TAM — A Compiler Controlled Threaded Abstract Machine

David E. Culler
Seth Copen Goldstein
Klaus Erik Schauser
Thorsten von Eicken
{tam@boing.CS.Berkeley.EDU}
Computer Science Division
University of California, Berkeley

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

This paper provides a complete description of TAM and codifies the model in terms of a pseudo machine language TL0. Issues in compilation from a high level parallel language to TL0 are discussed in general and specifically in regard to the Id90 language. The implementation of TL0 on the CM-5 multiprocessor is explained in detail. Using this implementation, a cost model is developed for the various TAM primitives. The TAM approach is evaluated on sizable Id90 programs on a 64 processor system. The scheduling hierarchy of quanta and threads is shown to provide substantial locality while tolerating long latencies. This allows the average thread scheduling cost to be extremely low.

1 Introduction

Dataflow execution models have evolved considerably since their original formulation[1, 15], reflecting an improved understanding of hardware implementation techniques[1, 18, 16, 20, 31, 36], parallel programming languages[27, 30], compilation methods[38, 43], and resource management strategies[9, 34]. Several hybrid models[5, 12, 23, 28] have been formulated which eliminate operand matching, avoid redundant synchronization, or use more conventional processor organizations. In addition, message driven models[14] demonstrate that the architecture need not dictate the format and handling of tokens, or rather, messages. The Threaded Abstract Machine (TAM) draws together these diverse investigations into a coherent execution model that can be implemented efficiently on a variety of machine architectures. It extends previous work by paying particular attention to 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 hierachy and allows the compiler to control the dynamic scheduling of computation.

TAM is easily understood independent 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 arbitrary parallelism can be represented.

TAM forms a bridge between traditional dataflow and control flow execution models. We have utilized TAM primarily as an intermediate step in compiling the high-level dataflow language Id90 to conventional parallel machines, including the Thinking Machines CM-5. Compiling down to TAM involves translating a dataflow graph into control flow among a collection of threads. A code-generator translates from TAM 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 TAM primitives, weighted by the frequency of use of these primitives in programs.

This paper provides a complete description of TAM as an interface between the high-level compilation process of parallel languages and the low-level code generation for emerging parallel machines. We begin by placing TAM in this broad context to understand how the issues it addresses arise from the combination of features desirable in parallel languages and the technology constraints on parallel machines. The description centers around an implementation of TAM called TL0 (Thread Language Zero). While some details of TL0 are influenced by Id90, the concepts it embodies are applicable to parallel languages and machines in general.

Consider for a moment the evolution of sequential machines and languages. Modern sequential programming languages allow the programmer to build arbitrary dynamic data structures and to realize sophisticated algorithms on these structures using complex control flow. In contrast, first-generation languages directly reflected the capabilities of the underlying hardware. Fortran, for example, did not support dynamic data structures or recursive control structures. The insistence on supporting these concepts in the Algol family of languages led to the development of abstract machines which demonstrated how to map the concepts to conventional hardware structures[25, 33]. This route proved more effective than supporting the language concepts directly in hardware[26].

By analogy, many current parallel languages remain close to the underlying machine, constraining the programmer to local data structures and static parallelism. For example, most extensions of Fortran and C with send/receive message passing provide exactly one thread of control per physical processor and a crude abstraction of communication channels. An emerging class of languages[7, 21, 24, 27] lets the programmer define arbitrary parallel data structures spread across the machine and dynamically spawn computations to perform coordinated actions on these data structures. Dataflow machines attempt to realize these concepts directly in hardware. TAM, instead, demonstrates how they can be mapped efficiently onto conventional parallel machines.

A second analogy can be drawn with sequential machines, where technological constraints enforced a consensus on the machine structure — pipelined, single-chip processors operating on a large register set. The RISC view held that the architecture should provide a set of efficient primitives, rather than general solutions, allowing the compiler to optimize for frequently occuring simple cases. For the forseeable future, parallel machines will consist of a number of processor-memory modules interconnected through a high-speed network. Processors operate directly on local memory and communicate with each other by transmitting messages through the network. Access to a remote memory location involves delivering a request to the remote processor associated with the memory and receiving a reply at some later time. Some machines will provide hardware support to accelerate the formatting, sending, and handling of certain messages, such as remote memory accesses, but the fundamental structure of the machine is unchanged. TAM defines simple primitives for initiating and handling communication events, allowing the compiler to optimize the use of these primitives in a manner consistent with the high-level language semantics.

Compiling for parallel machines is complicated because data structure accesses may involve long-latency communication and because multiple threads of control, at least one per processor, must be coordinated. To keep the utilization of each processor at an acceptable level, it is important to treat remote accesses as split-phase operations, that is, to continue executing instructions while the access completes[17, 35]. In some cases this can be accomplished using prefetching techniques, but in general multiplexing several logical threads of control onto each processor, called *multithreading*, is required. Coordinating threads of control on separate processors presents similar concerns, but the latencies involved are usually longer and potentially unbounded. The time for a response is determined by the logic of the program, rather than by hardware parameters. In general, it is necessary to provide multiple threads of control per processor to ensure forward progress, so it is natural to allow execution to proceed while events are pending. The challenge in multithreading is to keep the cost of thread switching low enough not to compromise the advantages. When a thread issues a remote reference it must be suspended and the next ready thread must be scheduled. When the reference completes it must synchronize with the computation to re-enable the waiting thread. Dataflow research has focused on the obvious costs: scheduling and synchronizing threads. However, optimizing scheduling costs while ignoring the effects on the storage hierarchy leads to unrealistic solutions. Instead, TAM exposes the scheduling of threads so that the compiler can coordinate scheduling with the usual management of the storage hierarchy. To aid in this coupling, TAM allows groups of related threads to be scheduled together. This reduces the cost of scheduling and permits the compiler to manage storage resources, *e.g.*, registers and local variables, across several threads. Finally, giving priority to related threads tends to improve cache behavior. Overall, the effect is that data can be kept at smaller and faster levels of the storage hierarchy.

To demonstrate the effectiveness of the TAM execution model, we have designed a threaded machine language, called TL0, and implemented a compilation path from Id90 to TL0. This uses the front-end of the MIT dataflow compiler, which produces program graphs[41] for the TTDA and Monsoon, with additional passes to partition the graph into threads and synthesize the control flow. The TL0 code is used as a machine independent intermediate form and can be input into a variety of code generators. To date, we have implemented code generators to translate into either C or machine code augmented with Active Messages[45] or conventional message passing as the network interface. The code generator described in this paper targets the CM-5 multiprocessor, consisting of Sparc processors with a memory mapped network interface.

The paper is organized as follows. Section 2 describes TAM in detail, emphasizing how the storage and scheduling hierarchies are exposed to the compiler. Section 3 discusses the mapping of high-level parallel languages to TAM. The compilation process of Id90 to TL0 is used as an example to show how the compiler can construct "storage directed" scheduling policies. Section 4 describes the implementation of TL0 on a CM-5 multiprocessor, focusing on the costs of scheduling and of accessing the network. These are combined with run-time statistics in Section 5 to demonstrate the effectiveness of the TAM scheduling hierarchy. Section 6 relates TAM to other parallel execution models and Section 7 summarizes the results and discusses open research problems.

2 The Threaded Abstract Machine

This section describes the TAM execution model. We begin with a general description of the model as a whole and then describe each aspect in more detail. The detailed descriptions can be skipped on a first reading. 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.

2.1 Concepts

A TAM program is a collection of *code-blocks*, where each code-block consists of several *threads* and *inlets*. Typically a code-block 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, 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 code-block involves allocating a frame, depositing arguments into it, and initiating execution of the code-block relative to the newly allocated frame. The caller does not suspend upon invoking a child code-block, so it 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 the unit of work distribution. 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 code-block means enabling threads of computation, where each thread is a simple sequence of instructions that cannot suspend. Each of the arguments to a code-block 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 containing 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 structure, but typically it 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 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 code-block contains a collection of inlets, which are compiler-generated message handlers for the remote accesses and call/return linkage. For example, arguments to a code-block invocation are sent to inlets of the code-block 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 dependences and control dependences, are enforced using synchronization counters within the frame. *Synchronizing threads* have an associated *entry count* which is decremented by forks and posts of the thread. The thread is enabled when the count reaches zero.

Observe that 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 clearly be kept in registers. In addition, whenever the compiler can determine that a collection of threads always execute in a single quantum, it can allocate values accessed by these threads to registers. As shown in Section 5 below, 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 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 does not cause the processor to stall, rather it is treated as a special form of message communication. A request is sent to the memory module containing the accessed location while threads continue to execute. The request specifies the frame and inlet that will handle the response. If the response returns during the issuing quantum, the inlet integrates the message into the on-going computation by depositing the value in a frame or register and enabling a thread. However, if a different frame is active when the response returns, the inlet deposits the value into the inactive frame and posts a thread in that frame without disturbing the register usage of the currently active frame. The global heap supports synchronization on an element-by-element basis, as with I-structures [4]. 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, activation frames, 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.

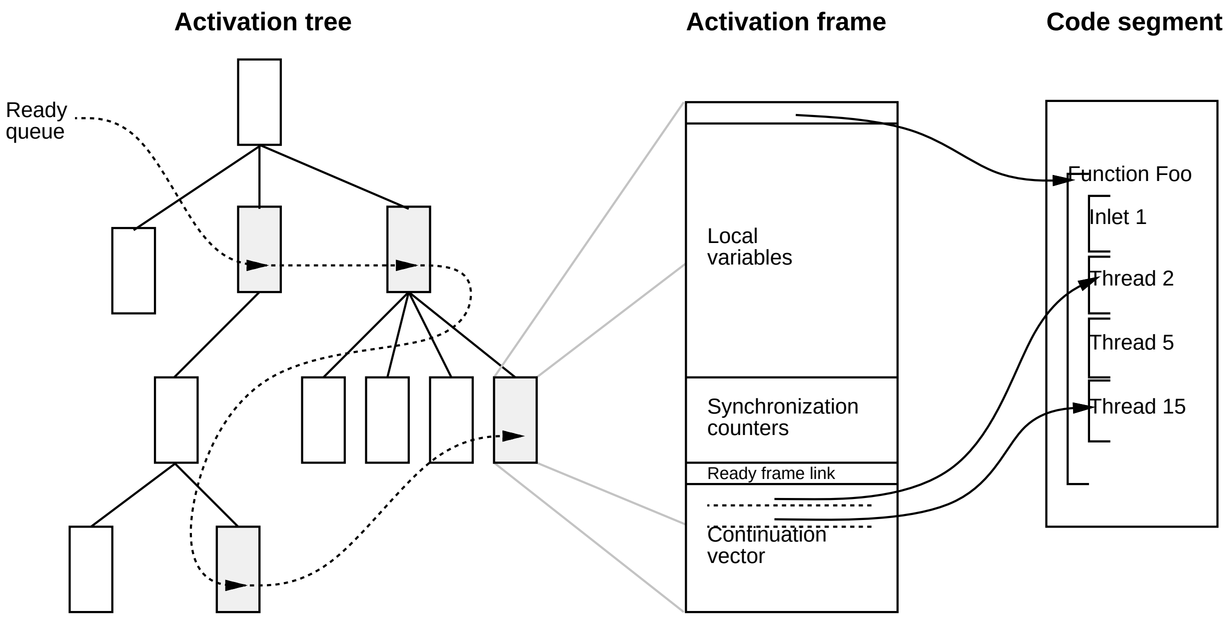

Figure 1: *TAM activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame, in addition to holding all local variables, contains counters used to synchronize threads and inlets, and provides space for the* continuation vector *— the addresses of 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 simply consists of pushing its instruction address into the continuation vector and sometimes linking the frame into the ready queue. Scheduling the next thread within the same activation is simply a pop-jump.*

The remainder of this section provides a more precise specification of TAM and its realization in TL0. It is included for completeness and as grounding for the empirical data presented later. However, it may be skimmed on first reading without compromising the main line of reasoning.

2.2 Storage Model

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

Frame storage is assumed to be distributed over processors, but each frame is local to some processor and only accessed from that processor.[1] Work is distributed over processors on a frame invocation basis.[2] Interframe communication is potentially interprocessor communication and is realized by sending values to inlets relative to the target frame (see §2.5).

A TAM processor contains contains data registers of various types and four special address registers: FP, the address of the current frame, IP, the address of the current thread instruction, IFP, the address of the target frame for the current inlet while it is executing, and IIP, the address of the current inlet instruction. A frame is *running* on a processor when it is referenced by the FP. Instructions can access registers or frame slots, relative to FP. In addition, the processor contains a *local continuation vector* (LCV) which holds addresses of threads forked within the current quantum. This can be viewed as a fast, short-lived extension of the *remote continuation vector* (RCV) in the frame, just as data registers extend the frame data storage. The processor switches from one activation frame to the next under instruction control, so there is no implicit management of registers. Typically, the register partitioning is realized by software convention.[3]

Heap storage contains objects that are not local to a code-block invocation, including statically and dynamically allocated arrays. Heap storage is assumed to be distributed over processors and is accessed through split-phase fetch and store operations, described below. In addition to data, each heap location holds a small number of tag bits, providing element-by-element synchronization as required for I-structures, M-structures, and thunks[4, 19].

In summary, the TAM data storage hierarchy is composed of three levels: registers, local frames, and heap-allocated structures. Registers are the least expensive to access, however their content is short-lived. Accessing locals from the frame is more expensive, essentially a local memory access, however, they persist throughout an invocation. Placement of data between frame and registers is under compiler control. The two-level scheduling policy increases the probability that frame accesses will be in the cache. Heap allocated structures are potentially in remote memory and are generally accessed through split-phase instructions with a corresponding inlet.

2.3 Program Structure

A TAM program is a collection of code-blocks and global structure definitions. A code-block contains declarations, inlets, and threads. Threads represent the computational body of the code-block and inlets specify the interface through which the code-block receives messages including arguments, return values, and heap-access responses. Each inlet is a message handler to receive a specific message. Threads and inlets are straight-line sequences of instructions.

Each code-block defines its frame layout and register usage. The frame layout includes: local variables, entry counters, the RCV, and an inter-frame scheduling data structure used to locate other ready frames. Synchronizing threads statically identify their entry counter, containing the number of remaining dependences. The compiler is responsible for initializing entry counters prior to the first fork or post of the corresponding thread. Typically, this is done in an initialization inlet and in threads forming a loop.

The RCV holds pointers to enabled threads while the frame is ready but not running. Logically, each enabled thread is described by a continuation comprising a frame pointer and an instruction pointer. However, since the continuation

[1] This condition can be weakened without disturbing the model if frame migration is supported directly in hardware. However, TAM does not require such sophisticated hardware support.

[2] Parallel iterations of a loop are treated as a restricted form of function call. Each chunk of iterations has an associated frame.

[3] Separate hardware register sets are permitted, but access to thread registers from inlets should be provided, otherwise additional restrictions must be placed on inlets.

is stored in its frame, rather than in some auxiliary scheduling storage, only the instruction pointer is recorded in the RCV. The scheduling queue is built by linking together frames, so the processor obtains the frame pointer when it switches to the frame, as discussed below. The size of the RCV is specified by the compiler and must be sufficient to hold the maximum number of concurrently enabled threads.

Initialized global structures are arrays or records which are allocated in the heap and initialized at program-load time. These are used, for example, to represent constant arrays and function closures over zero arguments.

2.3.1 Threads

A TAM thread is a sequence of instructions which executes from beginning to end without suspension or branching. It executes in the context of an activation frame accessible through FP. All control instructions (FORK, SWITCH, and STOP) are thread based. FORK attempts to enable a thread in the current code-block. SWITCH is a conditional fork to one of two threads. Any fork to a synchronizing thread involves decrementing the associated entry counter and enabling the thread if it reaches zero. A thread may contain zero, one, or many forks. Each thread is terminated by a STOP instruction, which causes the next enabled thread in the current frame to be executed.[4] Threads can also be enabled via interframe communication, as follows.

2.3.2 Inlets

An inlet is a sequence of instructions that handle a specific message, *i.e.*, the message format and message processing are coded in the instruction sequence of the handler. An inlet executes in the context of a frame specified in the message, via IFP. Typically, an inlet will receive the data of the message, store it into specific slots in the associated frame, and enable specific threads relative to that frame. Enabling a thread from an inlet, POST, is distinct from enabling one from a thread, FORK, and has a different optimization goal. The FORK enables computation that is closely related to the current processor state and attempts to maximize the coupling between the two threads. The POST may enable computation that is unrelated to the current processor state, so it tries to affect that state as little as possible. In addition, the POST is responsible for entering the frame into the scheduling data structure if the target frame had no enabled threads.

Inlets may preempt threads, but they may not preempt other inlets. (The model can easily be adapted to allow inlets to execute on a separate network processor with access to the frame store, but we will not consider that extension here.) To allow conditionals within inlets, an inlet may FORK or SWITCH to another inlet. The fork is handled just as within threads.

2.4 Execution Model

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

[4]Many researchers have come to use thread to mean a collection of instructions that executes without synchronization. Under this definition, a single instruction sequence may be part of different threads at different times. TAM threads are static and form a partition of the program. The dynamic quantity sought by the looser definition can be identified as a collection of TAM threads that are control dependent from a single synchronizing thread.

The TAM scheduling queue is a data structure obtained by linking together frames. The compiler defines the representation of this frame level structure by the code it places in the leave-thread. (TL0 provides a short-hand notation for simple scheduling structures, such as a list of frames per processor, to facilitate experimentation.) The compiler can also insert register saves in the leave thread and restores in the enter thread, if register values are carried across quanta.

The model intentionally does not specify the representation of the LCV. In translating TAM to a conventional machine, the LCV is simply a stack. The leave-thread address is placed at the bottom of the stack. FORK pushes an instruction address; STOP pops an address and jumps to it. Code generators will typically combine the last fork in a thread with the stop, producing a simple branch instead of push-pop-jump, as discussed in Section 4. A machine designed to execute TAM directly might represent the LCV as a queue to facilitate instruction prefetching on enabled threads. As discussed in Section 3, placing stronger constraints on the LCV implementation would allow more effective register usage.

Inlet execution may preempt the current thread when a message arrives, but certain TAM instructions must be performed atomically. The address of the inlet is loaded into IIP and the frame address specified in the message is loaded into IFP. Forked inlets have priority over the thread.

If the compiler can determine that two threads will execute in the same quantum, it can elect to carry values in registers from one to the other. Note, however, that the processor does not switch away from the running frame when inlets execute, so several threads may accumulate in the RCV before a ready frame runs. Also, split-phase operations may complete during the issuing quantum. Thus, the set of threads executed during a quantum may include many potential points of suspension.

2.4.1 Code-Block invocation

Invoking a code-block involves first allocating a frame. The caller sends arguments to inlets, established by convention, in the code-block relative to the newly allocated frame. The inlets are executed upon message arrival (possibly interrupting a thread on the processor holding the frame), store the values in the frame, and post threads of the code-block body for later execution. The activation thereby becomes *ready*, meaning that it has threads waiting to be executed, and it is linked into a pool of ready frames. Execution then continues with the interrupted thread. Eventually, the new frame is scheduled and its enabled threads are executed as described above.

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

2.5 Messages, inlets and atomicity

Communication between frames is performed by sending messages. A messages is sent to a specific inlet of a particular activation, identified by its frame address. The format of the message data is arbitrary. Although an inlet could in principle parse the message and dispatch on various alternatives, it is intended that the compiler produce specific handlers for specific formats so that no run-time message parsing is required.

The SEND operation packs a number of data values into a message and sends it to the inlet of the target activation. Execution then proceeds with the instructions following the SEND. When the message is received, processing of the current thread is interrupted and the receiving inlet is executed. After completing the inlet, processing continues with the interrupted thread.

To facilitate dispatching to the inlet quickly at message reception, the header of an incoming message contains the destination frame and inlet addresses. The context available to the inlet is the inlet registers and the locals of the receiving frame. The interrupted frame pointer (FP) and the inlet frame pointer (IFP) are also available. If the message is sent to the interrupted activation (*i.e.*, if IFP = FP) the thread registers can be used in the inlet to deliver the data into registers instead of frame slots. However, if IFP $\neq$ FP, accessing the thread registers will have unpredictable effects. In either case, certain locations in the interrupted frame may be accessed to link the new frame onto the scheduling queue.

To support the inlet model, an implementation of TAM must guarantee certain operations to be atomic relative to certain others. A SEND must be atomic relative to SENDs executed in inlets (*i.e.*, at interrupt time) to prevent generation of garbled messages. FORK (including the synchronization test), POST, SWITCH and STOP must be atomic relative to each other, since they require updates of synchronization counters and the RCV. Inlets are non-preemptive, so they are atomic relative to other inlets.

Threads initiate a heap access using IFETCH and ISTORE instructions. These are specialized forms of SEND that deliver a request message to the processor holding the accessed element and name the local inlet that is to handle the response. The requesting processor continues executing threads while the request is being serviced. The requests are typically handled by generic inlets that access the element and its synchronization bits. In some cases, such as fetches of an empty element, the inlet may in fact have to defer the response by enqueueing the request on the element. In all cases, a response is eventually returned to the inlet and frame specified in the request.

3 Compiling to TAM

The overall goal in compiling to TAM is to produce code that is latency tolerant, yet obtains processor efficiency and locality. TAM exposes parallelism, scheduling, and communication to the compiler and makes each cost explicit. Exposing the costs gives the compiler a clear optimization goal and allows it to map the various constructs of the parallel language to the best suited TAM primitives. On the other hand, TAM places the responsibility for correctly resolving several issues, such as management of frames, ordering of threads, and usage of local storage on the compiler. Although the source language for our compiler is the dataflow language Id90, the TAM parallel execution model is well suited for implementing other parallel languages. This section discusses the key aspects of the compilation process from a high-level parallel language down to TAM, including the representation of parallelism, communication, synchronization, scheduling, storage management, and the use of the storage hierarchy. These issues are addressed both in general and in the context of Id90.

3.1 A simple parallel program in TL0

To illustrate several of the compilation issues, we consider the following trivial program which computes the Fibonacci numbers. The source of parallelism is the recursive calls to `fib`. Arguments must be communicated to these parallel calls and the final result requires synchronization of the two partial results.

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

We will use the corresponding TL0 code, shown in Figure 2, to explain various compilation aspects of TAM. For this section only a high level understanding of TAM is required. A more detailed discussion of TL0 is provided in Section 4.

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

```
CBLOCK FIB.pc

FRAME_BODY RCV=3                             % frame layout, RCV size is 3 threads
        islot1.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
        pfslot0.pf jslot0.j                    % return frame pointer and inlet
REGISTER                                     % registers used
        breg0.b ireg0.i                        % boolean and integer temps

INLET 0                                      % recv parent frame ptr, return inlet, argument
        RECEIVE pfslot0.pf jslot0.j islot0.i
        FINIT                                  % initialize frame (RCV,my_fp,...)
        SET_ENTER 7.t                          % set enter-activation thread
        SET_LEAVE 8.t                          % set leave-activation thread
        POST 0.t "default"
        STOP
INLET 1                                      % receive frame pointer of first recursive call
        RECEIVE pfslot1.pf
        POST 3.t "default"
        STOP
INLET 2                                      % receive result of first call
        RECEIVE islot1.i
        POST 5.t "default"
        STOP
INLET 3                                      % receive frame pointer of second recursive call
        RECEIVE pfslot2.pf
        POST 4.t "default"
        STOP
INLET 4                                      % 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 recursive calls
        MOVE sslot0.s = 2.s                    % initialize synchronization counter
        FALLOC 1.j = FIB.pc "remote"           % spawn off on other processor
        FALLOC 3.j = FIB.pc "local"            % keep something to do locally
        STOP
THREAD 3                                     % got FP of first call, send its arg
        SUB ireg0.i = islot0.i 1.i             % argument for first call
        SEND pfslot1.pf[0.i] <- fp.pf 2.j ireg0.i % send it
        STOP
THREAD 4                                     % got FP of second call, send its arg
        SUB ireg0.i = islot0.i 2.i             % argument for second call
        SEND pfslot2.pf[0.i] <- fp.pf 4.j ireg0.i % send it
        STOP
THREAD 5 SYNC sslot0.s                       % got results from both calls (synchronize!)
        ADD ireg0.i = islot1.i islot2.i        % add results
        FORK 6.t
        STOP
THREAD 6                                     % done!
        SEND pfslot0.pf[jslot0.j] <- ireg0.i   % send result to parent
        FFREE fp.pf "default"                  % deallocate own frame
        SWAP "default"                         % swap to next activation
        STOP
THREAD 7                                     % enter-activation thread
        STOP                                   % no registers to restore...
THREAD 8                                     % leave-activation thread
        SWAP "default"                         % swap to next activation
        STOP                                   % no registers to save...
```

Figure 2: TL0 code for function `fib`

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. This example allocates one frame locally and one remotely. The FALLOC sends a requests 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 1 and 3, respectively. Assuming that the local allocation happens in-line, Inlet 3 is likely to interrupt Thread 2. Inlet 3 enables Thread 4 for execution. Therefore, at the end of Thread 2, f will continue with Thread 4, after which f will have no more enabled threads (unless the remote 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 1 will be triggered to receive a pointer to the remotely allocated frame into frame slot pfslot1. It posts Thread 3 using the default frame scheduling policy and enables the frame.

Thread 3 computes an argument value in a register and sends it to Inlet 0 of the frame for the first recursive call. The return frame pointer and return inlet are sent as well. 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 2. In the meantime, the argument message triggers Inlet 0 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 2 and 4, 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. We now consider these issues in greater depth.

3.2 Representation of Parallelism

Parallel languages provide a variety of ways to express parallelism, *e.g.*, function calls, loops, co-routines, tasks, or futures. The coarsest grain of parallelism is represented in TAM by frames, which can be distributed over processors. Finer grain parallelism within a frame is represented by threads, which can be used to mask communication latency. Lastly, instruction level parallelism can be exploited within a thread. The compiler must manage the parallelism in the program by mapping it to the appropriate TAM level.

The preceding description of TAM tacitly assumes parallelism is expressed in terms of some form of parallel call, however, all other forms of parallelism can also be represented using the frame mechanism. The parallel call is challenging because it can generate an arbitrary amount of parallelism, as in the unfolding of the call tree in `fib`. This must be mapped onto a fixed set of physical processors. The chunk of work associated with a frame need not correspond exactly to a user-defined function in the program: it may be desirable to execute individual expressions in parallel, as with futures[24], or to inline several calls to use a single frame. Frames could also represent the state of individual tasks, communicating through messages. The dynamic allocation of frames implies that the task structure need not be static. In fact, non-strict functional languages[42] behave at the TAM level like co-operating processes, since the child may need to return certain results in order for the parent to make further progress and deliver additional arguments.

At the other extreme, parallelism may be limited to a single loop. In this case, a set of frames each holding the local variables of a different iteration of the loop body may be allocated on a set of processors. Under TAM, this can be easily extended to handle nested loop parallelism, where each loop frame maintains several frames for iterations of the inner loop[32]. The assignment of iterations to frames can be addressed by a variety of policies. A very general form of parallel loop structure, called k-bounded loops[11], is used in compiling Id90. In this scheme, the amount of parallelism, *i.e.*, the number of frames, is determined at the time the loop is invoked, possibly depending on values within the program. The loop builds a ring of k frames and cycles through them. Each iteration detects its own completion and sends a signal to the previous frame indicating that the frame is ready for the next iteration. Nested loops produce a ring of rings.

Allocating a frame for a chunk of computation does not guarantee that the computation will operate in parallel with other frames, but gives it the opportunity to do so. For example, two frames could co-routine by exchanging messages. The Id90 compiler aggressively exploits parallelism in function calls and loops by allocating frames.

When allocating a frame, it must either be allocated locally or on another processor. TAM provides mechanisms to control the placement of frames, but does not dictate how to use it. Thus, the compiler or run-time system must allocate frames in a manner that provides adequate load balancing. For highly irregular parallel problems it is difficult for the compiler to determine the mapping statically, so dynamic load balancing techniques provided by the run-time system are needed.

3.3 Frame Storage management

The representation of dynamic parallelism presents a fundamental storage management challenge. TAM allows storage management to be addressed within the specific structure of the high-level language, but dictates that it will take place in terms of explicit allocation and release of frames. The size of the frame is fixed at the time of allocation. This should be contrasted with providing an arbitrary collection of stacks, as with typical "threads packages". If each frame were provided with a stack, a large chunk of the address space would need to be provided per frame. Parallel languages typically generate a large amount of tightly controlled parallelism, so the generality of an arbitrary collection of stacks is not required. Under TAM, the compiler can (and must) map these specific parallel structures onto frames. For example, recursive call structures are supported by allocating multiple frames. The linkage between frames is expressed in terms of messages sent to inlets in the target frame.

The management of frames is typically integrated with the calling convention. In a sequential language, arguments and results are deposited in predefined locations on the stack, or passed in registers. In TAM, argument and result passing is represented in terms of inter-frame communication. The caller sends arguments to predefined inlets of the callee and the callee sends results back to inlets specified by the caller. Two additional arguments are passed to the callee: the parent frame pointer and the return inlet number. If tail call optimization is performed, the caller can pass it's own return frame pointer and inlet directly to the callee. The Id90 compiler augments each function with code to detect the completion of all computation, so that the frame is released by the last thread in the code-block (*cf.* Thread 6 in the `fib` example above). K-bounded loops detect completion of each iteration and include additional code to release the ring of frames when the entire loop finishes.

3.4 Communication

Sharing of information and coordination of activity among portions of the program are represented in TAM via sends to inlets and heap requests delivered to inlets. This encourages a latency tolerant style of code-generation. When a remote access is initiated, the computation continues; the response will be received asynchronously and will enable whatever computation depends on it. The execution model places no limit on the number of outstanding messages, although architectural factors such as latency, overhead, or available bandwidth may introduce a practical limit[10]. The communication model is efficient because no buffering is required in the communication layer. Storage to receive the message is pre-allocated in the frame so that the inlet can move the data directly from the network interface to the frame[45].

The Id90 compiler generates a specialized message handler for each heap reference, function argument, and result in the program text. This specialization reduces the message size, since the message format is encoded in the inlet, and reduces the cost of message handling, since no parsing is required.

Currently, remote references to global data structures are handled by generic remote reference message handlers. These handlers are executed on the processor on which the accessed data element resides. The handlers perform synchronization if needed, access the data element, and send the reply back. In our compilation scheme for Id90, the only case where a message handler may need buffering is when a read must be deferred: a continuation consisting of the requesting processor, inlet, and frame is enqueued on the element so that the read can be completed when the data is written.

There are various techniques the compiler can employ to make the code latency tolerant. By issuing several remote requests in the same thread, the latency of multiple requests can be overlapped. By issuing remote requests as early as possible, the latency can be overlapped with computation not dependent on the reply. This non-speculative form of prefetching can be achieved by pulling remote references as far up in a thread as possible, and by ensuring that threads with remote references get scheduled first. In both cases there must be adequate parallelism available to overlap the communication latency with computation. If the program does not not have enough parallelism, techniques such as loop unrolling may be applied to introduce it. In general, the more parallelism a program exploits, the more storage resources it needs. Thus a tradeoff has to be struck between good latency tolerance and storage requirements[13].

3.5 Storage hierarchy

Compilers for sequential languages manage the placement of data between registers and call frame locations based on the analysis of a single flow of control. The basic technique is to construct an interference graph describing which variables may have overlapping lifetimes. Variables are then assigned to available registers with priority given to the ones most frequently used. The compiler uses all the registers that are available. Depending on the calling convention, it may be necessary to save variables to the call frame across calls. Register allocation is more involved under TAM because the ordering among threads can be affected by the order of message arrivals, as well as the evaluation of conditionals. The analysis must track the interleaving of multiple flows of control which causes the interference graph to be much denser. Furthermore, establishing a large register footprint is at odds with fast frame scheduling.

The compiler can easily allocate values to registers within a thread and across threads that are provably within the same quantum. If it allocates values to registers that cross possible suspension points, than it can use the enter and leave threads to save and restore register values between quanta. In the function `fib` above, registers are only used where it is possible to statically determine that the threads execute in the same quanta. For example, ireg0 is defined in Thread 1 or Thread 5 and is used in Thread 6. Given that the second recursive call is allocated locally, Thread 2 (allocating the frame) and Thread 4 (sending the argument) will typically execute in the same quantum. This would allow islot1 to be kept in a register[5].

Processor efficiency can be improved by sending messages directly out of the processor registers into the network and by receiving messages into registers. For example, the message handlers that receive the frame pointers for the

[5] The current compiler for which statistics are presented in Section 5 does not yet perform these optimizations.

recursive calls in `fib` (inlet 2 and inlet 4) may check to see whether the caller frame is currently running. If so, the callee frame pointer can be placed directly into a register instead of into the caller frame. As a consequence, the callee frame pointers can be accessed from registers by the threads that use them for sending the arguments. In the case that the caller frame is not currently running, the inlet will deposit the callee frame pointers into the frame, and the enter thread will load them into registers when the frame gets scheduled.

3.6 Synchronization

For sequential languages, the instruction ordering and control flow produced by the compiler ensures that all forms of data and control dependencies are enforced. In a parallel setting, additional explicit synchronization is needed, especially when non-blocking remote communication is use. For example, a computational thread has to synchronize with the reply of a remote request before using its value. Similarly, a thread that needs values produced by two other threads, has to synchronize on both threads having completed. Dataflow machines have both forms of synchronization built into their execution model. Explicit control flow in TAM provides two ways to enforce data and control dependencies. Inside a thread, the linear ordering of instructions determines their execution order. Across threads, the use of explicit synchronization counters and control primitives, such as FORK, SWITCH, and POST, specifies the order in which the threads get executed and allows computation to synchronize with communication. The compiler orders the instructions within a thread, and initializes the synchronization counters, such that all data and control dependencies are enforced. This is straight-forward for static dataflow graphs, where all dependencies are visible at compile time and no cycles exist[37].

3.7 From dataflow graphs to threads

Non-strict languages, such as Id90, introduce an additional compilation issue. In general, any input to a function, including arguments, heap access responses, and results returned from subordinate calls, can potentially depend in some manner upon an output of the function, including returned results, stores to heap locations, and arguments passed to subordinate calls. (Somewhere outside the function body, some output may be used in deriving an input to the function.) These dependencies may go through any number of levels of indirection and usually cannot be identified at compile time. Thus, the task of the compiler in partitioning the dataflow graph into threads is to prove where such external dependencies cannot exist.

There are three basic partitioning techniques: certain dependence, dependence sets, and demand sets. If a node u is connected by certain dependence arcs to node v, then u must precede v. There can be no hidden dependence in the reverse direction or the original dataflow graph would deadlock. If there is no certain dependence between a pair of nodes, they are apparently independent and we must prove that one cannot depend on the other through some external effect. This is accomplished by examining how portions of the graph interact with the external interface. Dependence analysis uses sets of inputs to establish independence. Given two apparently independent nodes in the dataflow graph, if they both depend unconditionally on the same set of inputs, neither can depend on the other, since the dependence would need to be conveyed through one of the inputs[22]. Demand analysis uses sets of outputs to establish independence. If two apparently independent nodes unconditionally affect the same set of outputs then neither can depend on the other, since the dependence would have to be conveyed through one of the outputs[38]. The compiler repeatedly reduces the dataflow graph into macro nodes representing threads using analysis of dependence sets (input nodes on which a node depends) and demand sets (output nodes that depend on the node) to drive the reduction process. Recent work shows how this analysis can be carried out globally[44]. Improvements in partitioning increase the thread length, decrease the number of dynamic scheduling events, and reduce the total number of synchronizations by eliminating redundant ones.

4 Implementation of TAM

This section describes an implementation of TAM on the CM-5 multiprocessor[40]. The discussion is centered around the thread language TL0 which takes a position on many of the alternatives left open in TAM by defining an instruction set with precise semantics. TL0 is a machine independent assembly language for TAM and the concrete target for the compilation process described in the previous section. By dividing up the compilation process into two separate phases, from Id90 to TL0 and then from TL0 to native code, we isolate high level compilation issues from the specific hardware support for threaded execution.

A TL0 program, like the one in Figure 2, is composed of code-blocks consisting of the activation frame layout, registers, and the code for the threads and inlets which execute relative to the frame. Each frame slot and register is statically typed and reuse across types is not possible. The TL0 storage hierarchy consists of an unlimited number of machine registers, frame storage and the global heap. TL0 instructions can operate directly on registers or on the activation frame. TL0 has five different instruction categories.

ALU instructions use a standard three-address format and operate on typed variables in registers and the frame.

Network access is provided by SEND and RECEIVE instructions. SEND is used in threads to send an arbitrary number of values to an inlet of another frame. Receive is the first instruction in an inlet and stores the message data fields into frame slots or registers.

Thread control is expressed using FORK and SWITCH instructions, and each thread is terminated by a STOP instruction. TL0 uses boolean variables to represent the result of a comparison; which is then used by the SWITCH instruction.

Frame scheduling is expressed using POST and SWAP. Although it is possible to generate the code that explicitly manages frame allocation and scheduling, currently, a small set of "high-level" instructions are used to facilitate experimentation. All of the "high-level" instructions take a *policy* argument which conveys the compiler's intent to the code generator.

Heap access is provided via a variety of fetch and store instructions. Conceptually, these instructions simply send a message to the memory controller holding the designated location. The response for a fetch is received by an inlet, but there are no explicit acknowledgments of stores.

The remainder of this section presents the mapping of the storage model and the implementation of these instruction categories on the CM-5 processor by describing the optimizations performed and the cost in terms of instruction and cycle counts for the most interesting TL0 instructions. Section 5 combines them with instruction frequency statistics to draw a comprehensive picture of the effectiveness of TAM.

4.1 TL0 on the CM-5 multiprocessor

The CM-5 is a massively parallel MIMD computer based on the Sparc processor. Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte direct-mapped write-through cache), 8 MBytes of local DRAM memory and a network interface. The nodes are interconnected in two identical disjoint "hypertrees" (also described as an incomplete fat tree), and a broadcast/scan/prefix control network.[6]

[6]Each node may also contain vector units which we do not address in this paper.

Operand (32-bits) location	access costs instr.	cycles
Register	0	0
Constant	0–2	0–2
Cache	1	2–3
DRAM	1	20†

†: 32-byte cache-line refill

Table 1: *Access cost to each level of the local storage hierarchy on a CM-5 node.*

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

Table 2: *Reserved special-purpose registers hold important variables and constants used by the TL0 implementation to provide fast scheduling of computation and network access.*

4.1.1 Storage model

Mapping the TL0 storage hierarchy onto the CM-5 is relatively straight-forward. TL0 registers are mapped onto Sparc registers as described below. Activation frames are allocated in local memory and are expected to reside mostly in the cache. The heap is divided into two regions, one for small arrays which are allocated local to a node and the other for large arrays which are spread across the nodes such that logically consecutive elements are mapped onto different processors. Program code is placed on every processor.

Since TL0 does not limit the number of available registers, it is the responsibility of the code generator to spill excess TL0 registers to the activation frame. TL0 instructions allow frame relative addressing. To accommodate the Sparc instruction set, operands to instructions residing in the frame must be temporarily loaded into registers. Table 1 summarizes the cost of accessing operands at the various levels of the storage hierarchy. In all of the remaining sections the cost of TL0 expansions are based on the frame operands already being in registers. Section 5 presents the program dependent cost of bringing operands from the frame into the registers.

The TL0 registers are implemented on the CM-5 as a flat register file in a single register window. Register windows cannot be used between function invocations because the computation unfolds as a tree rather than as a stack.[7] Further, due to the tight coupling between threads and inlets it proves to be more efficient to partition a single window, than for inlets to run in a second window. The single register window is divided into three categories: special-function registers, thread registers and inlet registers. The special-purpose registers (g0–g7), as shown in Table 2, hold important variables and constants used by the TL0 implementation. The TL0 IP and IIP registers are both mapped to the Sparc PC register. There are sixteen thread registers (i0–i7 and l0–l7) which are fully under control of the register allocator. The eight inlet registers (o0–o7) are generally reserved for inlets but may be used by the register allocator between successive network polls to hold thread temporaries. The RECEIVE instruction at the beginning of an inlet typically

[7] Using the multiple windows to cache the registers of several activations is not an efficient alternative since switching among register sets requires a kernel trap.

moves the message from the network interface FIFOs into inlet registers and uses the IFP to store the message data into the frame locations.

4.1.2 Arithmetic and logic instructions

Once the operands have been loaded into registers most TL0 arithmetic and logical operations map into a single machine instruction. A few instructions such as ABS, MAX, MIN require short instruction sequences and integer divide, and multiply are implemented by calling a library routine. Table 3 summarizes the costs of the basic instructions.

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

Table 3: *Mapping of TL0 arithmetic and logic instructions to the Sparc.*

4.1.3 Sending messages

The TL0 SEND instruction can send a message of arbitrary length to an inlet of another frame. The CM-5 implementation limits the message to three 32-bit words of arguments and uses the first two words of the message for the frame pointer and the inlet start address. The code generator will convert a SEND of a longer messages into multiple sends. Since each SEND is paired with an inlet, the code generator also creates new inlets, each of which receives a piece of the original message. The new inlets all synchronize before executing any code dependent on reception of the logical message.

The network interface (NI) is attached to the node MBUS and consists of a pair of memory mapped FIFO queues for each of the two data networks. Accessing the network interface costs 7 cycles for 32-bit accesses and 8 cycles for 64-bit accesses. The implementation of the SEND pushes the message into the outgoing FIFO using store-double instructions. The cost of SEND is shown in Table 4. Note that a global register is reserved to point to the base address of the memory mapped NI. After pushing the message into the FIFO, a status register in the NI indicates whether the message was accepted. The NI discards the message if the network is backed-up, and the message must be resent explicitly. The cost of a SEND is relatively high because access to the NI requires uncached loads and stores. For this reason, sends to the local node are special-cased in software, even though the CM-5 hardware supports loop-back.

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

Table 4: *Costs for sending a message limited to three 32-bit words of arguments. Access to the network interface involves uncached loads and stores which take 7–8 cycles each.*

The CM-5 has two identical disjoint networks. The two networks are used separately to avoid deadlock: all messages sent from thread level use one of the networks and the other network is reserved for replies sent back from inlets. This has the consequence that if the outgoing network is backed-up, then a send at thread level must accept incoming messages on both networks and that sends at inlet level (which are really replies) must accept incoming messages only on the reply network.

4.1.4 Receiving messages

In TL0, when a message is received an inlet is invoked. The first instruction of the inlet is a RECEIVE, which specifies the frame slots where the message data is to be stored. On the CM-5 the arrival of a message can be detected either by enabling message interrupts or by polling the network interface regularly. Dispatching a message interrupt into the user program incurs approximately 140 cycles of overhead. Although multiple messages can typically be received during one invocation of the interrupt handler, thereby amortizing the overhead over several messages, the overhead is still high. Furthermore, the cost of the FORK, SWAP, and I-structure operations would increase if message interrupts are used, due to maintaining atomicity in accessing synchronization variables, the frame queue, and local heap structures. The strategy employed in the CM-5 implementation is to explicitly poll the network once in every thread. If the thread contains an instruction which might access the network, then the poll is combined with that instruction. All other threads have an explicit poll inserted at the end of the thread. If a message has arrived, the appropriate inlet is called. Table 5 shows the cost of polling the network and the cost of running an inlet.

Operation	costs	
	instr.	cycles
Explicit poll	3	9
Poll as part of a send	2	2
Message handling		
Inlet overhead	6	13
Receive 32-bit word	$1\frac{1}{2}$	6

Table 5: *Cost of polling the network and of running an inlet. The inlet overhead includes dispatching to the inlet and returning from the inlet. The receive cost refers to transferring message data into the activation frame.*

4.1.5 Thread scheduling

In TL0, thread control is realized by the FORK, SWITCH, and STOP instructions. The distinction between synchronizing and non-synchronizing threads is indicated by a SYNC statement placed at the beginning of synchronizing threads. The SYNC statement contains the name of the synchronization variable; which is initialized by setting its value to the appropriate entry count before any attempt is made to fork the thread. Although the SYNC declaration is placed at the beginning of the thread, the synchronization test (decrement and test for zero) is performed as part of the FORK instruction. Non-synchronizing FORKs do not require the decrement and test and thus are cheaper than synchronizing ones.

Conditional control flow is implemented in TL0 through compare instructions which set a boolean variable and a SWITCH instruction which forks one of two threads depending on a boolean. For the Sparc, the code generator attempts to allocate the result of the compare to the condition-code register and to propagate the condition to all SWITCH instructions based on that comparison[8]. The SWITCH itself is essentially translated into an if-then-else around two FORKs.

[8] No optimization is required when mapping these TL0 instructions to architectures without conditions codes, *e.g.*, the MIPS or Motorola 88k.

The code generator specializes the last FORK in a thread into a fall through or branch, which eliminates the STOP at the end of the thread. The remaining FORK instructions translate into a push onto the LCV. A STOP ends a thread by popping the next thread from the LCV and jumping to it. A summary of the instruction and cycle counts involved in the various cases is shown in Table 6.

Operation	costs	
	instr.	cycles
Fork a thread		
Fall through	0	0
Branch to thread		
unsynchronizing	1	1
successful sync.	3	4
unsuccessful sync.	4	8†
Push thread onto LCV		
unsynchronizing	3	5
successful sync.	6	10
unsuccessful sync.	4	7
Switch one of two threads	fork+2	fork+2
Stop and pop thread from LCV	3	5
Initialize sync. counter	2	4

Table 6: *Cost of TL0 thread synchronization and scheduling instructions. For* FORK *several variations are shown, depending on whether a* FORK *can be combined with a stop and optimized into a branch, whether the target thread is synchronizing and whether the synchronization was successful or not.* †*: The cost of unsuccessful sync. branch does not include the cost of the* STOP *that is executed to end the thread since the synchronization fails.*

The LCV is implemented as a stack of 16-bit offsets from the current code-block base (kept in a register) to the beginning of the enabled threads. A push onto the LCV consists of three instructions: setting the offset, storing it into the LCV and incrementing the top of LCV pointer which is kept in a register. The pop-jump for a STOP adds the offset to the code-block base as part of the Sparc jump instruction. The bottom-most offset on the LCV always points to the activation's leave thread, which is responsible for switching to the next frame.

4.1.6 Frame scheduling

In TL0 the details of frame allocation and scheduling operations are hidden from the Id90 compiler behind a small number of high-level instructions. Exposing the details for a specific target machine would be straight-forward, but doing so in a machine independent manner seems difficult. To give the compiler control over frame allocation and scheduling, the appropriate TL0 instructions take a "policy" argument which conveys the compiler's intent to the code generator. The policies are arbitrary names (strings) on which the two parties agree. Currently, the compiler uses *default*, *local*, *remote*, and *cyclic* frame allocation policies and *default*, *fifo* and *lifo* frame scheduling policies.

FALLOC instruction is an example of the high-level instructions that take a policy. It allocates the frame for a new activation and passes a number of arguments to its inlet 0. The choice of processor is controlled by the policy attached to the instruction. Instead, the compiler could directly output the TL0 code which would choose the processor, allocate a new frame, and finally, send a message to inlet 0 of the new frame. The FFREE instruction deallocates a frame, possibly the current frame. Typically, this is followed by a SWAP which terminates the current activation.

For the Sparc, the RCV is implemented similarly to the LCV using 16-bit offsets. The pointer to the top of the RCV is kept in the frame, not in a register. Initially, the RCV is empty and the frame is not part of the scheduling queue. As messages for the activation arrive, inlets are executed and enable threads into the RCV using the POST instruction.

The cost of a POST (shown in Table 7) is generally higher than that of a FORK and depends not only on whether the target thread is synchronizing or not, but also on the state of the frame. If the frame is idle (*i.e.*, it has no threads in its RCV), then it will have to be enqueued onto the ready queue. In addition for both idle and ready frames, the cost of manipulating the pointer to the top of the RCV is higher than for the LCV since it is in the frame, not a register.

Operation	costs	
	instr.	cycles
Post a thread from inlet		
Idle frame		
unsynchronizing	12	18
successful sync.	15	23
unsuccessful sync.	4	7
Ready frame		
unsynchronizing	9	14
successful sync.	12	19
unsuccessful sync.	4	7
Running frame		
unsynchronizing	5	7
successful sync.	8	12
unsuccessful sync.	4	7
Swap to next frame		
first 3 threads	14	26
per extra 4 threads	6	12

Table 7: *Cost of TL0 frame synchronization and scheduling operations.*

If the target thread is for the running frame, then instead of pushing onto the RCV, the POST instruction can push the thread onto the LCV. Thus, for the cost of a compare between the FP and IFP, the cost of a POST can be brought down to that of a FORK and remote requests which return during the issuing quantum save on the cost of SWAPs and POSTs.

At the end of a quantum the leave thread of the activation is executed; it is responsible for switching to the next frame. In TL0 the SWAP instruction selects the next frame according to a specified policy, places the new frame's leave thread as a sentinel at the bottom of the LCV, copies the RCV onto the LCV (4 threads at a time using double-word loads and stores) and jumps to the enter thread.

4.1.7 Heap access

Implementing synchronizing data structures (such as I-structures and M-structures) presents three challenges: representing the presence bits, checking and updating their state on every access, and maintaining the lists of deferred reads. The Id90 type system requires that each structure element is represented by 64 data bits and 3 presence bits, which must be simulated in software. For each 8-byte I-structure element one tag byte is allocated. The tags are stored in a memory area disjoint from the data area. A global register (Izero) holds the offset from data address zero to tag address zero. I-structure addresses are represented such that the tags of element N are stored in the byte at location N and the data is stored in the 8 bytes starting at location $8N$+Izero. All I-structure accesses must check the tag byte before reading or writing the data. Deferred reads are enqueued as a linked list, the head of which is stored in the structure element. Each link in the list holds the node, inlet, and frame information necessary to satisfy the read when a write to that element occurs. When initializing and allocating I-structures the tags of eight elements can be initialized using a single store-double instruction.

TL0 provides a special syntax for issuing remote references (*e.g.*, IFETCH and ISTORE). Each instruction specifies the base and offset of the I-structure being accessed. The expansion first calculates the node and address of the element being accessed. Then, the expansion determines if the access is a local access and, if so, performs it inline. Otherwise, a request is sent to the node that contains the element. The different cases are reflected in the costs specified in Table 8.

Operation	costs	
	instr.	cycles
I-structure fetch		
Local, data present	8	11
Local, data not-present	25	58
Remote		
Initiate request	18	38
Service, data present	29	91
Service, data not-present	39	115
I-structure store		
Local, no waiting fetches	9	15
Local, waiting fetches	18	30
Remote		
Initiate request	18	38
Service	13	44
I-structure allocate (N words)	$5 + 4\lceil\frac{N}{8}\rceil$	$6 + 7\lceil\frac{N}{8}\rceil$

Table 8: *Access to global data structures with synchronization on a per-element basis. Local fetches are special-cased. The entries for remote requests include the cost of the request send and the receive by the serving node. The remote service numbers include the cost of the reply and the cost of starting the overhead for the inlet that receives the reply. The cost of a fetch of an empty element includes the cost of both enqueueing a continuation on the element and for fulfilling the request when the write occurs allowing the request to be satisfied. I-structure allocation includes initializing all tags to empty, but the time spent in the memory manager for allocation is not included.*

4.2 Discussion

This section has shown that the scheduling costs at the two levels of the hierarchy are distinct. Most thread scheduling can be optimized away and the remainder costs a few cycles each. Not well represented in the cycles counts are the costs due to the memory hierarchy. On the Sparc processor used in the CM-5, loads and stores take multiple cycles each. On the next generation processors, accesses to the LCV will be cached and thus execute in a single cycle, whereas accesses to the RCV are less likely to be cached and thus will cost more. The cost difference between thread and frame scheduling is thus likely to increase. The cost of thread scheduling can be further reduced by taking advantage of the prefetch possibilities offered by the LCV. A thread ending in a pop-jump could load the next thread address early-on and pass the address to the instruction prefetch unit.

The cost tables in this section show clearly that communication is still rather expensive. Sending an argument to a remote frame costs about fifty processor cycles while accessing a remote heap location costs on the order of a hundred (five cache misses). The main cause for this high cost on the CM-5 is the slow access to the network interface across the node memory bus. Placing the NI on the cache bus or integrating it into a co-processor, for example as proposed in the MIT/Motorola *T project[29], would reduce the overhead of messages considerably.

5 Dynamic Measurements

In this section we present empirical data based on the implementation of TAM described above to validate the TAM approach and its effectiveness on current parallel machines. The data is obtained on the CM-5 multiprocessor using a version of the code-generator which inserts a few instructions into the threads and inlets to collect roughly one hundred dynamic statistics on each processor. At the end of the program, the overall counts are accumulated. This provides TL0-level dynamic instruction frequencies, which characterize the requirements of Id90 programs. The frequency of dynamic scheduling events is obtained as well. We see that the TAM scheduling hierarchy is indeed effective in grouping together a sizable number of threads, which reduces the cost of thread scheduling and improves the use of the storage hierarchy. Applying the instruction costs presented in Section 4, we can estimate the fraction of execution time devoted to each aspect of program execution.

5.1 TL0 instruction frequency

Figure 3 shows the dynamic frequency of the basic TL0 instruction categories on six benchmark programs ranging from 50 to 1,100 lines. These were developed by other researchers in the context of other platforms, especially the GITA dataflow graph interpreter and the Monsoon dataflow machine. *QS* is a simple quick-sort using accumulation lists. The input is a list of random numbers. *Gamteb* is a Monte Carlo neutron transport code[6]. It is highly recursive with many conditionals. The work associated with a particle is unpredictable, since particles may be absorbed or scattered due to collisions with various materials, or may split into multiple particles. Splitting is handled by recursive calls to the trace particle routine. Particles are independent, but statistics from all particle traces are combined into a set of histograms represented as M-structures. The input consists of 8192 initial particles. *Paraffins*[3] enumerates the distinct isomers of paraffins. *Simple*[2, 8] is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id90. It integrates the solution to several PDEs forward in time over a collection of roughly 25 large rectangular grids. Each iteration consists of several distinct phases that address various aspects of the hydrodynamics and heat conduction. Simple is irregular, due partly to the relationship between the phases, which traverse the data structures in different ways. In addition, table look-ups are performed inside of the grid-point calculation and boundaries are handled specially. The problem size is a 128x128 grid. *Speech* determines cepstral coefficients for speech processing. *MMT* is a simple matrix operation test using 4x4 blocks; two double precision identity matrices of size 60x60 are created, multiplied, and subtracted from a third.

The programs toward the left of Figure 3 exhibit very fine-grain parallelism. Observe that they are control intensive. The moderate blocking and regular structure of MMT shows a significant contrast. We will focus primarily on the two large programs, Gamteb and Simple, as these include a variety of usage patterns and exhibit significantly different instruction mixes.

Figure 4 shows the speedup obtained on the CM-5 for the two application benchmarks. In both applications we observe a linear speedup beyond a small number of processors. At 64 processors the performance is comparable to that on a 16-node Monsoon configuration (8 processor nodes and 8 I-structure nodes). Although communication latency is tolerated, roughly half the processor is lost to message handling overhead in Gamteb and three-quarters in Simple. The difference is attributable to the remote reference rates in the two programs, as discussed below. On the high end, this correlates with the speedup obtained on the full machine; on the low end it correlates with the number of processors required before any significant speedup is obtained. Although the programs could be tuned to obtain better performance on this particular machine, our goal is to evaluate TAM in the regime of implicit parallelism and implicit data placement. In what follows, we examine the execution behavior in more detail.

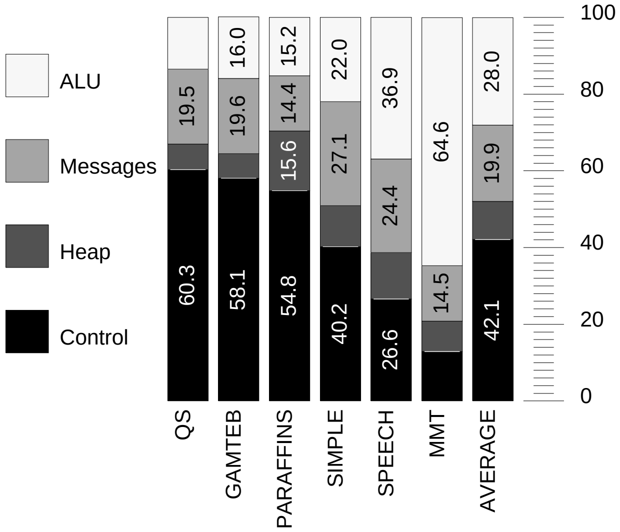

Figure 3: *Dynamic instruction mix statistics for several benchmark programs.* ALU *includes integer and floating-point arithmetic,* messages *includes instructions executed to handle messages,* heap *includes global I-structure and M-structure accesses, and* control *represents all control-flow instructions including moves to initialize synchronization counters. The final column shows the arithmetic mean of the distributions.*

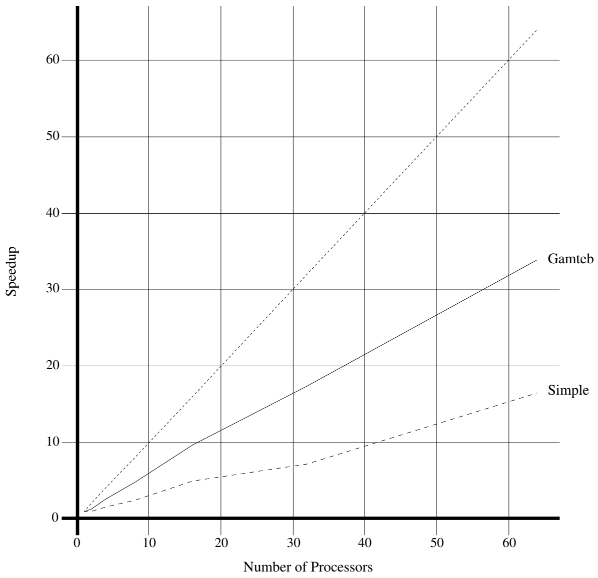

Figure 4: Speedup for Gamteb and Simple from one to sixty four processors on the CM-5

	QS	Gamteb	Paraffins	Simple	Speech	MMT
Ave TL0 Insts. per Thread	2.6	3.2	3.1	5.3	6.3	17.6
Threads per Quanta	11.5	13.5	215.5	7.5	16.7	530.0
RCV Size when Scheduled	1.1	1.6	1.3	1.4	1.0	1.6
Threads forked during Quantum	8.8	10.2	168.4	4.1	11.7	406.6
Threads posted during Quantum	1.5	1.6	45.7	1.9	4.0	121.9
Quanta per Invocation	4.1	3.4	2.7	4.8	21.7	3.4

Table 9: Dynamic scheduling characteristics under TAM for two programs on a 64 processor CM-5

5.2 Scheduling hierarchy

Table 9 indicates the effectiveness of the two-level scheduling hierarchy under long remote access latency and parallel execution. We see that depending on the application the compiler generates threads of about 3 to 17 TL0 instructions. However, anywhere from 7 to 530 threads execute in each quantum. The cost of posting and swapping the frame is amortized over this amount of work. Further, if register allocation is performed across threads, there are roughly forty TL0 instructions, to work with, rather than five to ten. Below (Figure 5) we show that each TL0 instruction is about 17 Sparc cycles, making the average quantum a few hundred cycles long. Viewed another way, a typical frame experiences about four periods of activity during its lifetime. This is comparable to a function in a sequential language that makes three calls.

The origin of the threads comprising a quantum is given in the middle rows Table 9. We see that when a frame is run, it has usually accumulated multiple threads. Since each successful post of a thread requires multiple posts (*e.g.*, in Gamteb 2.5 posts are required before a thread is pushed onto the RCV), a sizable amount of data will have been accumulated in the frame before it is scheduled. As a result, several potential synchronization events are passed without suspension and many threads are forked while the frame is running.

Typically, more than one message response arrives during the quantum in which it was issued, triggering further activity. Notice, Ifetches that are serviced locally and not deferred will complete during the issuing quantum. In fact, due to the amount of time it takes to issue a remote Ifetch, if the requests are not deferred, any series of four requests will generally cause a response in the same quantum.

Latency tolerance occurs in two ways: within a quantum and across quanta. Within the quantum, multiple requests are issued and before the quantum ends more than one of them will have completed and returned. Between quanta there is sufficient parallelism that when one frame finishes its quantum other frames have accumulated work to do.

5.3 Handling Remote Access

The large quanta size is achieved in spite of a fairly high remote reference rate. Table 10 shows the breakdown of split-phase operations for the two programs. For heap accesses, fetch and store operations are divided further to show the fraction of accesses to an element that is local to the processor issuing the access. The I-structure allocation policy recognizes two I-structure mappings. I-structures that are smaller than some threshold are allocated within a single processor, while those larger than the threshold are interleaved across processors. Small structures are allocated on the processor that requests the allocation, if space is available. We see that in Gamteb almost all the I-stores are local under this policy, as are one quarter of the fetches. Gamteb allocates many small tuples dynamically as particles are traced through the geometry. Under this policy, these are created and filled in on one node, but only a quarter of the I-fetches are local. Simple, on the other hand, operates mostly on large grids interleaved over the machine. No correspondence

Split-Phase Type	Gamteb 8192	Simple 128
I-Fetch	33.3	66.3
Local	26.4	2.1
Remote	73.6	97.9
I-Store	20.1	9.1
Local	99.1	15.2
Remote	0.9	84.8
Send	33.7	19.1
other	12.9	5.5

Table 10: Breakdown of split-phase operations into instruction types and locality. In Gamteb local allocation of small structures allows most of the stores and many of the fetches to be serviced without network access. Localization in Simple is much less successful, since the data structures are large and no correlation is established between program and data mapping.

is established between the data structures and the computations that access them. As a result, the cost of an average remote reference for Simple is higher than for Gamteb (which is reflected in Figure 5).

By extending our attention to another layer in the memory hierarchy, the heap, we can try to avoid latency as well as tolerate it. In order to reduce the number of remote references and the communication overhead, we introduce caching of remote references. For programs like Simple, where the remote reference rate is high and the number of deferred Ifetches is low (23%), the cache works remarkably well. Besides seeing the immediate effect of lowering remote references from 97% to 21%, it boosts the quantum size from 7.6 threads to 12.6.

5.4 Net Scheduling Cost

The two level scheduling hierarchy in TAM is supported by three basic scheduling operations: SWAP and POST at the frame level and FORK at the thread level. The quantum sizes above imply that FORK is by far the most frequent. Table 11 shows the result of the optimizations which specialize general FORK and SWITCH instructions into less costly jumps and branches. FORKs and SWITCHes that were executed at the bottom of the thread are listed as jumps and branches, respectively. The rows labeled with 'sync' are for the FORKs and SWITCHes to synchronizing threads. The row labeled 'other' represents the SWITCH instructions executed with both a synchronizing and non-synchronizing target. The rows labeled 'push' represent those control instructions that are unchanged.

The most common control transfer operation executed is the synchronizing, unconditional fork. Roughly half the pushes are simplified in favor of jumps or branches. In addition to the cost of enabling the thread, there is the cost of the STOP instruction, which is executed when a thread ends without a jump or successful branch, or about half the time. The bottom line is that, on average, the thread fork is performed in seven cycles.

The POST instruction has varying costs depending on whether it is executed for an idle, ready, or running frame (see Table 7). If most POSTs are executed when a frame is either ready or running, as is the case for the two benchmark programs, then the average cost of the POST instruction will be close to that of a FORK instruction. In Simple, on average, for each quantum, one post occurs to an idle frame, 0.4 occur while it is ready, and 1.9 occur while the frame is running, yielding an average cost of 9.4 cycles per post. The net result is that the average POST instruction, which handles both frame and thread level scheduling jobs, is only 30% more expensive than the average FORK.

Type	Cycle Cost	Gamteb	Simple
Fork a thread			
Fall through	0.0	3.9%	0.8%
Branch to thread			
unsynchronizing	1.0	3.2%	2.6%
successful sync.	4.0	8.2%	9.8%
unsuccessful sync.	8.0	21.7%	29.2%
Push thread onto LCV			
unsynchronizing	5.0	0.1%	0.4%
successful sync.	10.0	16.6%	20.0%
unsuccessful sync.	7.0	13.1%	27.4%
Switch a thread			
Branch to thread			
unsynchronizing	2.0	5.2%	4.0%
successful sync.	5.5	4.5%	0.6%
unsuccessful sync.	9.5	7.9%	1.3%
Push thread onto LCV			
unsynchronizing	6.5	5.4%	1.0%
successful sync.	12.0	1.9%	0.3%
unsuccessful sync.	8.5	7.7%	2.1%
Percentage of TL0 instructions		28.2%	15.0%
Average Cost		7.04 cycles	7.23 cycles

Table 11: Sparc cycle count and frequency of the different thread control transfer operations for Gamteb and Simple. As shown in Section 4 the code generator can specialize FORK and SWITCH to fall through or a branch.

Thread Characteristics	QS	Gamteb	Paraffins	Simple	Speech	MMT
Non-synchronizing Threads	60.7%	40.6%	66.5%	45.9%	65.0%	73.4%
Average entry for synchronizing Thread	2.4	2.5	3.0	3.7	4.4	7.0
Inlets per Threads	0.3	0.4	0.2	1.1	1.2	1.6
Ave TL0 Insts. per Inlet	4.0	5.1	3.0	3.4	3.0	3.0

Table 12: Dynamic thread characteristics under TAM for two programs on a 64 processor CM-5

5.5 TL0 Thread Structure

We now examine the structure of threads in more detail. Our compilation regime produces threads of about five instructions each, which, given TL0's frame relative addressing and single instruction network access, makes it roughly the length of a typical basic block. This is not surprising given that FORK is the only form of control transfer in TL0. The first row of Table 12 shows the fraction of non-synchronizing threads. This would seem to indicate that threads are much larger with branching permitted. However, more than half of the non-synchronizing threads are posted from inlets and these would remain distinct threads even under the looser definition.

The second row shows the average entry count for synchronizing threads. Under traditional dataflow execution mechanisms the entry count would be two. Grouping together the nodes that depend on a single matching event to form a thread, as on Monsoon or EM-4, will not change the entry count. Our partitioning algorithm is more aggressive and will group larger collections of nodes together to form a thread with a greater entry count. In addition, it also eliminates redundant forks, thereby reducing the entry count. The combination of entry count and thread length indicate the cost of each scheduling event and the amount of work per event.

The last two rows indicate the split of work between message handling and thread processing. From this we see that about one third to one half the program is directly related with handling the network. A simple inlet contains three instructions: receive, post, and stop. However, inlets also initialize thread entry counts, accounting for the remaining portion of the instructions per inlet.

5.6 Summary

The CPT is the work involved in an average TL0 instruction. This is obtained by multiuplying the frequency of each instruction with the cost in cycles for that instruction. Figure 5 shows the CPT for each of the benchmark programs. The CPT is further broken down into bars showing the contribution resulting from each TL0 instruction class. In addition, the Operand bar at the top reflects the memory access penalty in bringing data into registers for ALU instructions, assuming a 5% miss rate. The Atomicity bar at the bottom accounts for the overhead introduced by polling. The Heap bar has been split into three distinct implementation components. We see that the focus on efficient thread scheduling pays off, as this represents only 10 to 30 percent of the total costs. By comparison, one half to three quarters of the time is spent in the processor network interface. Even though the handling of these messages is very efficient; the simple fact that the network interface is on the memory bus, rather than the cache bus, accounts for almost all of this cost. Given the compilation techniques represented by TAM, the most important architectural investment for supporting emerging parallel languages is simply to bring the network interface closer to the processor. However, this must be accomplished without unduly increasing the memory access and atomicity costs.

6 Relationship to Other Models

TAM builds on a rich history of execution models developed to support general-purpose parallel programming. This section discusses the aspects of these models that TAM strives to retain and the short-comings that TAM attempts to avoid.

6.1 Dataflow

Under dynamic dataflow models, the program can be mapped arbitrarily onto the machine by simply hashing the tag associated with each instruction-level activity. Data is transmitted between instructions in the form of tagged tokens, where the tag carries control information (the context) for the destination instruction. When a token arrives at a processor, its tag is compared with the tags in a matching store. If no match is found, storage is allocated and the token is placed in the store to await its partner. The matching partner causes the token to be removed, its storage deallocated, and the enabled operation scheduled for execution. In addition, it is necessary to include a buffer in each processor to hold either fully enabled operations or tokens that have not been considered for matching. The simplicity of this model derives from the implicit allocation of storage and scheduling associated with each message arrival. Any operation can execute on any processor, simply by sending the tokens to that processor.

The shortcomings of the model derive from precisely the same factors: implicit allocation of storage and scheduling based on message arrival. First, the matching store serves essentially to hold the state of the overall computation, *i.e.*, the parallel analog of the call stack. Given its associative nature, it is impractical to make the matching store extremely large, so deep recursion or extensive parallelism cause the store to fill up and the program to deadlock. The only way to avoid this problem is to estimate the amount of storage that is implicitly allocated by the assignment of various units of work to a processor. By limiting the problem size and explicitly constraining the scheduling of computation, a

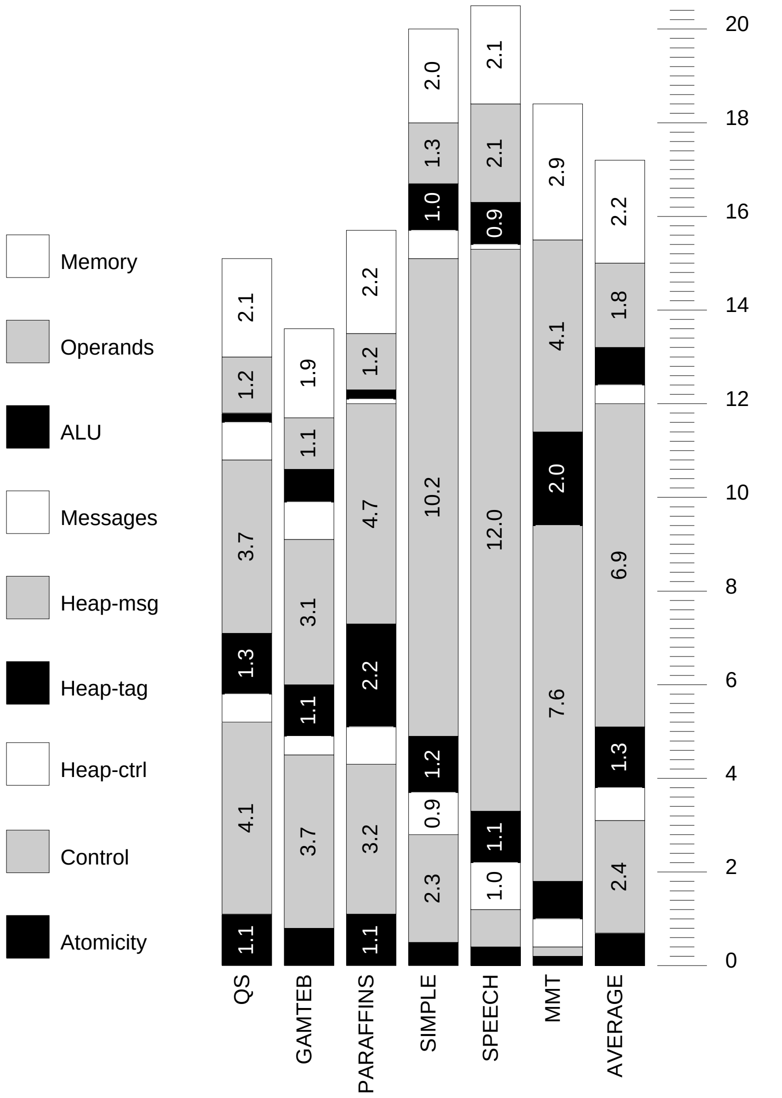

Figure 5: *Distribution of processor time for the benchmark programs. The metric used is Sparc cycles per TL0 instruction (CPT); bars show the contribution to the CPT resulting from each TL0 instruction class.*

program can be made to operate within the machine resource limits[9]. Second, the token queue serves to represent the excess parallelism in the program. To avoid overcommitting this resource, it is necessary to constrain the scheduling of computation to limit the maximum exposed parallelism. Finally, all of the information provided to the scheduled operation must be carried on the input tokens, so the amount of work performed per scheduling event must be small. The work per message event cannot be enlarged by referring to the context of the operation.

In refinements of the MIT Tagged-Token Data Machine[1] the local processor state took on an ever more significant role. Pure dataflow graphs circulate loop constants through the body of the loop for every iteration. To eliminate this overhead, a loop constant area was provided. Since it was necessary to allocate and initialize the loop constant area, work could no longer be assigned to processors by simply hashing the tag. The Manchester machine[18] and

Sigma-1[20] adopted an alternative approach of "sticky" match operations with similar drawbacks.

6.2 ETS

The ETS model[12] embodied in Monsoon[31] is a partial remedy to these problems. Implicit allocation of the matching store is eliminated by explicitly allocating an activation frame to hold the local storage for each function invocation. Synchronization bits are associated with each frame location to support a dyadic match. Using a k-bounded loop scheme, constants can be initialized in each of the loop frames.

However, the token buffer remains. When a message arrives, storage is allocated for it in a large queue. To avoid deadlock, this buffer must be large enough to represent the excess parallelism in the program. As in the pure dataflow case, this is determined by the scheduling of computation, not the relative rates of system components. (The token buffer in Monsoon is kept in a separate memory proportional in size to the frame store. It provides storage for roughly 16 tokens per frame on average, with a token queue store of 64K tokens for 256K words of frame store and an expected average frame size of 64 words. Since the token is removed from the queue when processed, LIFO or FIFO storage management is sufficient. Both are provided.)

The amount of work that can be performed per scheduling event is limited by the extent to which data can be drawn from the local processor state. In its final version, Monsoon allows short instruction threads to be scheduled, using frame slots and a small set of registers (an accumulator and three temporary registers) to convey data between instructions in a thread, based on the Hybrid work, discussed below. Furthermore, the machine cycle time is limited by the read-modify-write on the frame synchronization bits and the frame access per instruction. These times could potentially be reduced via caching, but the arbitrary interleaving of tokens in the queue is unlikely to provide any useful locality.

6.3 Hybrids

The Hybrid[23] proposal was the first to observe the interplay between register allocation and thread scheduling. This model provides a machine language of multiple threads operating against an activation frame and registers. However, each frame slot includes presence bits, like the ETS. A thread is suspended upon access to a frame slot marked not-present. The suspension is accomplished by building a queue of instruction pointers into waiting threads. The queue is rooted in the frame slot, so the eventual store of the value enables the threads. Since no registers are saved upon suspension, the compiler is required to evacuate the registers across any potential point of suspension. This elaborate suspension, queuing and scheduling mechanism is part of the basic model and required in any machine that implements it. Scheduling (and the scheduling structure) is outside the programming model, so there is no means by which the compiler can organize the program to make efficient use of processor resources.

P-Risc[28] observed that presence-bits can be kept in the frame like local data, rather than as special tags, and that matching could be simulated by toggling the tag bit atomically and suspending on the result. This is easily extended to a general counter, as in TAM. P-Risc eliminated the notion of suspension within a thread. However, it failed to retain the distinction between registers and frames of the Hybrid model. Instead the entire frame is viewed as a set of registers. Like the hybrid model, scheduling is outside the execution model. When a thread completes, any enabled thread could execute next, so there is no means by which the compiler can develop a higher level strategy for utilizing processor resources while tolerating latency.

6.4 Message Driven Processing

Message-driven processing generalizes the dataflow model by allowing the token to carry an arbitrary amount of data and eliminating the synchronization discipline implied by matching. Parallelism is generating by sending a message with the address of a handler at its head, followed by a sequence of data. The message-driven model holds that when a message arrives at the processor, storage is automatically allocated for it in a scheduling queue. When the message reaches the head of the queue, its handler is invoked. The handler can comprise an arbitrary computational task, including sending of messages and waiting for synchronization events. Therefore, storage allocated for the message can have an arbitrary lifetime and simple stack or queue based storage management is not sufficient. The message buffer (or scheduling queue) serves to represent a significant portion of the state of the complete computation and the excess parallelism in the program. Furthermore, the amount of work performed per scheduling event must be small, unless stronger assumptions are made about the state of the computation assigned to the particular processor. Thus, the fundamental shortcomings of the dataflow approach remain, the tokens are simply larger.

The J-Machine[14] approximates the message-driven model, rather than implementing it directly. A portion of the on-chip memory provides a message buffer and scheduling queue managed in hardware as a fixed-size ring buffer. Arriving messages are transferred into the queue and serviced in FIFO order. The first word of each message is interpreted as an instruction pointer and the message is made available to the handler as one of the addressable data segments. If the handler does not run to completion, it must copy its message data to an allocated region of non-buffer memory. This happens for roughly $1/3$ of all messages. (Close to $1/3$ of the messages hold a request to which the handler immediately replies and general allocation and scheduling is not required.) In reflection of the small amount of computation per message, the instruction set provides only four data and three address registers.

The fundamental difference between the message driven model and TAM is where computation-proper is performed: in the former, computation occurs in the message handlers whereas in the latter it is in "background" threads. TAM inlets only remove messages from the network and integrate them into the computation. This difference significantly affects the nature of allocation and scheduling performed at message arrival. Inlets execute immediately upon message arrival, cannot suspend, and have the responsibility to terminate quickly enough not to back-up the network. The inlet either moves the message data into the data structures of the ongoing computation or, in the case of remote service requests, immediately replies to the requester.

This seemingly slight restriction on message handlers has a significant conceptual implication — the on-going computation on the node is assumed to be manipulating a significant amount of local state in a manner consistent with the local storage hierarchy. The higher level programming model must ensure that the storage required to handle a message is allocated and available when the message arrives. Also, by accumulating several messages and making use of local data, the amount of work per dynamic scheduling event can be made larger and better use can be made of large register files.

6.5 Light-weight thread packages

TAM can be compared to light-weight threads packages in that a goal of both is to allow the representation of arbitrary parallelism and to support interleaved execution of logically parallel tasks mapped onto a single processor. It is important, however, not to be mislead by the terminology: *threads* in threads packages are roughly comparable to TAM activations, rather than TAM threads.

The frame scheduling of TAM can be implemented using a threads package: when a frame is allocated a *thread* is created for the activation, when the first TAM thread is posted the *thread* is enabled. Swapping to the next activation corresponds to suspending the current *thread*.

However, TAM frame allocation and scheduling is much cheaper to implement than that of a threads package. The size of a frame is fixed and specified when it is allocated, whereas the stack of a *thread* can grow arbitrarily. The TAM frame swap code is produced in-line and is closely integrated with the frame data structure, which includes part of the scheduling queue, and the thread scheduling. Furthermore, the compiler is responsible for saving registers at the end of a quantum. In contrast, threads packages must assume that all processor registers are live and must be saved. Finally, threads packages do not provide the same degree of compiler control over scheduling.

7 Summary

This paper has presented TAM, a threaded abstract machine that serves as a framework for implementing general purpose parallel programming languages. TAM is a parallel machine language of multiple threads. It can represent the sophisticated and dynamic forms of parallelism arising in such languages. The model evolved from work on dataflow execution models and addresses many of the same goals. However, it fundamentally differs from dataflow models in allowing the compiler to control the scheduling of threads. Furthermore, TAM exposes, rather than hides, the critical performance aspects of modern multiprocessors: interprocessor communication, synchronization, and utilization of the storage hierarchy. This permits the compiler to apply optimizations in recognition of these factors. Perhaps the most important feature of TAM is the way it exposes the interaction between the handling of asynchronous message events, the scheduling of computation, and the utilization of the storage hierarchy.

The empirical investigation of TAM centers on an implementation of Id90 on the CM-5 multiprocessor. The TAM model is codified in a pseudo-machine language TL0. A compiler was implemented from Id90 to TL0 and a code-generator was constructed to translate TL0 into Sparc instructions with direct access to the network interface. This provides a detailed cost model for the various TAM primitives in the absence of specific hardware support. Combined with usage measurements on real programs it provides a baseline for assessing architectural alternatives. More importantly, it demonstrates that the TAM scheduling hierarchy of quanta and threads is effective in practice. Sizable chunks of work are scheduled from a single loop or function body in each quantum, so registers and caches can be well utilized. The quanta are much larger than what worst-case static analysis predicts. The most frequent scheduling event is a simple thread fork, which incurs a cost of seven cycles. Substantial latency tolerance is achieved since there are multiple enabled frames with multiple enabled threads on each processor most of the time. Moreover, remote accesses frequently complete within the issuing quantum, which allows the synchronization and scheduling to be handled almost as cheaply as the simple thread fork.

This work demonstrates that a dataflow graph representation of programs should be adopted at compile time, but a control flow representation at run-time. The TAM model provides a bridge from one to the other. It allows static scheduling to the extent possible in the formation of threads, while making dynamic scheduling efficient by biasing it toward logically related threads that share resources in the level of the storage hierarchy closest to the processor.

While this work strives to settle certain issues regarding the effectiveness of dataflow or multithreaded execution models, we believe it opens several avenues of research. It clearly identifies the areas where architectural efforts can be applied most profitably, especially improved network access. The division of work between inlets and threads suggests that separate processors should be considered, but to handle the frequent case of remote responses arriving during the issuing quantum there needs to be a very tight coupling between the two processors[39]. Looking at the anticipated evolution of microprocessor architectures, the reliance on branch prediction will be undermined by the TAM style of indirect transfer to threads. This is an interesting trade-off because the fork-based model allows ample opportunity for thread prefetching, which is lost when mapped to jumps and branches. The vast area of open problems lies in compilation for this kind of execution model. A prime example is the question of register management using estimates of quantum boundaries and remote access latencies. Additionally, how can static analysis contribute to compiler directed scheduling policies? It is expected that the simple storage directed scheduling policy embodied in TAM will not be sufficient in the long run. We have seen examples where a processor becomes so successfully focused on its local work that it starves other processors by failing to spawn off additional work[13]. Nonetheless, it is clear that latency tolerance and dynamic scheduling must be addressed in concert with the characteristics of the local storage hierarchy.

Acknowledgments

We are grateful to the anonymous referees for their valuable comments. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship. Klaus Erik Schauser is supported by an IBM Graduate Fellowship. Thorsten von Eicken is supported by the Semiconductor Research Corporation.

References

[1] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The tagged token dataflow architecture. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1983. Revised October, 1984.

[2] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[3] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proc. of the Fourth Int. Symp. on Biological and Artificial Intelligence Systems*, pages 255–286, Trento, Italy, September 1988. ESCOM (Leider).

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

[5] R. Buehrer and K. Ekanadham. Incorporating Dataflow Ideas into von Neumann Processors for Parallel Execution. *IEEE Transactions on Computers*, C-36(12):1515–1522, December 1987.

[6] 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.

[7] A. A. Chien. Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines. Technical Report AI-TR 1248, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1990.

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

[9] D. E. Culler. Managing Parallelism and Resources in Scientific Dataflow Programs. Technical Report 446, MIT Lab for Comp. Science, March 1990. (PhD Thesis, Dept. of EECS, MIT).

[10] D. E. Culler. Multithreading: Fundamental Limits, Potential Gains, and Alternatives. In *Proc. of the Supercomputing '91, Workshop on Multithreading*, 1992. (to appear).

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

[12] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10:289–308, January 1990.

[13] D. E. Culler, K. E. Schauser, and T. von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL.* North-Holland, January 1993. (Also available as Technical Report UCB/CSD 92/716, CS Div., University of California at Berkeley).

[14] W. J. Dally. The J-Machine system. In P. Winston and S. A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, chapter 21, pages 536–569. MIT Press, 1990.

[15] J. B. Dennis. First Version of a Data Flow Procedure Language. In G. Goos and J. Hartmanis, editors, *Proc. Programming Symposium, Paris (Lecture Notes in Computer Science 19, Springer Verlag)*. Spinger-Verlag, New York, 1974.

[16] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *Proc. of Compcon90*, pages 88–93, March 1990.

[17] A. Gupta et al. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 254–263, Gold Coast, Australia, May 1992.

[18] J. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[19] S. K. Heller. Efficient Lazy Data-Structures on a Dataflow Machine. Technical report, MIT Lab for Comp. Science, February 1989.

[20] K. Hiraki, S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Technical report, Computer Systems Division, Electrotechnical Laboratory, Japan, 1987.

[21] W. Horwat. Concurrent Smalltalk on the Message-Driven Processor. Master's thesis, Massachusetts Insitute of Technology, Cambridge, MA, May 1989.

[22] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Lab for Comp. Science, Cambridge, MA, May 1988. (PhD Thesis, Dept. of EECS, MIT).

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

[24] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[25] P. J. Landin. The Mechanical Evaluation of Languages. *Computer Journal*, 6(4):308–320, January 1964.

[26] W. Lonergan and P. King. Design of the B5000 System. *Datamation*, May 1961.

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

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

[29] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Killer Micro for A Brave New World. Technical Report CSG Memo 325, MIT Lab for Comp. Science, Cambridge, MA, January 1991.

[30] R. R. Oldehoeft, D. C. Cann, A. P. W. Böhm, J. T. Feo, and D. H. Grit. SISAL Reference Manual Language Version 2.0. Technical Report UCRL-JC-104008, Lawrence Livermore National Laboratory, Livermore, CA, December 1988.

[31] 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.

[32] C. D. Polychronopoulos. Toward Auto-scheduling Compilers. *J. of Supercomputing*, 2:297–330, 1988.

[33] B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, 1964.

[34] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine*. PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[35] R. Saavedra-Barrerra, D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the 2nd Annual Symp. on Par. Algorithms and Arch.*, July 1990.

[36] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, pages 46–53, Jerusalem, Israel, June 1989.

[37] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge, MA*, pages 202–211, August 4-6 1986.

[38] K. E. Schauser, D. Culler, and T. von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991. (Also available as Technical Report UCB/CSD 91/640, CS Div., University of California at Berkeley).

[39] 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.

[40] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, January 1992.

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

[42] K. R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-strict Functional Languages. In *Proc. of the Apenas Workshop on the Implmentation of Lazy Functional Languages*, Chalmers Univ., Goteborg Sweden, September 1988.

[43] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Lab for Comp. Science, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[44] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proc. of the ACM Conf. on LISP and Functional Programming*, San Francisco, CA, June 1992.

[45] 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).