TAM — A Compiler Controlled Threaded Abstract Machine [1]

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 hierarchy 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

[1] This report appears in the Journal of Parallel and Distributed Computing, Special Issue on Dataflow, vol 18, June 1993.

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 occurring simple cases. For the foreseeable 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 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.

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