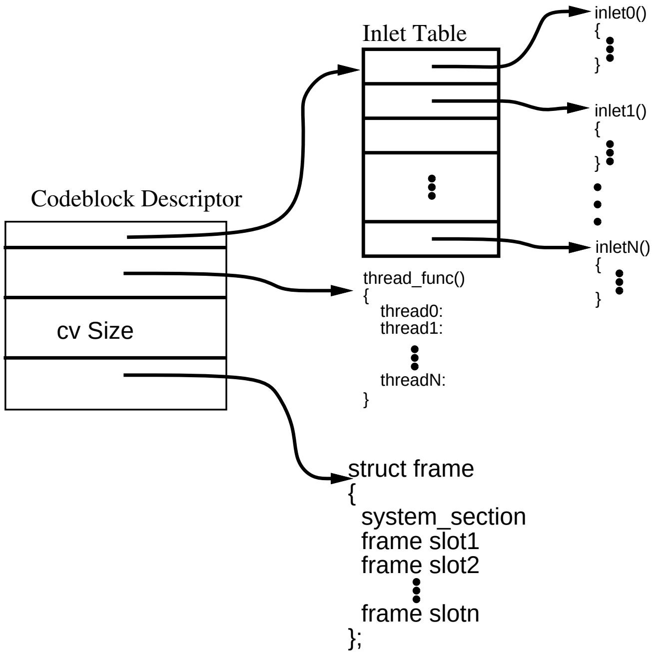


Figure 3.2: *The codeblock descriptor glues together the pieces that make up a codeblock—inlets, threads, and the frame definition.*

3.1.3 Main Scheduling Loop

One of TL0's distinctive features is the two-level scheduling hierarchy. The frame level scheduling is handled in TLC by a frame dispatch loop.

```
localFramePtr        fp;
extern localFramePtr Queue;

while(1)
{
    fp = Queue;
    Queue = Queue->next;
    fp->cb->threadFunc(fp);
}
```

In this loop, `fp` is the frame pointer register and `Queue` is the global ready queue pointer. The function invoked is the thread function associated with the frame's codeblock descriptor field. The thread function takes one argument, a pointer to the current frame, i.e. the activation frame to which the function is relative. Frames are added to the ready queue by POST instructions from within inlets. The thread function exits when the frame no longer contains any enabled threads.

The program is started by having a system frame send the command-line arguments to the main codeblock. When the main codeblock sends back its last argument, the system frame is finally scheduled and exits the program.

```
thread_func(framePtr fp)
{
   lcvDeclaration
   ThreadPointer nextThread;
   thread registers declared here

   rcvInit();

   execute enter thread

   while(1) {
     nextThread = lcvPop();
     switch(nextThread) {
     case 1: label_1:
        code for thread 1 goes here
       break;
     case 2: label_2:
        code for thread 2 goes here
       break;
     ⋮
     case n: label_n:
        code for thread n goes here
       break;
     default:
        code to execute for an empty lcv goes here
        includes code to execute the leave thread
     }
   }
}
```

Figure 3.3: *A prototype for a thread function showing the main thread scheduling loop and n threads.*

3.1.4 Codeblocks and Their Components

As shown in Figure 3.2, every TL0 codeblock is translated into a collection of inlet functions, a thread function, an activation layout, and a codeblock descriptor which ties these all together. During normal execution of a TL0 program, the codeblock descriptor is used for three purposes. First, when a new codeblock is invoked, the codeblock descriptor is used to allocate the appropriately sized frame. (See Section 3.2.6.) Second, when the frame is scheduled to run, the thread function is located by indirection through the descriptor. Finally, inlet arithmetic (described in Section 3.1.6) uses the inlet table, also reached through the descriptor.

In many ways the codeblock descriptor can be likened to a class object in an object-oriented language such as Smalltalk. Similarly, the activation frame is analogous to an instance object, while the inlet table and the thread function pointer in the codeblock descriptor are like the method dispatch tables. Keeping this analogy in mind should make understanding the frame allocator and inlet/thread dispatch easier.

3.1.5 Threads

Figure 3.3 shows the overall structure of the thread function. All the threads of a codeblock are grouped together into a single `switch` statement. Every thread becomes one arm of the `switch`. The `switch` statement itself is embedded in a `while` loop that never terminates. Preceding the `while` loop are the declarations for all the thread registers declared in the codeblock, a macro to initialize the continuation

```
I_codeblock_n(localFramePtr ifp, arg1, arg2, ... argn)
{
    inlet registers declared here

    ifp->arg1 = arg1;
    ifp->arg2 = arg2;
    ⋮
    ifp->argn = argn;

    body of inlet
}
```

Figure 3.4: *Translation schema for inlet n in the codeblock named* codeblock.

vector, and code to execute the enter thread.

When control enters the thread function the continuation vector is initialized. The local continuation vector can have different representations depending on the compilation strategy employed. The macros `lcvDeclaration` and `rcvInit` implement the initialization code.

Once the continuation vector is initialized, the enter thread is executed. If the enter thread is known at compile time (as it always is in the current version of TL0), then either the code for that thread is inlined before the `while` loop, or a direct jump is made to the thread. After the enter thread executes, the `while` loop is entered and the `switch` pops a thread identifier from the LCV, and then executes the identified thread. When no more threads are in the LCV, the sentinel thread identifier is popped from the LCV. The sentinel thread is created by the compiler to execute the leave thread when there are no more enabled threads in the frame. It is up to the leave thread to eventually execute a `SWAP` instruction returning control to the main dispatch loop.

3.1.6 Inlets

Each inlet of a codeblock is compiled into a separate function. Figure 3.4 shows the default translation schema used for inlets. The name of an inlet is created by prefixing the inlet number with the name of the codeblock and the text `I_`. Thus, inlet n of the codeblock `cblk` has the name: `I_cblk_`n. The formal parameters of the function include the inlet frame pointer and one parameter for each argument of the RECEIVE. In order to avoid confusion, we will use the variable `ifp` to represent the frame pointer used in inlet functions. If any inlet registers are declared for the codeblock, they are declared in the function as well. The function begins with a series of assignments, one for each variable received by the inlet, transferring the arguments to the frame. After the receives, any instructions in the inlet are included in the inlet function.

All the inlets are included in an inlet function table, which forms the basis for the inlet arithmetic used in SEND instructions. The inlet data type in TL0 becomes an `inlet` type in TLC, and it is a pointer not to the inlet function itself, but rather to the inlet function's entry in the table. Inlet arithmetic is just adding or subtracting integers from inlets, which corresponds to changing the pointer into the inlet table.

The additional complexity in translating inlets arises from the case where all the arguments to an inlet cannot fit into a single message. This is the one case where the underlying architecture affects the TLC code and macros are not sufficient to handle machine specialization.

If the total number of words for all the arguments in the receive list exceeds a parameter specified during compilation, then the inlet is divided into enough supplementary inlets so that no inlets require more

```
supplement_inlet_m1(ifp, arg0, arg1, ... argn)
{
   ifp->arg0 = arg0;
   ifp->arg1 = arg1;
   ⋮
   ifp->argn = argn;

   codeblock_inlet_m(ifp);
}

codeblock_inlet_m(ifp)
{
   inlet registers declared here

   if (--ifp->supplementInletSynchm) return;
   ifp->supplementInletSynchm = 2;

   body of inlet
}
```

Figure 3.5: *Example supplement inlet and its corresponding body inlet. The body inlet resets the compiler-generated synchronization variable back to two—implying that there are two supplemental inlets that need to be run before the body can run.*

arguments than the maximum number of words in a message. As shown in Figure 3.5, each of the supplementary inlets has a receive section and forks a new body-inlet, which executes the body of the original inlet. The body-inlet becomes synchronizing, so that it is executed only after all the supplementary inlets have executed. This scheme allows for message reordering and makes only the assumption that the synchronization variable has been initialized before any of the inlets execute. In the case where the inlet being executed is inlet 0, the synchronization variable is set by the frame allocator. (See Section 3.2.6.)

3.1.7 Registers

One of the compromises made in TLC is that explicit control of the processor register set is lost. Instead, TLC relies almost entirely on the analysis done by the C compiler to place heavily used variables in registers.[1] That said, TLC does make it easy for the C compiler to do the right thing by declaring user thread (and inlet) registers as automatic variables in the thread (and inlet) function(s). The dedicated TL0 registers become global variables, arguments, or local variables, as described below.

Dedicated Registers

fp (frame pointer) is passed into the thread function (as variable `fp`) or the inlet function (as variable `ifp`) and all addressing of frame-relative slots is achieved through structure dereference.

tfp (thread frame pointer) does not exist explicitly. If an inlet is called while a thread is executing, the thread's FP is saved on the stack. The C stack serves as the "register" in which the FP is saved when an inlet is running.

[1]As we show later, some variables can be forced to reside in particular registers by certain C compilers, like GCC.

data type	TL0 abbrev	TLC typedef	description
Boolean	.b	bool	Booleans are 1-bit values that are mapped onto C integers. They obey the C convention that false is zero and true is nonzero.
Character	.c	char	A character.
Integer	.i	int	An integer.
Float	.f	double	A double-precision floating point number.
Synchronization counter	.s	synchCounter	A synchronization counter.
Inlet descriptor	.j	inlet	A pointer to an entry in an inlet table. The inlet is called by indirecting though the variable.
Thread descriptor	.t	threadPtr	A integer that denotes a thread in the current codeblock.
Frame pointer	.pf	framePtr	A pointer to a frame activation. A `struct frame` defines the system part of the frame, which is the same for every codeblock.
Codeblock pointer	.pc	cbDesc *	See Figure 3.2.
Structure pointer	.ps	structPtr	A pointer to a heap object
General	.g	general	This type is big enough to hold any of the types listed in this table.

Table 3.1: *Each* TL0 *data type is translated to a C data type as defined in this table.*

pc (program counter) does not exist explicitly.

tpc (thread program counter) does not exist explicitly. When an inlet is invoked, the C compiler saves the current program counter as it does for any function call.

qp (ready queue pointer) is a global variable: `Queue`.

lcvptr (local continuation vector) is implemented as a macro: `lcvPointer`. It is only valid when a thread function is running.

User Registers

Every user register specified in an inlet or thread is implemented as an automatic local variable in the corresponding function.

3.1.8 Data Types

Table 3.1 lists the TL0 data types and the TLC data types to which they are translated. In addition to the TL0-defined types, TLC introduces two new pointer types: `localFramePtr` and `localStructurePtr`. These two types are pointers to frames and structures, respectively, on the local processor. For a machine with an explicit global address space the `localFramePtr` (or `localStructurePtr`) is the same type as the `framePtr` (or `structPtr`). On distributed memory machines they are quite different. (See Section 3.6.1.)

The `localFramePtr` is defined as

```
typedef struct frame* localFramePtr;

struct frame
{
   SYSTEM_SECTION
};
```

The type `general` is defined as a union, so that any of the TLC types can fit into it.

```
typedef union
{
    int i;
    double f;
    char c;
    int b;
    framePtr pf;
    cbDesc *pc;
    inlet j;
    structPtr ps;
    deferred *todo;
    long long asArg;
} general;
```

The `asArg` member is used to guarantee that when a `general` is passed as an argument to a C function, it is passed in a register. Thus, it must be as large as the largest type. In many C compilers, structures are passed by invisible reference, which won't work when sending or receiving messages across the network. The definition above reflects the GCC implementation; other C compilers would have appropriate definitions.

3.2 Instructions

Now that the basic mechanisms have been laid out, we describe how each of the TL0 instructions is translated into TLC. One of the goals of TL0 is to expose as much of the underlying scheduling mechanism as possible to the compiler. As a result, the runtime system and the instruction translations are intimately connected, and are discussed together in the following sections.

3.2.1 ALU

The translation of the ALU instructions from TL0 to TLC is straightforward, with the possible exception of `ISPTR`, which checks to see if its argument is a pointer. The operators, as described in Section 2.4.6 and Tables 2.2, 2.3, and 2.4, are each translated into C in the obvious way. Table 3.2 shows the translation into TLC.

The ISPTR instruction uses the macro `adrPart`, which returns the address portion of its argument. This allows TL0 pointers to take on the multipart structure that is required to implement global pointers on distributed memory machines such as the CM-5. (See Section 3.6.1.)

<table>
<tr><th>Instruction</th><th>TLC code</th></tr>
<tr><td>MOVE dst = src1</td><td><code>dst = src1;</code></td></tr>
<tr><td>ABS dst = src1</td><td><code>dst = abs(src1);</code></td></tr>
<tr><td>ADD dst = src1 src2</td><td><code>dst = src1 + src2;</code></td></tr>
<tr><td>SUB dst = src1 src2</td><td><code>dst = src1 - src2;</code></td></tr>
<tr><td>MUL dst = src1 src2</td><td><code>dst = src1 * src2;</code></td></tr>
<tr><td>DIV dst = src1 src2</td><td><code>dst = src1 / src2;</code></td></tr>
<tr><td>MOD dst = src1 src2</td><td><code>dst = src1 % src2;</code></td></tr>
<tr><td>MAX dst = src1 src2</td><td><code>dst = src1 > src2 ? src1 : src2;</code></td></tr>
<tr><td>MIN dst = src1 src2</td><td><code>dst = src1 < src2 ? src1 : src2;</code></td></tr>
<tr><td>NOT dst = src1</td><td><code>dst = !(src1);</code></td></tr>
<tr><td>AND dst = src1 src2</td><td><code>dst = src1 && src2;</code></td></tr>
<tr><td>OR dst = src1 src2</td><td><code>dst = src1 || src2;</code></td></tr>
<tr><td>XOR dst = src1 src2</td><td><code>dst = src1 != src2;</code></td></tr>
<tr><td>BITNOT dst = src1</td><td><code>dst = ~(src1);</code></td></tr>
<tr><td>BITAND dst = src1 src2</td><td><code>dst = src1 & src2;</code></td></tr>
<tr><td>BITOR dst = src1 src2</td><td><code>dst = src1 | src2;</code></td></tr>
<tr><td>BITXOR dst = src1 src2</td><td><code>dst = src1 ^ src2;</code></td></tr>
</table>

Translations for the arithmetic and logical operations

<table>
<tr><td>EQ dst = src1 src2</td><td><code>dst = (src1 == src2);</code></td></tr>
<tr><td>NE dst = src1 src2</td><td><code>dst = (src1 != src2);</code></td></tr>
<tr><td>LT dst = src1 src2</td><td><code>dst = (src1 < src2);</code></td></tr>
<tr><td>LE dst = src1 src2</td><td><code>dst = (src1 <= src2);</code></td></tr>
<tr><td>GT dst = src1 src2</td><td><code>dst = (src1 > src2);</code></td></tr>
<tr><td>GE dst = src1 src2</td><td><code>dst = (src1 >= src2);</code></td></tr>
<tr><td>ISPTR dst = src1</td><td><code>dst = ((unsigned int) adrPart(src1) >= 256);</code></td></tr>
</table>

Translations for the comparison operators.

<table>
<tr><td>ITOF dst = src</td><td><code>dst = src;</code></td></tr>
<tr><td>ITOC dst = src</td><td><code>dst = src;</code></td></tr>
<tr><td>ITOP dst = src</td><td><code>dst = asAdr(src);</code></td></tr>
<tr><td>FTOI dst = src</td><td><code>dst = src;</code></td></tr>
<tr><td>CTOI dst = src</td><td><code>dst = (int)(src) & 0xff;</code></td></tr>
<tr><td>PTOI dst = src</td><td><code>dst = (int)adrPart(src);</code></td></tr>
</table>

Translations for the conversion operators.

Table 3.2: *C code produced for the* TL0 *ALU instructions.*

3.2.2 Control

Although the specification of control instructions is entirely in terms of how they affect the frame queue, the RCV, and the LCV, the compilation process specializes the instructions to avoid the use of these data structures. For instance, every effort is made to turn a FORK into a direct jump, eliding the STOP at the end of the thread. In this section, we describe the TLC macros and data structures that implement the control instructions described in Section 2.4.6. We first describe the simple thread-control instructions (FORK and STOP), proceeding to the conditional forks (SWITCH and CASE) and then frame-based scheduling instructions (POST and SWAP). The inlet-control instructions FORK and STOP are explained last. The thread control instructions—STOP, FORK, SWITCH, and CASE—have different expansions depending on their location in the thread. The frame-based control instructions have different execution paths optimized for the state of the machine at the time of their execution.

The FORK instruction as defined in Table 2.5 only enables threads. Control is actually transferred by the STOP at the end of the thread. In TLC, the STOP instruction is translated into a break statement.

STOP ⟹ `break;`

Control is then transferred to the top of the scheduling loop, which pops off the next enabled thread, finally dispatching control to the correct arm of the switch. (See Figure 3.3.) In the general case, the non-synchronizing FORK itself is translated as follows.

FORK *thr* ⟹ `lcvPush(`*thr*`);`

If the FORK is synchronizing, the synchronization counter associated with the thread (represented by *sthr*`.synch` in the code examples) is first decremented. If the counter has reached zero, then the thread is enabled. The resulting code is

FORK *sthr* ⟹ `if (!(--fp->`*sthr*`.synch)) lcvPush(`*sthr*`);`

In TLC we have chosen to implement the LCV as a stack. In a straightforward implementation, a FORK at the end of a thread pushes a thread number, which is popped off by the immediately following STOP. Obviously, this push/pop pair can be eliminated in favor of a faster `goto` instruction. Thus, FORKs at the end of a thread are combined with the STOP into a single instruction. We expect the peephole optimizer of the C compiler to eliminate the `goto` if the destination thread is the subsequent one. The resulting translations are

```
FORK thr    ⟹   goto label_thr;
STOP

FORK sthr   ⟹   if (!(--fp->sthr.synch))
STOP                goto label_sthr;
                 else
                    break;
```

Since the compiler makes no assumptions about the structure of the LCV we know that it is safe to reorder the forks in a thread. Often, it can be advantageous for the back end to do so. For instance, since we cannot be sure that a synchronizing fork will succeed, we perform instruction reordering whenever possible to make the last fork in a thread non-synchronizing.

The translations for the conditional fork instructions are based on the translations of the basic fork instructions above. The SWITCH instruction turns into a branch where each arm is translated as if it were a basic fork.

```
SWITCH cond thr1 thr2   ⟹   if cond {
                                  code for FORK thr1
                              } else {
                                  code for FORK thr2
                              }
```

Likewise, the CASE instruction wraps some control structure around translations of each fork. In this case the control structure is a `switch` statement.

CASE *index* *thr1* *thr2* ... *thrn* $\Longrightarrow$
```
switch ( index) {
case 1:
      code for FORK thr1
      break;
case 2:
      code for FORK thr2
      break;
⋮
case n:
      code for FORK thrn
      break;
}
```

The frame-based scheduling instructions each take a policy argument. As discussed earlier, the policy argument is a shorthand that encodes the intent of the front end instead of using an explicit series of TL0 instructions to carry out the desired action. The POST instruction currently allows for three policies: default, fifo, and lifo. Currently, the default policy is the lifo policy—it treats the ready queue as a stack and enqueues ready frames on the front of the queue. The fifo policy enqueues ready frames on the back of the ready queue. The details of the POST instruction are all encoded in the appropriate TLC macro.

POST *thr* *policy* $\Longrightarrow$ `pushPost`*policy*`(` *thr*`);`

POST *sthr* *policy* $\Longrightarrow$
```
if (!(fp-> sthr.synch))
  pushPost policy( thr);
else
  break;
```

The SWAP instruction also takes a policy argument and is translated into a single TLC macro.

SWAP *policy* $\Longrightarrow$ *policy*`_swap();`

The inlet control instructions, FORK and STOP, both cause control to be transferred immediately. The FORK is translated as a call to the destination inlet. Unlike the thread FORK, the synchronization code is placed at the destination instead of at the FORK. (See Section 3.1.6 and Figure 3.5.)

FORK *inlet* $\Longrightarrow$ *inletName*`(ifp);`

The fork, if there is one, is always the penultimate instruction in an inlet. If the synchronization fails, the destination inlet returns to the caller, and then the caller executes a STOP, which is just a return statement.

STOP $\Longrightarrow$ `return;`

3.2.3 Messages

The send and reply instructions in TL0 are mapped to analogous code sequences in TLC. The sequences include a start instruction, a possibly empty sequence of conversion instructions, and finally the send (or reply) macro itself. This sequence of instructions allows TLC to run on a wide variety of message layers. The `startSend(cnt, len)` macro is used to indicate the number and size of the arguments being sent. On some systems this macro reserves a message buffer, on others it does nothing. For example, the macro reserves a buffer for the nCube-2, but does nothing for the CM-5.

If the arguments in the send need to be converted into `generals`, then each one is assigned to a general `argReg`. Within the send macro, the `argReg` is used instead of the original variable.

SEND *frame* [*inletExpr*] <- arg_1 arg_2 ... arg_n $\Longrightarrow$ `startSend(n,` *number_of_words*`);`
conversion to generals
`send`n`args(` *inletExpr*`,` n`,` *frame*`,` *arglist*`)`

Translations are used when an argument of a specific type is cast to a general type, e.g. `arg.i.g`, which sends the integer argument `arg` as a general. Each argument that is translated is assigned to a `general` argument register, which is then used in the send macro. The argument registers, `argReg(`i`)`, are macros that are implemented to reflect the structure of a `general` on the target machine. For instance,

```
SEND frame.pf[1.i/CB.pc] <- anint.i arg.i.g  ==>

          startSend(2, sizeof(int)+sizeof(general));
          {
            general argReg(1);
            argReg(1).i = fp->arg;
            send2args(I_CB_1, 2, fp->frame, fp->anint, argReg(1).asArg);
          }
```

In this example, we see how the inlet expression in the TL0 code is translated into the name of an inlet routine. In this case, the codeblock for the inlet is known at compile time and the address of the C routine for the inlet can be directly included. In the case where the codeblock is not known at compile time, the codeblock is determined by the frame part of the continuation. There are four ways that an inlet continuation can be written in TL0. The optimal one uses the `/codeblock` notation. When the codeblock is known at compile time the actual name of the inlet function is used.

`frame.pf[num.i/codeblock.pc]` $\Longrightarrow$ `I_` *codeblock*`_` *num*

The second specifies the inlet as an integer relative to a frame. The codeblock pointer from the frame is used to find the base codeblock.

`frame.pf[num.i]` $\Longrightarrow$ `*(frame->cb[num])`

If an inlet specifier has type `.j`, then it is a pointer into the inlet table. Thus, the inlet routine can be accessed with the C indirection operator. The last two representations show how this is translated.

```
frame.pf[inlet.j]        ==>  *(inlet)
frame.pf[inlet.j+num.i]  ==>  *(inlet+num)
```

3.2.4 Foreign Language Interface Instructions

Because a TLC program is itself a C program, the interoperation instructions are handled by simply translating the name operand and argument operands into a C function call of the same name. The CTRAP instructions are thus simply

CTRAP *slot* = *name* arg_1 arg_2 ... arg_n $\Longrightarrow$ `fp.` *slot* = *name*`(` arg_1, arg_2, ... arg_n`);`
CTRAP *name* arg_1 arg_2 ... arg_n $\Longrightarrow$ *name*`(` arg_1, arg_2, ... arg_n`);`

User functions that are called via the ctrap mechanism must match the number and type of arguments used in the CTRAP instruction. If the first of the two listed above is used, then it must return a value of the same type as *slot*. The second function must have a return type of `void`.

The STRAP instruction is translated similarly, except that an inlet continuation is passed to the function so that it may return the results to the inlet specified.[2] The continuation is passed as the first two arguments.

[2]The standard TLC library includes I/O and other operating system functions. On parallel machines, these functions require the involvement of the "host" processor and are thus long-latency operations. The compiler uses STRAPs to invoke these functions.

The FP of the current function is passed as argument one. The name of the inlet is passed as the second argument. Assuming the STRAP is found in *codeblock*, the translation is:

STRAP *num* *name* arg_1 arg_2 ... arg_n $\Longrightarrow$

name`(asGlobalFP(fp), I_`*codeblock*`_`*num*`,` arg_1`,` arg_2`,` ... arg_n`);`

The associated user-supplied function for the above STRAP is as follows.

```
typedef void (*cbPtr)();
void name(framePtr f, cbPtr ret, type1 arg1, ... typen argn)
{
     user supplied code
     sendnargs(ret, size, f, n results go here);
}
```

Where $type_n$ is the C type for argument n and *size* is the size (in `sizeof(char)` units) of the n arguments being sent to the inlet specified by `ret`.

3.2.5 Heap

The heap instructions are discussed along with the implementation of the heap objects in Section 3.3.

3.2.6 Frame Allocation

The Falloc Instruction

The FALLOC instruction initiates the split-phase operation of frame creation. In TLC, it is implemented as a call to a library function that creates the new frame and sends a message either to the new frame or back to the caller, depending on the policy argument. We first describe the default (i.e. non-strict) policy and then the strict policy.

The non-strict protocol for frame allocation causes a new frame to be allocated and initialized. A message containing the return continuation is then sent to the new frame at inlet 0. Inlet 0 terminates with a send back to the caller frame containing the new frame's address. This two-way link allows the caller to deposit arguments in the new frame and allows the callee to send back results.

The most general FALLOC instruction becomes

FALLOC *num*`.j=` *cblk* `default` $\Longrightarrow$ `falloc_default00(fp,&(`*thiscblk*`_inlets[`*num*`]),` *cblk*`);`

The second argument is an inlet offset relative to the current codeblock into a variable of type `inlet`, i.e., a pointer into the inlet dispatch table. If the codeblock being invoked can be determined at compile time, then the more efficient `falloc_default11` is used instead. This more specialized allocator takes a fourth argument, the address of inlet 0 for the new routine. Thus we have

FALLOC *num*`.j =` *cblk* `default` $\Longrightarrow$

`falloc_default11(fp, &(`*thiscblk*`_inlets[`*num*`]),` *cblk*`, I_`*cblk*`_0);`

The only difference between the two routines is how they resolve the address of the callee's inlet 0 function. Both routines allocate and initialize the new frame's storage and then invoke inlet 0. The routines themselves are described in Section 3.5.2.

The strict FALLOC is translated similarly except that the names of the library routines are `falloc_strict00` and `falloc_strict11` respectively. In addition, since the strict protocol requires the memory allocator to send back the child frame's address (instead of the child performing the send), a further optimization—inlet inlining—is enabled.

The acknowledge inlet can be inlined when the acknowledge inlet is a constant, the allocation routine is run on the same processor as the caller, and the falloc routine returns the child frame pointer directly. The acknowledge inlet has the following form.

```
inlet x
   receive child.pf
   post y.t
   stop
```

The falloc puts the child frame pointer into `child` and forks thread *y*. Assuming the above inlet x and that `child.pf` is a frame slot (which must be the case since it appears in an inlet), the third and most optimal translation for a strict FALLOC is

```
FALLOC x.j = cblk strict  ==>  fp->child = falloc_strict12(cblk);
                               translation for fork y.t
```

Init and Free

FINIT is used by the child to initialize the frame. For frames allocated by `falloc_default`, FINIT is turned into a nop, since the default allocator also initializes the frame. For strict calls the FINIT instruction serves two purposes. First, it serves as a synchronization variable in the case where inlet 0 is divided into secondary inlets. Second, it initializes the `cb` field of the frame structure.

Since strict frames can take many arguments, inlet 0 may be partitioned into secondary inlets. (See Figure 3.5.) Due to possible message reordering, a flag is needed in the frame that indicates that the frame is uninitialized, so that the inlet 0 synchronization variable can be set properly. This is handled by setting the `cb` field of the frame to `NULL` in the allocator. If inlet 0 is broken into secondary inlets, then before the synchronization variable is decremented the `cb` is checked. If it is `NULL`, the frame is initialized, which also initializes the secondary inlet synchronization counter.

So, for non-strict frames we have

```
FINIT  ==>  /* do nothing */
```

For strict frames where inlet 0 handles all the incoming arguments FINIT becomes

```
FINIT  ==>  ifp->cb = pointer to the codeblock descriptor;
```

And, finally, for strict frames in which inlet 0 has been divided into supplementary inlets, we have

```
FINIT  ==>  if (ifp->cb == NULL) {
                 ifp->cb = pointer to the codeblock descriptor;
                 ifp->inlet0.synch = number of supplementary inlets;
            }
```

FFREE is simply translated into a library call that puts the frame on the free list.

```
FFREE ptr.pf policy  ==>  ffree_policy(ptr);
```

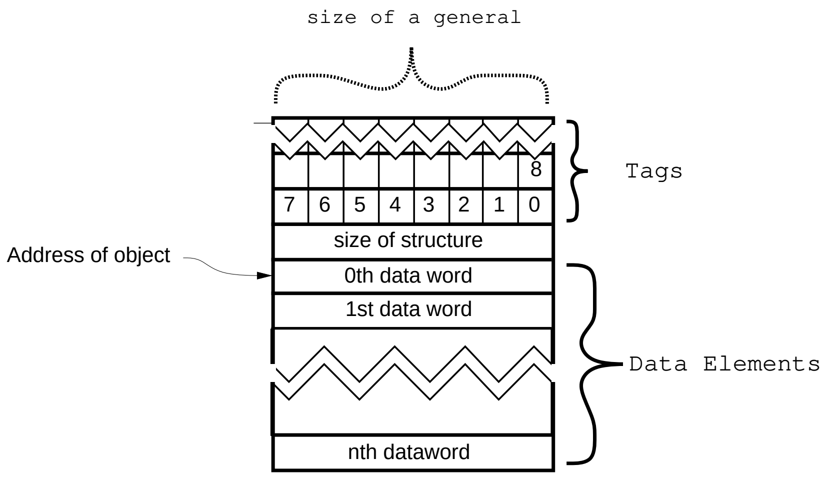


Figure 3.6: *Representation of an unpacked heap object.*

3.3 The Heap

The TLC implementation of a tagged heap is designed to be completely portable across machines, C compilers, and linkers. Since no special support from the linker is assumed[3], each structure must be completely self-contained. In this section we first describe the general structure of the heap, then how the deferred lists are maintained, and, finally, the instructions themselves.

3.3.1 Heap Representation

The unpacked heap objects, used to represent I-structures and M-structures, each contain data/tag pairs. The data elements are each as large as a `general`, which, in all of the C representations, is the size of two integers. In order to facilitate fast indexing and efficient packing of the data and tags, all the data is stored in a contiguous region and all the tags are in a separate contiguous region, one tag per byte. The schema for this representation is shown in Figure 3.6.

An n-element structure requires n data elements, each of size `sizeof(general)`, one word for system use, and $\lceil n/t \rceil$ words, where t is the number of tags that fit in a general-sized word. The data is indexed as a normal C array, i.e., the i^{th} element is offset i from the base pointer, e.g. `base[i]`. The tags are indexed in the negative direction from the base pointer, i.e., the tag for element i is found at the i^{th} byte less than the base of the tags, e.g. `((char *)((base)-1))[-(i+1)]`. Unfortunately, this representation prevents us from performing the index arithmetic at the request site and passing a pointer into the I-structure, since the base of the structure must be known in order to calculate the address of a tag.

A packed heap object of n elements is represented as a $n + 2$ sized array. The size of the structure and the outstanding store count are stored at offsets 0 and 1 respectively. The i^{th} element is stored at offset $i + 2$.

[3]Linker support would allow all I-structures to be grouped in their own segment. This would permit the data and tag sections of the structures to be separated from each other and allow a single pointer to be passed instead of a base/index pair, etc.

3.3.2 The Deferred List

One of the goals of TAM is to show how to explicitly manage the machine resources for programs with dynamic logical parallelism. This includes the register set, memory system, and network. Registers are explicitly named in each codeblock. All of the local memory required to execute a function is completely specified at the time the function is invoked, i.e. at frame allocation time. No message activity on the network is started unless the compiler can guarantee that the resources exist to extract the messages. (This is actually closely related to frame allocation.) The goal is that except for the algorithmic storage requirements—which obviously can't be bounded in advance—all memory requirements should be explicitly stateable in TL0.

In the TLC implementation of TL0 we relax this requirement in one case—the deferred list. The deferred list, as defined in Section 2.2.1, is used to keep track of the outstanding requests to an empty I/M-structure element. It could be stored in preallocated memory. If the deferred list has only one element, it could be stored in the I-structure slot itself. This assumes, however, that every element of an I-structure is big enough to hold a continuation. In practice this is possible for all uniprocessor and some multiprocessor machines, but a problem arises if further deferred fetches or takes are issued, since there is then no longer room in the I-structure itself. In principle, we could create the deferred list out of the I-structure slot and the frame slots that are expecting the fetched data. In this way, the linked list of deferred elements would be built out of the requesting frames, and the memory for the deferred list, like all other memory resources, would be preallocated.

For several reasons we do not require this of a TL0 implementation. First, individual deferred lists tend to be small. Second, it seemed overly restrictive to require that deferred entries be distributed around the machine. Finally, it would require each frame slot that gets filled by an IFETCH to be as large as a continuation, even if the data type of the fetch was smaller.

If an element builds up a deferred list of more than one element, then when the ISTORE is processed, more than one message will have to be sent from the "inlet" that processed the ISTORE. If the library routine is indeed executed as an inlet (i.e., it is run on a parallel machine and the request arrives from a remote processor), then the proper way to implement this is to send only one reply, enqueueing the rest of the deferred list on a system frame which will send the remaining replies. For pragmatic reasons, we do not follow the TL0 specification. First, we have observed that in actual practice deferred lists tend to be small. Second, we thought that minimizing the latency of the operation was important. We do, however, implement all other system inlets in conformance with the TL0 constraints outlined in the previous chapter.

3.3.3 Heap Instructions

Allocation

Heap allocation is performed with one of the two allocation instructions, IALLOC and AALLOC. These instructions are translated into library calls as follows.

IALLOC *num*`.j = [` *size*`]` *policy* $\Longrightarrow$ *policy*`_ialloc((localFramePtr)fp,I_`*thiscblk*`_`*num*`,` *size*`);`

AALLOC *num*`.j = [` *size*`]` *policy* $\Longrightarrow$ *policy*`_aalloc((localFramePtr)fp,I_`*thiscblk*`_`*num*`,` *size*`);`

The library calls allocate the appropriate amount of memory[4], clear the tag section (or initialize the counter in the case of A-structures), and send the pointer of the newly allocated and initialized memory back to the reply inlet.

[4]See Section 3.6.6 regarding special processing done on parallel machines for large unpacked objects.

The free operations are straightforward. The TL0 instructions are translated into direct library calls, which return the allocated memory to the heap.

```
IFREE ptr.ps policy   ⟹   policy_ifree(ptr);
AFREE ptr.ps policy   ⟹   policy_afree(ptr);
```

The clear operation, which allows an I-structure to be reused, also issues a call to the library, which in turn clears the tag area by setting the entire region to zeros.

```
ICLEAR ptr.ps   ⟹   iclear(ptr);
```

I-Structure Operations

The most basic translation of the I-structure operations turns IFETCH into a send to the system inlet which handles the ifetch operation, and does the same for an ISTORE. In the case of IFETCH, the library routine checks the tag and, if it is `full`, sends the result back to the specified inlet. If that tag is `deferred`, it enqueues the request on the specified I-structure slot. Otherwise it reports an error. While this scheme is adequate, it introduces unnecessary overhead in the most common case, which occurs when the data is present before the fetch is executed. In this case, as in the case of FALLOC, if the I-structure is local to the processor executing the fetch, the inlet can be inlined into the thread performing the fetch. At the cost of some code expansion, a store to a local empty element can also be performed without a function call to the library.

The simplest translation of IFETCH turns the TL0 instruction into a library call which depends on the type of the element being fetched. This is determined by the type of the RECEIVE instruction's argument in the inlet receiving the element fetched. For instance,

```
INLET x                            ⟹  I_thiscblk_x(localFramePtr ifp, generalAsArg arg0)
  RECEIVE result.i.g                  {
  POSTQ y.t default                     ⋮
  STOP                                }
⋮                                     ⋮
  IFETCH x.j = base.ps[index.i]       ifetchg(base, index, I_thiscblk_x, asGlobalFP(fp));
```

In this case the expected result is a `general`, which is stored into an integer slot in the frame. If instead the argument to the RECEIVE instruction were `result.i`, then the library routine called would be `ifetchi`. The type suffixes appended are the same as the suffixes of the types in the RECEIVE instruction.

The translation in the inlined case is more involved than that of FALLOC because it must also take into account the possibility that the fetch will be deferred. This is handled by translating the fetch into a conditional. If the data is present, the inlined inlet is executed. In other words, the thread both performs the fetch and handles the result. If the data is not present, a call to the library is made, and the thread executes only the request part of the fetch.

```
INLET x                            ⟹  ⋮
  RECEIVE result.i.g                   if (*tagOf(base, index) == FULL) {
  POSTQ y.t default                      fp->result = base[index].i;
  STOP                                   translation for fork y.t
  ⋮                                    } else {
  IFETCH x.j = base.ps[index.i]          ifetchg(base, index, I_thiscblk_x, asGlobalFP(fp));
                                       }
```

The ifetch library routines implement I-structure semantics by testing the tag associated with the fetched element and then switching on one of the four possible values: `full`, `empty`, `deferred`, and `thunk`. If `full`, it executes a reply which sends the data back to the requester. If `empty`, it allocates a deferred structure into which it stores the reply continuation, changes the tag value to `deferred`, and stores the pointer to the deferred structure in the I-structure element. If the tag is `deferred`, it performs the same operations as in the `empty` case, but also builds a linked list with the pointer (to a previously deferred request) already in the element.

If the tag indicates that a thunk needs to be evaluated, the value in the element is retrieved and used to initiate the thunk, while the original request is enqueued on the element as if the element had been empty at the time of the fetch. The thunk terminates its execution by issuing an ISTORE into the element. This ISTORE processes the original—and any new—deferred requests, just as it would handle any any other store into a deferred element.

The ISTORE instruction is specialized to handle the different TL0 types depending on the argument to the store.

`ISTORE` *base*`.ps[`*index*`.i] =` *data*`.`*type* $\Longrightarrow$ `istore`*type*`(`*base*`,` *index*`,` *data*`);`

Like the ifetch operation, the library call checks the value of the tag and takes appropriate action. If `full` or `thunk`, it signals an error. If `empty`, it stores the value and sets the tag to `full`. If `deferred`, it sends the value being stored to every reply continuation in the deferred list, freeing each deferred structure as it finishes. It then stores the value in the element and sets the tag to `full`.

M-Structure Operations

The translations for the M-structure operations follow the same format as for the I-structure operations, substituting ITAKE for IFETCH and IPUT for ISTORE. The only real difference is in the library routines that implement the functionality of the operations.

Like the ifetch implementation, the itake implementation checks the tag associated with the element being "taken" and takes appropriate action. If the tag value is `full`, it returns the value in the data element and sets the tag value to `empty`. If the tag is `empty`, it changes the tag value to `deferred` and enqueues a deferred request on the element. If the tag is `deferred`, it builds up a deferred list by adding a deferred entry for the current request. All other tag values produce an error.

The iput routine on an `empty` element stores the data value in the element and changes the tag value to `full`. If the tag value is `deferred`, it dequeues *one* of the deferred structures and sends a reply back to the reply continuation. If there are no more entries on the deferred list, then it changes the tag value to `empty`; otherwise, it leaves the tag in the `deferred` state. All other tag values cause an error.

A-Structure Operations

The translations for A-structure operations parallel those for I-structure and M-structure operations. However, unlike those for unpacked arrays, the library routines do not check any tags. Rather they check the counter stored in the structure. For every AWRITE the counter is decremented. If the count goes negative, an error is signaled. If an AREAD is executed before the counter is zero, a deferred entry is enqueued on the structure; otherwise the value being read is returned.

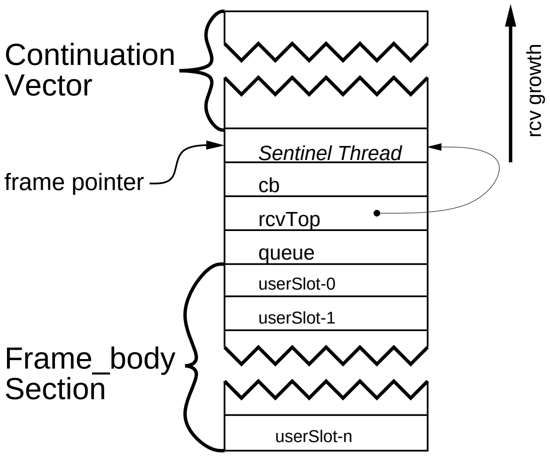


Figure 3.7: *Frame and* LCV *structure for the* RCV*-only implementation.*

3.4 Control Implementations

The rest of this chapter discusses the implementations of the TLC macros. It first presents the machine-independent macros, those concerning the basics of control, i.e., manipulations of the RCV, LCV, and the scheduling queue. This is followed by the implementations specific to sequential and parallel machines, i.e., those related to frame allocation, heap management, and send instructions.

The implementation of the TLC control macros is intimately tied up with the structure of the system section of every frame. We will present three approachs to implementing the macros, each of which is composed entirely of C code and, therefore, completely portable. The simplest and least efficient is an RCV-only implementation. The second two use an LCV; one copies on entry to a function, the other swaps on entry to the function.

3.4.1 The RCV-Only Implementation

The RCV-only implementation, as its name implies, does not use an LCV. Rather, the CV attached to every frame (see Figure 3.7) is big enough to hold the total number of threads that can be posted or forked by the codeblock. As in all the implementations, the RCV grows downward, away from the base of the frame. The pointer to the top of the RCV is `rcvTop`. Finally, at the bottom of the CV stack is a thread sentinel, placed on the bottom of the CV at the time the frame is initialized.

The `lcvDeclaration` macro is a nop, as is the `rcvInit` macro. Threads are popped off the RCV using the `lcvPop` macro.

`lcvPop()` $\Longrightarrow$ `*(fp->rcvTop++)`

Since there is no LCV, the `lcvPush` also accesses the frame in order to push a thread on the CV.

`lcvPush(x)` $\Longrightarrow$ `*(--fp->rcvTop) = x`

Instead of directly testing the CV in order to detect when a frame has no more threads, we use a sentinel thread. The sentinel thread is a synthetic thread created by TLC, which performs any necessary cleanup and executes the leave thread. The sentinel's thread identifier is placed at the bottom of the RCV, so that it is executed only when there are no more threads for the current frame.

In the RCV-only implementation, the sentinel thread resets the `rcvTop` pointer before invoking the leave thread. The pointer is reset in order to assure that the next post instruction does not overwrite the sentinel thread. If there is only one leave thread, then it can be inlined into the sentinel thread. The actions taken at the sentinel thread are thus

```
/* sentinel thread runs this code */
fp->rcvTop = fp;
leave thread goes here
```

When this scheme is used, no extra state is needed to determine if the frame is idle. If the RCV is empty, then the `rcvTop` pointer is the same as the `ifp`.

```
pushPost(thr)  ==>  {
                      int idle = (int *)ifp == ifp->rcvTop;
                      *(--ifp->rcvTop) = x;
                      if (idle) {
                        ifp->queue = Queue;
                        Queue = ifp;
                      }
                    }
```

This scheme requires that the `lcvPop` and `lcvPush` macros execute atomically with respect to inlet execution. Note that if an inlet is executed between the time the sentinel thread is popped off the RCV and the time the sentinel thread executes, the frame will be re-enqueued on the ready queue. In addition, the pushed thread will overwrite the sentinel thread. Thus, atomicity must be assured between the thread pop and the execution of its first instruction, or the sentinel thread will have to atomically check the value of `rcvTop`.

The fatal drawback to this scheme is that the very common thread operations `lcvPush` and `lcvPop` require three memory operations each—one to fetch the CV pointer, one to read (or write) the thread, and one to store the updated CV pointer. If, instead, we use a register to point to the top of LCV, we can eliminate the first and last memory references.

A secondary consideration is frame size. The CV attached to the frame must always be big enough to accommodate the maximum number of threads that can accumulate due to post operations. In the above scheme, the CV must also be large enough to accommodate the maximum number that can be forked while the frame is running. If a separate CV is used for fork operations, then the RCV can be made smaller and the total frame size can be reduced, leading to more efficient frame allocation and better cache behavior.

If we separate the LCV and the RCV, then we also need to introduce an LCV pointer. This pointer can be kept in a register, speeding up fork operations. In this case, the `lcvDeclaration` becomes a local C variable.

```
lcvDeclaration  ==>  int* lcvTop;
```

3.4.2 The RCV-copy Implementation

In order to remove both problems with the above implementation we use a single LCV per processor in addition to each frame's RCV. The LCV buffer is made large enough to contain the LCV of the largest codeblock. This allows us to reduce the size of the RCV to that required by the post instructions. Figure 3.8 shows the frame and LCV layouts. On entry to the thread function, the `rcvInit` macro copies each thread in the RCV onto the LCV.

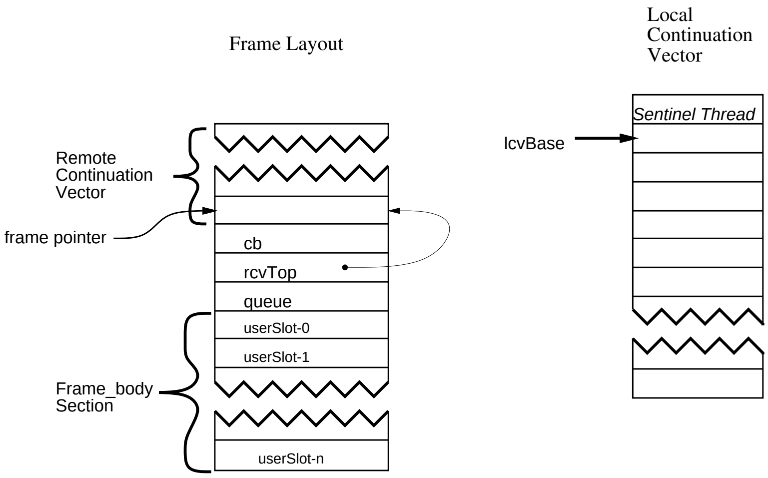


Figure 3.8: *Frame and* LCV *structure for the* RCV*-copy implementation.*

```
rcvInit()  ⟹  {
                 int num = (int*)fp - fp->rcvTop;
                 memcpy(lcvBase, fp->rcvTop, num * sizeof(int));
                 fp->rcvTop = fp;
                 lcvTop = lcvBase + num;
              }
```

The initialization code must also reset the `rcvTop` pointer, so that any inlets arriving during the execution of the frame are properly placed on the RCV. This change will effect the post macros and the actions taken in the sentinel thread.

In this model, the RCV grows downward, but the LCV grows upward. This allows the `rcvInit` macro to use the efficient `memcpy` routine instead of a `for` loop.

```
lcvPop()     ⟹  *(--lcvTop)
lcvPush(x)   ⟹  *(lcvTop++) = x
```

Using two separate continuation vectors adds complexity to implementing the two-level scheduling hierarchy. If we use the above implementation for the post instruction, then, even if the frame is running, it appears idle and is enqueued when the first thread arrives. Although this is a correct implementation it increases the number of times a frame is enqueued—increasing the number of swap instructions executed, and decreasing the number of threads executed per quanta.

In order to eliminate this unnecessary overhead, the sentinel thread can check the RCV before executing the leave thread. If the RCV is empty, the leave thread is executed; otherwise, the contents are copied into the LCV and execution begins again at the top of the loop.

```
/* sentinel thread runs this code */
if (fp->rcvTop == fp) {
   leave thread goes here
} else {
   int num = (int*)fp - fp->rcvTop;
   memcpy(lcvBase, fp->rcvTop, num * sizeof(int));
   fp->rcvTop = fp;
   lcvTop = lcvBase + num;
}
```

Since the sentinel thread is one arm of the `switch` embedded in the `while` forever loop (see Figure 3.3), execution just continues as if the LCV had not been empty.

Using this scheme, the post instruction can no longer check the value of `rcvTop` to determine if the frame is idle. At the cost of a memory write every time a frame is scheduled, we save the value of the current frame pointer and check this value against the value of `ifp` in the post instruction.

```
pushPost(thr)  ⟹  {
                    int idle = ((int *)ifp == ifp->rcvTop)&&(ifp != cfp);
                    *(--ifp->rcvTop) = x;
                    if (idle) {
                      ifp->queue = Queue;
                      Queue = ifp;
                    }
                  }
```

The main scheduling loop becomes:

```
localFramePtr        fp, cfp;
extern localFramePtr Queue;

while(1)
{
    fp = Queue;
    Queue = Queue->next;
    cfp = fp;
    fp->cb->threadFunc(fp);
}
```

Although this implementation keeps frame sizes small and allows fork operations to use registers, it is unsatisfactory in that the asymptotic cost of running a frame is proportional to the size of the RCV. In addition, if requests made during a quantum complete in the same quantum, then we still must copy the thread pointers from the RCV into the LCV.

3.4.3 The RCV-Swap Implementation

We can make the cost of scheduling a frame a constant at the expense of a modest increase in the frame size. If the CV attached to each frame is large enough to accommodate all the local fork operations, we could use the RCV as an LCV when the frame is running, as in the RCV-only implementation. In order to use a register to point to the LCV, we point the `rcvTop` pointer at the temporary CV buffer. Thus, on entry to the frame, we need to change the `rcvTop` pointer and initialize the `lcvTop` pointer. In addition, we need two new variables to keep track of the base of the RCV (as explained below). Figure 3.9 shows the frame and LCV layouts.

```
lcvDeclaration  ⟹  int* lcvTop;
                   int* rcvBase;
                   int* lcvBase;

rcvInit()       ⟹  {
                      fp->rcvTop = sysLcvTop;
                      rcvBase = sysLcvTop;
                      lcvTop = fp->rcvTop;
                      lcvBase = fp;
                   }
```

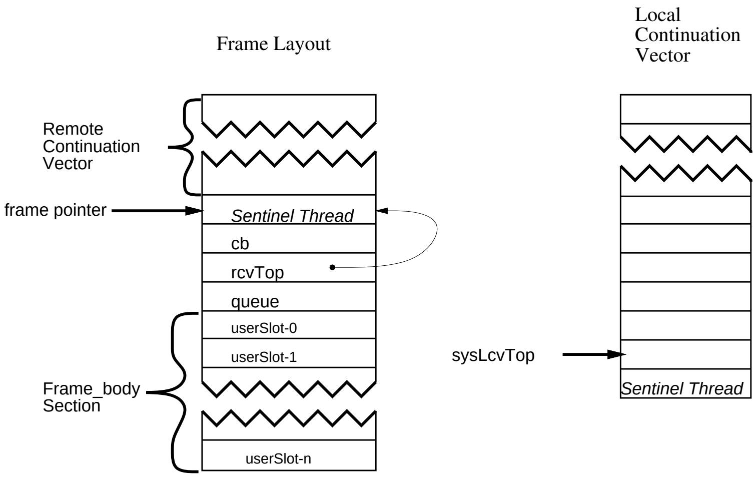


Figure 3.9: *Frame and* LCV *structure for the* RCV*-swap implementation.*

The LCV again grows downward and the LCV operations become

```
lcvPop()      ⟹   *(lcvTop++)
lcvPush(x)    ⟹   *(--lcvTop) = x
```

The sentinel thread must not only check to see if threads have been enabled by post instructions, but also, if the RCV is empty, must point the frame's `rcvTop` back into the frame itself. In order for the sentinel to determine that the RCV is empty it compares the current top of the RCV as stored in the frame with the local variable `rcvBase` (hopefully in a register). If they are the same, it resets `fp->rcvTop` to point back to the frame and runs the leave thread. If the RCV is not empty, it swaps the RCV and LCV pointers and continues execution of the thread function.

```
/* sentinel thread runs this code */
if (fp->rcvTop == rcvBase) {
  fp->rcvTop = fp;
  leave thread goes here
} else {
  int *rtop = fp->rcvTop;
  int *rbase = rcvBase;

  fp->rcvTop = lcvBase;
  rcvBase = lcvBase;
  lcvTop = rtop;
  lcvBase = rbase;
}
```

The post instruction remains unchanged. Indeed, whenever the RCV pointer is pointing into the `sysLcv` buffer it will not even need to check the `cfp` variable to determine that the frame is not idle!

In summary, the first implementation, though simple, is unacceptable due to the high cost of each LCV operation. The second implementation requires $4n + 14$ memory operations to schedule a frame with n threads on the RCV, but has the smallest possible frame size. In the best case the RCV-swapping implementation takes $2n + 15$ memory accesses to schedule a frame with n threads. In the worst, where `rcvBase` and `lcvBase` are not kept in registers, it takes $2n + 19$ memory accesses. Either way, it increases frame size due to the requirement that the RCV be large enough to act as an LCV. In addition, the threads posted while the frame is running require half as many cycles. Using our benchmark suite [8], we discovered that on sequential machines the average RCV size is 3.3 and 1.2 threads arrive on average while the frame is running, making the last implementation on average about six memory references cheaper than the copying implementation. Similarly, on parallel machines it is about three memory references cheaper. This seems to warrant making the frame size slightly larger to accommodate the LCV.

Though we have achieved a reasonable implementation for these operations it is obvious that using C as a target language has severely reduced our ability to optimize them. For instance, if the inlet function had access to the thread function registers, it could determine which frame was running without touching the memory system. Indeed, a post operation for the current frame should be able to put the thread *directly* on the LCV—eliminating all superfluous memory references as well as the slightly bothersome check in the sentinel thread. These and other optimizations are discussed in Section 3.7.

3.5 Sequential Machine Implementations

In general, the implementation of the macros for uniprocessors is straightforward and should follow from the previous descriptions. Here we describe the two remaining undiscussed issues—send and frame allocation.

3.5.1 Send Macros

On a sequential machine the send macros are implemented as function calls. In other words, the network interface is removed from the active message loop and the "sender" calls the inlet directly.

send*n*args(dest, size, arg_1, ... arg_n); $\Longrightarrow$ dest(arg_1, ... arg_n);

3.5.2 Falloc

The library routines that implement the different translations of the FALLOC instruction all use the same frame allocator. They differ only in how the acknowledge and initialization inlets are specified. All the default allocation routines are composed of the allocation macro and a send (or call) to inlet 0 of the new codeblock. This is shown for the most general of them, `default_falloc00`.

```
default_falloc00(localFramePtr callerFP, inlet callerInlet,
cbDesc* desc)
{
   f = DO_ALLOCATION();
   (*desc->inlets[0])(f, asGlobalFP(callerFP), callerInlet+1);
}
```

The `DO_ALLOCATION` routine, actually a macro in the implementation, finds an appropriately sized block of memory and initializes the RCV and `cb` pointer. Inlet 0 is then called with a pointer to the new frame. The allocator is built to optimize for a common feature that many programs exhibit: often the same codeblock is invoked many times, but each activation has a short lifetime. We optimize for this by allowing each codeblock to keep a pool of previously freed frames. These frames already have their RCV initialized, etc., so allocating them is cheaper.

```
DO_ALLOCATION()
{
  localFramePtr f;

  if (desc->pool)
  {
    f = desc->pool;
    desc->pool = f->queue;
  }
  else
  {
    int* i = (int *)myMalloc(desc->totalSize);
    f = (localFramePtr )(i + desc->frameOffset);
    f->rcvTop = (int *)f;
    *(int *)f = THREAD_SENTINEL;
    f->cb = desc;
  }
  return f;
}
```

The code for initialization shown above reflects the RCV-swapping model described in Section 3.4.

3.6 Parallel Machine Implementations

The implementation of the TLC macros presented in this section are for the distributed memory machines to which TLC has been ported. Shared memory machines, like the Sequent, have also been ported to, but are little different than the sequential implementation described above. For distributed memory machines, TLC must deal directly with the network interface (by using active messages), handle the separate address spaces of each processor, and consider the multiple threads of control.

The presentation here applies for the most part to any distributed memory machine. However, certain issues are specific to the CM-5 and are noted as such in the text. First we show how the global address space is reflected in the data types used by TL0. Then we discuss atomicity issues. We then present the implementations for the send, frame allocation, and heap access instructions.

3.6.1 Data Types

Except for the pointers, all types are translated the same way regardless of whether the target is a uni- or multiprocessor. In the case of the frame and structure pointers, the uniprocessor versions use standard C pointers, while the multiprocessor translation must use a *global pointer*. A global pointer is a structure which denotes both a processor and the local address on the processor.

In order to ensure that global pointers are passed in registers some hackery is required. In the case of the CM-5 implementation, which used the GCC compiler, the global pointers are represented as `long long`. This ensures that they are passed in registers. In order to access their components, macros and inline functions are used to cast the `long long` to a union, `GPunion`, which is then dereferenced to access the processor and address fields. See Figure 3.10 for the details.

```
typedef long long framePtr;
typedef long long GlobalPtr;
typedef long long structPtr;
```

These are each big enough to hold a processor id and a local address.

```
typedef struct {
    int pid;
    int adr;
} GlobalPtrAsStruct;
```

Defines the internal structure of a global pointer.

```
typedef union {
    GlobalPtrAsStruct asStruct;
    long long         asLong;
} GPunion;
```

Used to transfer between the long long and the structure representations.

```
extern GPunion lgFPbuffer;
inline GlobalPtr asGlobalFP(void *ptr)
{
    lgFPbuffer.asStruct.adr = (int )ptr;
    return lgFPbuffer.asLong;
}
```

Constructs a global pointer with pid set to the current processor.

```
inline Processor toProc(GlobalPtr ptr)
{
    return (Processor )((GPunion) ptr).asStruct.pid;
}
```

Returns the processor part of the pointer.

```
inline localFramePtr asLocalFramePtr(GlobalPtr ptr)
{
    return (localFramePtr )((GPunion )ptr).asStruct.adr;
}
```

Returns address part.

```
#define GlobalPointerAsArg(x)   (x).asLong
```

Used to pass a global pointer as an argument to a function.

```
#define SET_PID(x, y)   (((x).asStruct.pid) = (y))
#define SET_ADR(x, y)   (((x).asStruct.adr) = (y))
```

Figure 3.10: *The structure of global pointers in the* CM-5 *implementation.*

3.6.2 Atomicity

One of the primary differences between the sequential implementation and the parallel implementation of TLC is that the parallel implementation must deal with issues of atomicity. As has been described, certain operations must be atomic with respect to inlet execution. There are two ways to ensure this: Either interrupts can be turned off and on around the portions of code that must be atomic, or interrupts can be disabled entirely and the compiler can insert polling statements that check the network interface when it is safe to do so. The first implementation of TLC, on the nCube-2, used the former approach. The current implementation uses the latter approach because the overhead in processing an interrupt on the CM-5 is prohibitive. The polling approach outlined here is easier to describe, as no interrupt enable or disable instructions need to be included. Our presentation, however, is not meant to suggest that we in general favor the second approach over the first; it is just simpler to explain.

The TLC macro that polls the network is `poll`. On uniprocessors the `poll` macro is a nop.

One major drawback to polling is, of course, the extra overhead introduced by the polling statements, which must be sprinkled liberally throughout the code. If we ignore the network interface for too long the network will become backed up and slow down all the nodes. Further, if the network is not serviced regularly, the latency for split-phase operations becomes intolerable.

3.6.3 The Main Control Loop

In the sequential implementation, when the frame queue becomes empty, the computation is finished. In the parallel case, a single node may have no work at present, but may receive messages that start, or re-start, inactive frames. Thus, if no frames are on the ready queue, the main scheduling loop polls the network. Otherwise, it processes frames just like the sequential implementation. When the computation has completed, i.e. when the main codeblock signals that it is done, the `notDone` variable is set to zero on all processors.

```
extern int           notDone;
localFramePtr        fp, cfp;
extern localFramePtr Queue;

while (notDone)
{
    while(Queue)
    {
        fp = Queue;
        Queue = Queue->next;
        cfp = fp;
        fp->cb->threadFunc(fp);
    }
    poll();
}
```

3.6.4 Send Macros

Because the cost of sending a message is substantially higher than that of calling a local function, the TLC `send` macro is specialized to invoke routines directly if the destination node is the same as the sending node.

```
sendnargs(dest, size, destFp, arg1, ... argn);   ⟹

      if (toProc(destFp) == MYPID)
        (*dest)(asLocalPtr(destFp), arg1, ... argn);
      else
        am_sendn(toProc(destFp), dest, asLocalPtr(destFp), arg1, ... argn);
      endif
```

The am_send$n$ macros are used to invoke the active message layer [16]. The translator generates only sends that can fit in a single message packet, which is determined by the machine. (See Figure 3.5.)

3.6.5 Falloc

In TL0 at the higher level of scheduling, the frame is the basic unit of work. A successful parallel implementation must allocate this work evenly among the processors to guard against processors becoming idle. Due to the high cost of sending a message, a balanced distribution of work is not sufficient to guarantee good performance. In addition to an even distribution of frames, we must attempt to make messages local whenever possible, thereby minimizing communication. This tension between load balancing and locality is central to the design of the frame allocator.

The TLC implementation of the frame allocator is oblivious to the source input program and attempts only to provide the best default behavior. To allow experimentation with different allocation policies, four allocators are provided. The user can choose among the four and, where applicable, set parameters for them before execution of the program.

The basic function frame allocators are built around two library routines, the request routines and the service routines. The request routine, either falloc_default*xx* or falloc_strict*xx*, decides where the frame will be allocated. The service routine, getMem, does the actual allocation and initialization. A separate allocator, the kloop allocator, handles the allocation of k-bounded loop frames. Each of the frames in a k-bounded loop is allocated on subsequent processors. This allows each of the iterates of the loop to work in parallel.

The four function-allocation policies are: local-only, distributed, random-range, and sensing. The local-only policy allocates a frame on the same processor that issued the FALLOC. This policy ensures a high degree of access locality. It works well with the kloop policy because it keeps all the functions spawned by a loop frame on the same processor as the iterate that spawned it. If the amount of work for each iterate of the loop is constant, this policy is best, since the kloop allocator keeps the processor's load balanced while the frames needed to carry out the work of each iterate all remain local.

When there are no k-bounded loops in the application, or when the work for each iterate is not constant, the work must be distributed by the function calls themselves. The distributed policy is the easiest to implement and understand. It takes one parameter, the distribution list, which is a list of integers. The request routine marches through the list and allocates the current frame on the processor number, determined by adding the current processor number to the index in the distribution list. Thus, each successive frame is allocated relative to the processor determined by the list. When the list is exhausted it is reused. For instance, a local-only policy could be created by using the distribution list $[0]$, which allocates each frame on the same processor. A policy which allocates every other frame on the current processor or one of its immediate neighbors results from the list $[-1, 0, 1, 0]$.

The random range policy also uses a distribution list, but chooses the processor on which to allocate the frame by picking an offset within a random range specified in the list. This works well for problems with low locality needs [4]. If the distribution list contains the number n, it will allocate the frame on a random processor $\pm n$ from the current one. For instance, to randomly allocate approximately 42% of the frames within a neighborhood of seven processors and 58% on the local processor, the following distribution list could be used: $[3, 0]$.

Finally, the sensing policy allocates the frame on the local processor unless another processor has requested a frame. Processors request frames by sending request messages to other processors when they are idle. The set of processors that can be sent a sensing message is determined by a distribution list, e.g.$[-10, -5, 0, 5, 10]$.

These frame allocators can be compared using a two-dimensional space that represents the different instances of the parallel frame allocators described above. One axis, the selfishness of the policy, is the probability that a frame is allocated locally. The second axis denotes the size of the neighborhood into which a frame will be allocated. We use the notation sXXnYY to denote a policy, where XX is the percent of selfishness and YY is the size of the neighborhood. s100n1 denotes a completely local policy. The distributed policy with the list $[-1, 0, 1, 0]$ is described by s50n3. While, s64n7 describes the random policy of $[3, 0]$.

3.6.6 The Heap Instructions

The parallel heap is divided into three sections: local, spread, and constant. When heap objects are allocated, they are either allocated on the local heap or they are spread among the processors in the spread heap. We allocate all "small" objects on the local heap. In general, small means less than 128 words. Large heap objects are spread cyclically across the processors such that element n is on processor $n(mod P)$, where P is the total number of processors. Finally, the constant heap constrains all the statically defined heap objects. In order to maintain consistency, we assign a single processor, processor 0, the task of servicing all static heap objects.

The heap macros support the above division of the heap. Every access macro checks if the heap element being accessed is local. If it is, the macro invokes the library routine that implements the access involved. If the element is remote, then a routine is invoked which checks to see if the object is a spread object. If it is not a spread object, then it sends a message to the appropriate service routine on the destination processor. In the case of a spread object, it first translates the address and index to reflect the structure of the spread heap, and then tests again to see if the translated address is local to the processor. It either sends a message to the remote handler or invokes the local handler directly. The service routine then invokes the local library routine which carries out the requested task.

For the ifetch instructions there are six routines: `ifetch1`, `ifetch2`, `fetch1`, `fetch2`, `service_fetch1`, and `service_fetch2`. The suffix 1 or 2 indicates whether one or two words are being fetched. Both `ifetch`*x* routines perform a local fetch. Both `fetch`*x* routines perform the request part of the fetch for remote objects. The two service routines are the active message handlers for performing the fetch on remote objects.

The non-optimized version of `ifetchg` thus becomes

```
ifetchg(base, index, inlet, fp)  ==>
              if (toProc(base) == MYPID)
                ifetch2(asLocalPtr(base), index, inlet, asLocalPtr(fp));
              else
                fetch2(base, index, inlet, fp);
```

The local fetch routine `ifetch`*x* is no different from its sequential implementation. The `fetch2` routine checks to see if the remote address is a pseudo-address—that of the global constant processor—and performs the appropriate message send. The macros `gis2pid` and `gis2adr` map the base/index pair of the original element into a processor id and local index. The processor id indicates which processor owns the element. The local index specifies where the element is found on that processor. The macro `INDEX_PID` maps an object index and processor number into a single integer word. This is needed on the CM-5 implementations since only four data words can fit in a single message. The overhead of sending an additional message—including synchronization—is prohibitive, so instead we combine these two. If we ran on more than 2^{10} processors or had indices of more than 2^{22} we would have to change this code.

```
void fetch2(structPtr base, int index, cbPtr dest, framePtr fp)
{
  int            pid   = toProc(base);
  localStructPtr lbase = asLocalPtr(base);

  if (pid == GLOBAL_IS_PID) {
    pid = gis2pid(lbase, index);
    index = gis2adr(lbase, index);
    if (pid == MYPID) {
      ifetch2(lbase, index, dest, fp);
    }
  }

  am_send4(pid, service_fetch2, lbase,
           dest,  INDEX_PID(index, toProc(fp)), asLocalPtr(fp));
}
```

The remote service routine handles the fetch and sends a reply if necessary.

```
void service_fetch2(localStructPtr base, cbPtr dest, unsigned int,
                    indexPID, void* adr)
{
  ifetch2(base, INDEX_OF(indexPID), dest,
                         makeGPTR(PID_OF(indexPid, adr)));
}
```

The equivalent optimized macros, which attempt to inline the request, are

```
ifetchgStart(base, offset, var)  ⟹
                        if ((toProc(base) == MYPID) &&
                            (flagOf(asLocalPtr(base), offset) == FULL)) {
                          var = base[offset];
ifetchgEnd(base, offset, dest, fp)  ⟹
                        } else {
                            if (toProc(base) == MYPID)
                              ifetch2(asLocalPtr(base), offset, dest, fp);
                            else
                              fetch2(base, offset, dest, fp);
                        }
```

The heap allocation routines are more complex in the multiprocessor case because they must also handle the allocation of structures across all processors. If the size of the request is small, then a local heap structure is allocated and the result is returned to the specified reply inlet. If the size indicates that a spread allocate is needed, then a message is sent to the processor that manages the spread heap. Since we wish the spread structure to occupy the same addresses on each processor, we divide the heap into two address spaces: local and spread. The local space contains the frames and local I-structures. It grows from low to high memory. The spread heap grows from high memory down. The heap-manager processor keeps track of how much of

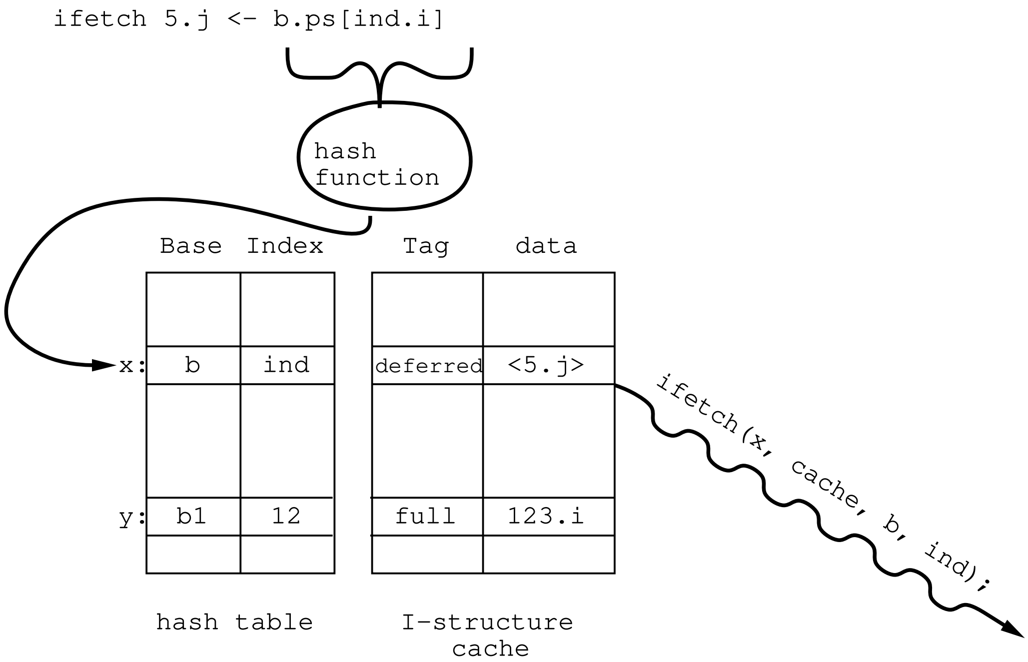


Figure 3.11: *Example of an ifetch made to an empty slot in the cache. The pair (b,ind) is hashed to index x of the cache I-structure, which causes a deferred request to inlet 5 to be enqueued in the I-structure. It also sends out a fetch message which will reply to the cache inlet. Instead of specifying an* `ifp` *in the request, the address of the cache slot is included, so an* ISTORE *can complete the cycle. If a request for (b1, 12) is made, it will be fulfilled immediately.*

the spread heap has actually been allocated.

On each processor there is a variable that indicates the maximum local heap it is allowed to use. If it ever tries to allocate more space, it sends a request to the heap-manager for more local space. The heap-manager grants the request as long as the request does not encroach on already allocated spread memory. If the heap-manager needs to allocate more memory than is available to it, it will broadcast a request to all processors to lower the top of the local heap. This protocol allows most memory allocation requests, local or remote, to occur without requiring all processors to be involved in the request.

The spread allocation request is handled by sending a message to the heap-manager processor. This processor then schedules a system frame, the `spreadAllocator` frame. The reason that a frame must be scheduled is that the request requires the acquisition of resources, an operation that cannot occur in an inlet.

When the `spreadAllocator` frame runs it gets a chunk of memory from the spread heap—possibly requiring a broadcast to change the top-of-heap address. All memory in the spread heap is cleared at start-up time, so no messages are necessary to initialize the tag portion of the heap object. The allocator then sends the address of the newly allocated heap back to the reply inlet of the original requester. The global pointer used for spread objects includes a pseudo-processor id of -1, which signals all the heap access functions that it is such an object.

3.6.7 I-structure Caching

Though TL0 tolerates latency through the use of split-phase instructions, we also attempt to eliminate latency by implementing a software cache for I-structure operations. We chose to implement caching for I-structures, as opposed to M-structures, because the consistency of I-structures is guaranteed by their semantics. The only consistency operation required is to flush the cache when heap structures that were freed are reallocated.

The beauty of this is that the cache is implemented simply by changing the `fetch`*x* routines and adding two new system objects. The cache data is stored in an I-structure allocated by the system. We also create a table which stores the (base, index) pair associated with each element of the cache-data I-structure.

If an access is made to a remote element, we hash the address into the local cache I-structure. If the element is `empty`, we record the original (base, index) pair in the hash table and enqueue a deferred request in the I-structure cache, which will forward the result of the fetch to the original requester. (See example in Figure 3.11.) In addition, a fetch is sent to the remote location. However, the reply continuation is a system routine which performs a store to the hashed location. If the element is `full` and the hash table indicates the fetch is for the same element, it is immediately returned; otherwise we treat the element as if it were `empty`.

The interesting case is when the element has a `deferred` tag. If the entry is for the same base and index as the current fetch, we add a deferred entry which is fulfilled when the original request is satisfied. If it is for a different entry, we send the request to the remote location, bypassing the cache.

3.7 Using GCC

TLC as described in this chapter implements TL0 in a reasonably efficient yet portable manner. In some sense, TLC is just a representation, in C, of TL0. The instructions are mapped one-to-one to either macros or simple C statements. However, some control of the process is lost by going to C instead of directly to a native assembler. This section addresses this problem by presenting a compromise between a completely native mode translator—with its loss of portability and reduced ease of experimentation—and the C version described above. All of the extensions used here are described in the chapter on C extensions in [9].

We keep the overall structure used by TLC, but implement the macros using GCC extensions to C. This allows us to virtually eliminate the extra overhead for the control instructions.

3.7.1 Sources of Overhead

There are four sources of overhead introduced by using C as a target for TL0: the inability to specify register usage, code placement, and data structure placement; and C's structured control flow. The GCC extensions allow us to specify certain registers, to find and use the addresses of code fragments, and to bypass structured control flow with indirect jumps. Without modifying the linker we are for the most part unable to direct data placement effectively.

The overhead manifests itself in three areas: the specification of continuations, the TL0 control instructions, and the heap access routines. In addition, the compiler produces poor code for the inlet functions. We touch on the first and second issues in the remainder of this section.

In order to support the TL0 call and return protocol, inlet arithmetic must be performed. Since we do not have control over the placement of functions in the code space, we cannot directly add any constant to the the address of an inlet function to arrive at another inlet function. Thus we pass a pointer into a table with the function addresses. (See page 48.)

This is additional overhead imposed on the TLC translator. If a native mode translator were used, the inlet table would be unnecessary. Instead, all the inlets for a particular codeblock would start at some fixed address, each occupying a fixed set of words. Since most inlets consist entirely of a receive and a post, taking about 16 instructions on a risc processor, most inlets could execute in the small fixed space allocated them. Thus, if inlets were to be passed as first-class objects, the pointers to the routines themselves could be used. Since the TLC implementation does not have control of code placement, the indirection table must be used.

The translation of control instructions into C introduces the greatest source of overhead. In the thread function both the while loop and the switch statement insinuate extra branches and memory references. But, most important, since the inlet function is unable to access the registers and variables of the thread function, any threads posted for the running frame are unnecessarily put in the RCV, instead of the LCV.

All of the heap access functions require two memory accesses, one for the tag and one for the data. Since each heap object must be self-contained, we are unable to combine the base and index used in the instructions and both of these must be sent across the network interface. In addition, the index arithmetic is made more complicated by the required layout of heap objects.

3.7.2 Control

The first step in reducing overhead is to force the registers defined by the TL0 model into real registers that all functions can access. Thus, we put the FP, QP, and the LCVPTR in global registers. On a parallel machine, putting the address of the network interface status register into a global register speeds up the frequent poll instruction. So we add the following global declarations.

```
register localFramePtr fp     asm ("g6");
register localFramePtr Queue  asm ("g5");
register int*          lcvTop asm ("g7");
```

In addition to making the global state of the machine available to all functions, by judiciously putting local variables in certain registers we can speed up the send instruction when `generals` are involved.

We now revisit the discussion (in Section 3.4) about how to implement the RCV and LCV in light of the possible optimizations allowed by GCC. We begin by changing the main control loop to reflect the use of global registers, then show how first-class labels allows us to optimize the thread function, and finally show how the macros that manipulate the LCV and RCV are changed.

The main control loop stays essentially the same as specified in the RCV-swap implementation. However, we no longer need to save the current FP into `cfp`, since the current FP is always available in a global register. In addition, we no longer pass the FP into the thread-function. The inner loop now requires only three memory references to schedule a frame.

```
thread_func()
{
   threadPointer nextThread;
   static void* labels[] = { &&end,
                             &&label_1, ..., &&label_n,
                             &&phoney };
    thread registers declared here

   rcvInit();

    execute enter thread

   while(1) {
     nextThread = lcvPop();
     goto *labels[nextThread];
   phoney:
     switch(nextThread) {
     case 1: label_1:
         code for thread 1 goes here
       break;
     case 2: label_2:
         code for thread 2 goes here
       break;
     ⋮
     case n: label_n:
         code for thread n goes here
       break;
     default: end:
         code to execute for an empty lcv goes here
     }
   }
}
```

Figure 3.12: *A prototype for the improved thread function showing the main thread scheduling loop and* n *threads.*

```
extern int notDone;

while (notDone)
{
    while(Queue)
    {
        fp = Queue;
        Queue = Queue->next;
        fp->cb->threadFunc();
    }
    poll();
}
```

The layout of the thread function (see Figure 3.12) is changed to use first-class labels in GCC. The label for each thread is entered into the static array `labels`. The sentinel thread is numbered 0. The last label in the array is used to fool the compiler into thinking that the switch statement is reachable. Even though we no longer use them, keeping the `switch`, `break`, and `while` statments allows the compiler to continue to do flow analysis.

On entry to the thread function, `lcvTop` is initialized to point to the "top" of the RCV in the frame. In fact, we use the same basic form of the LCV manipulation macros as those in the RCV-only implementation. The frame's RCV is used for both the LCV and RCV. The difference is that when the frame is running, the RCV pointer in the frame is not used.

```
lcvPop()     ⟹  *(lcvTop++)
lcvPush(x)   ⟹  *(--lcvTop) = x
```

In addition, we translate the STOP instruction as a direct C `goto` statement. This eliminates the `while` loop overhead.

```
STOP  ⟹  goto *labels[lcvPop()];
```

The key optimization permitted by these changes is that the post instructions can now deposit arguments directly into the LCV if the target frame is the same as the running frame.

```
pushPost(thr)  ⟹  {
                      if (fp == ifp) {
                          lcvPush(thr);
                      } else {
                          int idle = (int *)ifp == ifp->rcvTop;
                          *(--ifp->rcvTop) = x;
                          if (idle) {
                              ifp->queue = Queue;
                              Queue = ifp;
                          }
                      }
                  }
```

These optimizations speed up all posts, especially those made to the running frame. Since no check is needed against the `cfp` variable, posts made to the running frame cost only one cycle more than thread forks.

Since no threads are ever inserted into the RCV during a quantum, the sentinel thread is executed only when there is no more work in the frame. So we need only to reset the `rcvTop` pointer before we exit. The sentinel thread becomes

```
/* sentinel thread runs this code */
fp->rcvTop = fp;
leave thread goes here
```

In summary, we have reduced the cost of scheduling a frame to five memory references, regardless of the RCV size on entry. We have also eliminated the overhead for posts arriving for the running frame.

The astute reader may have noticed that we still pay the cost of an indirection in dispatching threads in the thread `while` loop. This overhead could be eliminated by saving the actual addresses of the threads on the RCV. We use indices instead for two reasons. First, GCC does not really support exporting first-class labels outside of a function's scope, although there are ways to do this using tricky `asm` statements. Second, using addresses would make the code for pushing a thread identifier on a continuation vector less efficient on processors like the Sparc, since it would require two extra instructions to build the address.

Other less interesting (and less profitable) optimizations involving conversions to generals, send instructions, the poll macro, and heap access are also performed, but not discussed here.

3.8 Discussion

In retrospect, we see that TLC is not just an implementation of TL0, but can be thought of as an alternative realization of TAM. Indeed, we could have chosen to define TLC as the direct target language from our high-level compiler. However, the two-step approach we have taken allows us to avoid having to specify the cumbersome control structures that are introduced in the TL0-to-TLC translation. TLC could also be used directly as a programming language for parallel machines.

In spite of the two-step process, and the use of C as a target language, we are able to produce efficient executables. Indeed, using the C extensions outlined at the end allows us to produce code rivaling that of a translator that goes directly to assembler. The remaining differences are in the implementation of I-structures.

Chapter 4

Conclusion

In the preceding chapters we have defined TL0 by both its operational semantics and its implementation by our TL0-to-C translator. These definitions arose from the Berkeley Id90 compiler project. TLC is the back end for our Id90 compiler. It has been used to run Id90 programs on several commercial processors including the Sparc and Mips. In addition, TLC produces executables for the Sequent Symmetry, the nCube-2, and the CM-5.

To date, this is the only viable implementation of Id90 on conventional machines. Indeed, the performance of our current implementation on a Sparc-10 rivals the performance of Id90 on special-purpose machines like the Monsoon [12]. All of our benchmark programs run faster on a single Sparc-10 than on a single-processor Monsoon [2].

In addition to providing users with fast Id90 executables on stock workstations and commercially available parallel machines, we have created a tool that allows us to study the dynamic behavior of large parallel applications on real machines. The TLC translator is able to produce instrumented executables which allow us to analyze the dynamic runtime behavior of our programs.

The first studies we performed were used to determine if our front-end compilation strategy was effective in reducing the overhead normally found in dataflow implementations [8, 13]. This work, which looked at various partitioning schemes of the dataflow graph, forms the basis for the front end of the compiler. The front end attempts to partition the program into threads, thereby eliminating as much control overhead as possible while keeping the latency tolerance of the original code. However, it is not always possible to statically prove that certain threads can be run in the same quantum. Here TAM comes to the rescue. By providing inexpensive scheduling operations, it allows even the dynamic control cost to be lowered to a

Program	Input Size	Monsoon	Sparc-10
Gamteb 2-Cell	40,000	59	32.9
Paraffins	19	2.4	2.2
Simple	1 1 100	6	2.9
MMT	500	106	61

Table 4.1: *Dynamic runtime in seconds on a 1 processor Monsoon and for executables produced with our compiler on a SparcStation-10. The programs are described in [8].*

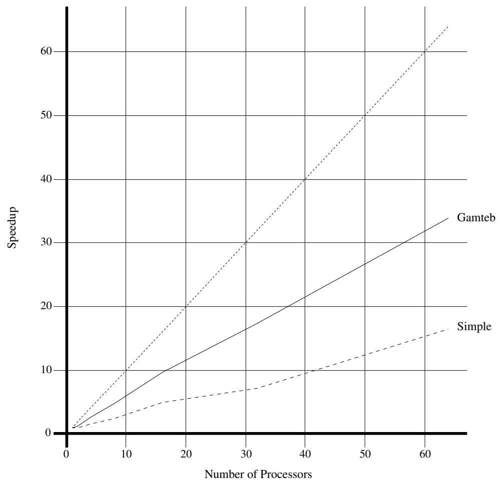


Figure 4.1: *Speed-up curves for Gamteb and Simple on the* CM-5*.*

reasonable limit.

One measure that can be used to evaluate the success of TL0 is the number of quanta per invocation. As we reported in [8], we found that there are generally between three and four quanta per invocation even in the presence of long remote latency as on the CM-5. Compare this to at least two quanta per activation on a sequential machine for a sequential language. Part of the reason for this seemingly low overhead is that before a frame is scheduled, it accumulates, on average, more than one thread in its RCV. Further, many of the split-phase operations return in the same quantum in which they are initiated. On all of our benchmarks, more than a quarter of the threads executed in a quantum are enabled by the post instruction; the others are forked from other threads.

In addition to evaluating TAM itself, we have also used TL0 to study different approaches to resource allocation on parallel machines. In [7], we investigate frame allocation polices, I-structure caching, I-structure spreading, and k-bounded loops. We studied our two largest benchmarks, Gamteb [3] and Simple [5], on the CM-5. Both programs exhibit linear speed-up from two to sixty-four processors. (See Figure 4.1. We don't get linear speed-up from one to two processors because of the effect of message-handling overhead.)

Using an instrumented executable that collects traces during the execution of a program, we investigated how different frame allocation policies affect idle time, work performed, and the behavior of the heap. We compared three policies: a random policy of s03n64, a local-only policy of s100n1, and a distributed policy described by s33n3. We found that if no spreading or caching of I-structures is used, then even though the behavior of the three executions differs over time, the result is the same. This is due to the heap access pattern, which causes a single processor to become a bottleneck. If we turn on caching and spreading of I-structures, then the different policies perform differently. The random policy produces the most balanced distribution of frames, but is the slowest due to a high remote access rate. The distributed policy performs the best—it chooses the middle road.

On a completely different tack, the TL0 back end was used in evaluating two different machine architectures in [14]. We compared the J-Machine [11], developed at MIT as a study in universal mechanisms for fine-grained parallelism, and the CM-5, developed at Thinking Machines Corporation as a commercial product supporting data-parallel programs. A complete quantitative comparison is difficult because there are many variables that influence performance and little consensus on what constitutes a representative workload. However, by using the Id90 compiler, we were able to normalize for software effects, including programming language, programming style, and high-level compiler optimizations. TLC allowed us to produce an instrumented version of the program which produces roughly a hundred dynamic statistics. These are combined with the machine cost coefficients to obtain the average cycles per TAM instruction. The cost coefficients come from examining the instruction sequences generated by each TL0 translator.

We found that the dominant effect has little to do with parallelism and everything to do with traditional architecture issues such as average memory access time and floating point performance. Once we corrected for these, by creating new cost coefficients, the synthetic times for the two machines were surprisingly similar. The two main differences were in the message instructions and the control instructions. The J-Machine's fast message send and receive proved to be a clear gain. Yet the interaction between the scheduling levels causes control instructions to take more time on the J-machine than on the CM-5.

In the previous chapters we have shown how TL0 is a portable parallel assembly language. In order to gain this portability, we defined only those characteristics of TL0 needed for a compiler—leaving specific implementation decisions to the final translator. The main architectural features left unspecified include the size of the register set, the width of the data types, the implementation of the scheduling mechanisms, and the structure of the continuation vectors. On the other hand, we described the reasons for the constraints that are defined in TL0. In particular, we showed why inlets are constrained to using the reply network and how inlets must initiate only computations for which resources have been previously allocated.

This thesis has defined a concrete implementation of TL0, TLC. We explained the trade-offs involved in making the decisions that were postponed in TL0. In particular, we explained the alternatives for implementing the scheduling instructions and how the split-phase operations are realized. Finally, we explained both the sources of and means of eliminating the overhead introduced by using C as a target language.

In short, the TL0 back end has served its purpose well. It is currently being used as a compiler for Id90. It is also the workhorse for studying large programs in the large. This report fills a gap by defining the exact semantics of TL0. It should allow others both to build high-level language compilers that use TL0 as a target and to port TL0 to still other parallel machines.

Bibliography

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report CSG Memo 269, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, February 1987. Also in *Proc. of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.

[2] Boon. Runtime of Monsoon. *Journal of Parallel and Distributed Computing*, July 1993. Special Issue on Dataflow.

[3] P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, and D. V. Pryor. Vectorization of Monte-Carlo Particle Transport: An Architectural Study using the LANL Benchmark "Gamteb". In *Proc. Supercomputing '89*. IEEE Computer Society and ACM SIGARCH, New York, NY, November 1989.

[4] Soumen Chakrabarti, Abhiram Ranade, and Katherine Yelick. Randomized load balancing for tree structured computation. In *IEEE Scalable High Performance Computing Conference*, Oak Ridge, Tennessee, May 1994.

[5] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[6] D. E. Culler. Managing Parallelism and Resources in Scientific Dataflow Programs. Technical Report 446, MIT Lab for Comp. Sci., March 1990.

[7] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Empirical study of a dataflow language on the CM-5. To appear in IEEE Publication on Hamilton Island Workshop.

[8] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, pages 347–370, July 1993. Special Issue on Dataflow.

[9] Free Software Foundation, Cambridge, Massachusetts. *Using and Porting GNU CC*, 1993. Found on most systems by using emacs and meta-x info.

[10] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo, to appear, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, 1990.

[11] Michael D. Noakes, Deborah A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proc. of the 20th Int'l Symposium on Computer Architecture*, San Diego, CA, May 1993.

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

[13] Klaus Erik Schauser. Compiling Dataflow into Threads. Master's thesis, University of California, Berkeley, Computer Science Div., July 1991. Report No. UCB/CSD 91/644.

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

[15] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, August 1986. (MS Thesis, Dept. of EECS, MIT).

[16] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley.