Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5

Ellen Spertus†, Seth Copen Goldstein‡, Klaus Erik Schauser‡, Thorsten von Eicken‡,
David E. Culler‡, William J. Dally†

†MIT Artifical Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
{ellens,billd}@ai.mit.edu

‡Computer Science Division — EECS
University of California
Berkeley, CA 94720
tam@cs.berkeley.edu

Abstract

This paper uses an abstract machine approach to compare the mechanisms of two parallel machines: the J-Machine and the CM-*5. High-level parallel programs are translated by a single optimizing compiler to a fine-grained abstract parallel machine, TAM. A final compilation step is unique to each machine and optimizes for specifics of the architecture. By determining the cost of the primitives and weighting them by their dynamic frequency in parallel programs, we quantify the effectiveness of the following mechanisms individually and in combination. Efficient processor/network coupling proves valuable. Message dispatch is found to be less valuable without atomic operations that allow the scheduling levels to cooperate. Multiple hardware contexts are of small value when the contexts cooperate and the compiler can partition the register set. Tagged memory provides little gain. Finally, the performance of the overall system is strongly influenced by the performance of the memory system and the frequency of control operations.*

Keywords: *Parallel Processing, Performance Analysis, Compilation.*

1 Introduction

Several experimental parallel architectures have been developed in recent years to demonstrate novel hardware mechanisms that may enhance the performance of programs written in emerging parallel languages. For example, Monsoon focuses on Id90, the J-Machine on CST, Alewife on Mul-T, the CM-5 on Fortran90, and Dash and KSR-1 on extensions to C and Fortran. All of these architectures provide a family of mechanisms that collectively support the requirements of the parallel language, are universal enough to support any of the other language paradigms, and are real enough to be constrained by the traditional technology forces. Thus, it would seem that the time has come for parallel architecture research to begin the shift from "big new ideas" to careful quantitative analysis of the effectiveness of various mechanisms. In this paper, we seek to evaluate the set of mechanisms in the MIT J-Machine with respect to the implicitly parallel language Id90 and draw a quantitative comparison with the CM-5.

At the current state of parallel computing, a completely satisfactory quantitative analysis of mechanisms is difficult to achieve because there is no well-established body of machine-independent software reflected in a standard set of benchmarks. There is not even a consensus on the programming languages of choice. Where benchmarks exist, they have been developed specifically for the machine that they are intended to evaluate [14, 9] or specifically avoid emerging languages and the novel mechanisms which could bring them within practical reach [3]. It is also difficult to obtain high-quality compilers for such new languages on more than one machine, yet it is well understood that the architectural support can only be evaluated in the context of sophisticated compilation, rather than direct execution of high-level constructs. Finally, the machines reflect substantially varying engineering budgets and designer capabilities, which should be factored out of the evaluation of the architectural contribution. Simply comparing execution times gives only a crude and noisy calibration, failing to isolate the reasons for the differences.

The method of analysis employed in this paper is as follows. We consider two recent parallel machines: the J-Machine, developed at MIT as a study in universal mechanisms for fine-grained parallelism, and the CM-5, developed at Thinking Machines Corp. as a commercial product supporting data-parallel programs. We take as a basis for comparison a powerful machine-independent parallel language, Id90, which was not the primary target for either architecture, but for which a high-quality compilation sys-

tem exists. The compiler performs a variety of high-level optimizations in translating the language down to code for a simple abstract machine, TAM [6, 13]. The TAM code is identical for the two machines, controlling for effects of high-level optimizations. The translator from TAM code to machine language, however, employs a variety of machine-specific optimizations reflecting the most advantageous use of the available mechanisms. The performance of isolated mechanisms is reflected in the cost of the individual TAM primitives on the machine. The overall effectiveness of the family of mechanisms is determined by weighting each of the primitives by its frequency in a suite of programs. The J-Machine essentially provides direct hardware support for every aspect of TAM; however, TAM does not use all the mechanisms in the machine. The CM-5 provides a variety of mechanisms for data-parallel programming, which are not useful to TAM. What remains is a very reasonable baseline machine, essentially a collection of workstation-class processors on a dedicated network. Thus, we can compare a sophisticated set of mechanisms against a familiar baseline architecture with respect to the dynamic load presented by Id90 programs compiled to TAM.

Section 2 describes the two architectures under study and explains the salient aspects of TAM. TAM-level dynamic instruction frequencies are produced for a variety of programs to serve as a basis for comparison. Section 3 corrects for a set of architectural and engineering factors that have a significant impact on execution time for the two machines, but for which conventional wisdom (and hindsight) applies. The remaining sections deal with architectural aspects that are unique to parallel computing. Section 4 examines the impact of the processor/network coupling on message-passing cost. Section 5 looks at three mechanisms related to asynchronous message arrival that interact with dynamic scheduling. Section 6 considers the utility of tagged memory words and Section 7 ties together our observations. Two important lessons arise from the study. First, novel mechanisms do not substitute for solid engineering of the processor pipeline and storage hierarchy. Second, mechanisms should not be evaluated in isolation, but in how they work together in the compilation framework for the programming language.

2 Background

2.1 CM-5

The CM-5 [16] is a massively-parallel MIMD computer based on the Sparc processor, interconnected in two identical disjoint "hypertree" networks. Each node consists of a 33 MHz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte cache), 8 MBytes of local DRAM memory and a network interface to the hypertrees and broadcast/scan/prefix control networks. (The node may also contain vector units with additional memory, but we will not address the vector capability.) The network interface consists of a pair of memory-mapped FIFO queues for each of the two data networks. Messages are limited to a maximum of five 32-bit words in length. Message delivery is reliable, but no guarantee is made on ordering. The study uses a 128-node CM-5, although machines of 1024 nodes are currently in the field.

2.2 J-Machine

The J-Machine is a massively-parallel MIMD computer based on the Message-Driven Processor (MDP) interconnected by a 3-D mesh network. The MDP is a single-chip processing node composed of a 16 MHz 32-bit integer unit, a 4K by 36-bit static memory, a closely integrated network interface, a packet router, and an ECC DRAM controller. The on-chip memory is augmented with a 256K by 36-bit off-chip memory. The 36-bit words include 4-bit tags, which indicate data types such as booleans, integers, and user-defined types. Two special tag values *future* and *cfuture* cause a trap when accessed. The MDP has three separate priority levels: background, 0, and 1, each of which has a complete context, consisting of an instruction pointer, four address registers, four general-purpose registers, and other special-purpose registers. A 512-node J-Machine has been built, and a 1024-node machine is planned.

The MDP implements a prioritized scheduler in hardware. When a message arrives at its destination node, it is automatically written into a message queue, consisting of a fixed-size ring buffer in on-chip memory. Background execution is interrupted by priority 0 message reception, which in turn may be interrupted by priority 1 message reception.

2.3 TAM

TAM defines a fine-grained parallel execution model used as a compilation target for Id90. Although it grew out of work on dataflow, it defines a simple model of self-scheduling threads that can be implemented on any machine. The key ways in which TAM differs from "thread packages" are that TAM threads are even lighter weight, the scheduling is integrated with aspects of compilation, such as register allocation, and there is no external scheduler.

A TAM program consists of a collection of *code-blocks*, which typically represent functions or loops in the source program. Each code-block consists of a collection of *threads*, which correspond roughly to basic blocks. Two instructions appear in the same thread only if they can be statically ordered and if no operation whose latency is unbounded occurs between them.

The TAM execution model centers on the *activation frame*, which is the analog of a stack frame for parallel calls. To invoke a code-block, a frame is allocated on a processor and initialized, and arguments are sent to the

frame. Initialization consists of setting the values of *synchronization counters* stored within the frame. A thread is allowed to run only when all its antecedents have been executed. To detect the completion of antecedents, a synchronization counter is associated with each thread. The counter is omitted for threads that have only one antecedent, *i.e.*, *unsynchronizing* threads. For each frame, a stack of instruction pointers, called the *continuation vector* (CV), holds the list of threads that are ready to run. The arguments to the code-block, results from subordinate calls, and responses to global heap accesses are received by *inlets*. Inlets are compiler-generated message handlers that copy the arguments into the frame and enable computation dependent on the message. In order to process requests from the network quickly, inlets are small and run at a higher priority than threads.

Maintaining the thread queue in the frames provides a natural two-level scheduling hierarchy. When a frame is scheduled, the *remote* continuation vector (RCV) is copied into the *local* continuation vector (LCV), from which enabled threads are executed until the LCV is empty. The set of threads that run during this time is called a *quantum*. Each processor maintains a queue of *ready* frames with non-empty CVs. A new frame is activated from the queue when a quantum completes.

Global data structures in TAM provide synchronization on a per-element basis to support I-structure and M-structure semantics [10]. In particular, reads of empty elements are deferred until the corresponding write occurs. Accesses to the data structures are split-phase and are performed via special instructions: `ifetch` reads an element by sending a message to the processor containing the data which returns the value to an inlet, `istore` writes a value to an element, resuming any deferred readers, and `ialloc` and `ifree` allocate and deallocate I-structures.

In the current implementation of TAM, instructions are primarily three address, where the operands are constants, registers, or frame locations. TAM registers and frame slots are statically typed into integers, floats, various pointers, and generals. Generals are sufficiently large to contain any TAM type but do not identify the type. Correct compilation ensures that the producer and consumer of a general agree on the type of the contained value. No fixed limit is placed on the number of TAM registers, although the compiler tries to use them as efficiently as possible. The translator from TAM to a target machine is responsible for mapping TAM registers to physical registers or spill areas.

The key issues presented by TAM are the parallel call, dynamic synchronization of computation with asynchronous responses from both remote requests and calls, split-phase remote operations, and the overlap of computation with communication.

2.4 Mapping to the machines

The basic mapping of TAM to the two machines is relatively straightforward. Program code is placed on every processor, but a given code-block invocation takes place on a single processor. Because the compiler may pull loops out into separate code-blocks, these can be spread across the machine to implement parallel loops [7]. The memory on each processor is divided into two areas. One holds small arrays and activation frames. The other holds large arrays, which are spread across all the processors such that logically consecutive elements are on different processors. Memory is managed explicitly through library routines.

The J-Machine implementation of TAM [15] makes direct use of the hardware support for different priority levels. Threads run at the background priority level, allowing them to be quickly interrupted by messages arriving in the priority 0 queue. (Priority 1 is currently not used.) Because each priority level has its own register set, inlets do not interfere with thread execution. An address register is set aside in each register set to hold the frame pointer. Threads use an additional general-purpose register to hold the address of the top of the LCV. Two general-purpose registers are used as temporaries to hold memory operands and to implement complex TAM instructions. The remaining general-purpose register is used to hold one TAM register. All other TAM registers are mapped to the base of on-chip memory, a region that can be addressed easily. Frames are stored in main memory.

A similar approach is followed on the Sparc with inlets using a new register window. However, due to the tight coupling between threads and inlets, it proves to be more efficient to simply partition a single window. The CM-5 implementation [8] uses 32 registers divided into three classes: *global registers* which hold frequently-used values, *TAM registers* which are preserved for the duration of a quantum, and *inlet registers*, used during inlet execution and to pass information from threads to inlets. The CM-5 translator attempts to keep as many TAM variables as possible in the TAM registers and spills the rest into the frame.

2.5 Benchmarks

The empirical basis for comparison is provided by six benchmark programs described below. TAM-level dynamic instruction distributions are collected by running an instrumented version of the program on the CM-5. The translator inserts in-line code to record roughly a hundred specific statistics on each processor, which are combined at the end of the program.[1] These are grouped into the basic

[1] The Benchmark programs, raw data, and tools to process the data can be retrieved by anonymous FTP from ftp.cs.berkeley.edu under /ucb/TAM/isca93.tar.Z.

instruction categories in Figure 1. *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.

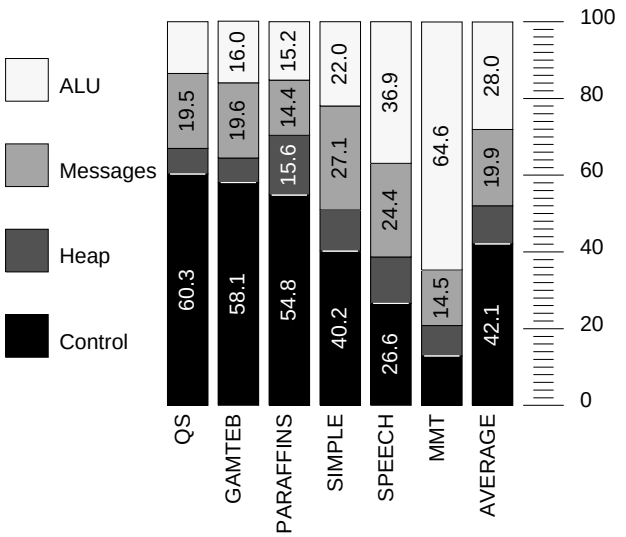

Figure 1: *Dynamic instruction mix statistics for the benchmark programs. The final column shows the arithmetic mean of the distributions.*

Six benchmark programs ranging from 50 to 1,100 lines are used. *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 [4]. It is highly recursive with many conditionals. *Paraffins* [2] enumerates the distinct isomers of paraffins. *Simple* is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id90 [5, 1]. *Speech* determines cepstral coefficients for speech processing. *MMT* is a simple matrix operation test using 4x4 blocks; two double precision identity matrices are created, multiplied, and subtracted from a third.

The programs toward the left of Figure 1 represent fine-grained parallelism. Notice that they are control intensive and make frequent remote references, as opposed to the blocked matrix multiply which is dominated by arithmetic. We will focus primarily on the two largest programs, Gamteb and Simple.

The Id90 implementations of these programs take about twice as long on a single processor as implementations in standard languages like C or Fortran [6]. Some of this overhead (mostly seen in the amount of control in Figure 1) is recouped in the parallel implementation.

3 Baseline Architectural Issues

By relating the dynamic statistics to the cost of the implementation of each TAM primitive we can obtain an estimate of how time is spent on each machine and isolate the contribution of specific mechanisms. However, there are several significant engineering differences between the study machines including cycle time, pipelining, floating point support, caches, and message size. These differences are determined primarily by circumstances under which the machines were developed and do not reflect significant architectural characteristics. The J-Machine was developed by a small academic team and many tradeoffs were made in favor of reduced design time at the expense of absolute performance. The CM-5 was developed by a relatively small company with significant time-to-market pressures. However, it was able to exploit the sizable investment in the Cypress Sparc chip set. To understand the impact of the novel mechanisms in detail, we first compensate for these differences. In particular we adjust the cycle time, floating point performance, and memory system of the J-Machine to reflect advances in technology and engineering that have occurred since it was implemented.

In this section we develop hypothetical versions of the two machines, called J′-Machine and CM-5′, with similar engineering characteristics. The predicted performance breakdown of the two real and two hypothetical machines on our benchmark programs is shown in Figure 2 and explained below. The metric used is cycles per TAM instruction (CPT). The bars show the contribution to the CPT resulting from each class of TAM instruction. Three new segments have been introduced to highlight important implementation issues. The *memory* system segment at the top indicates the penalty introduced due to cache misses. Since the J-Machine manages the movement between SRAM and DRAM explicitly, there is no direct penalty. The *operands* segment reflects the memory access penalty in bringing data into registers for ALU instructions. The *atomicity* segment at the bottom accounts for overhead in ensuring certain operations are atomic. The *heap* bar has been split into three distinct implementation components.

Our main task in the paper is to show how the cost coefficients are determined for each category, and specifically how the novel mechanisms in the J-Machine contribute toward reducing each cost. In this section we address the top two segments, because these have to do with conventional processor efficiency. The remaining sections examine the lower segments, which involve issues that are unique to fine-grained parallelism. The factors that we normalize are the following.

Cycle time and pipelining: The Cypress Sparc processor in the CM-5 has a cycle time of 30 ns and has a four-stage pipeline, while the MDP in the J-Machine has a

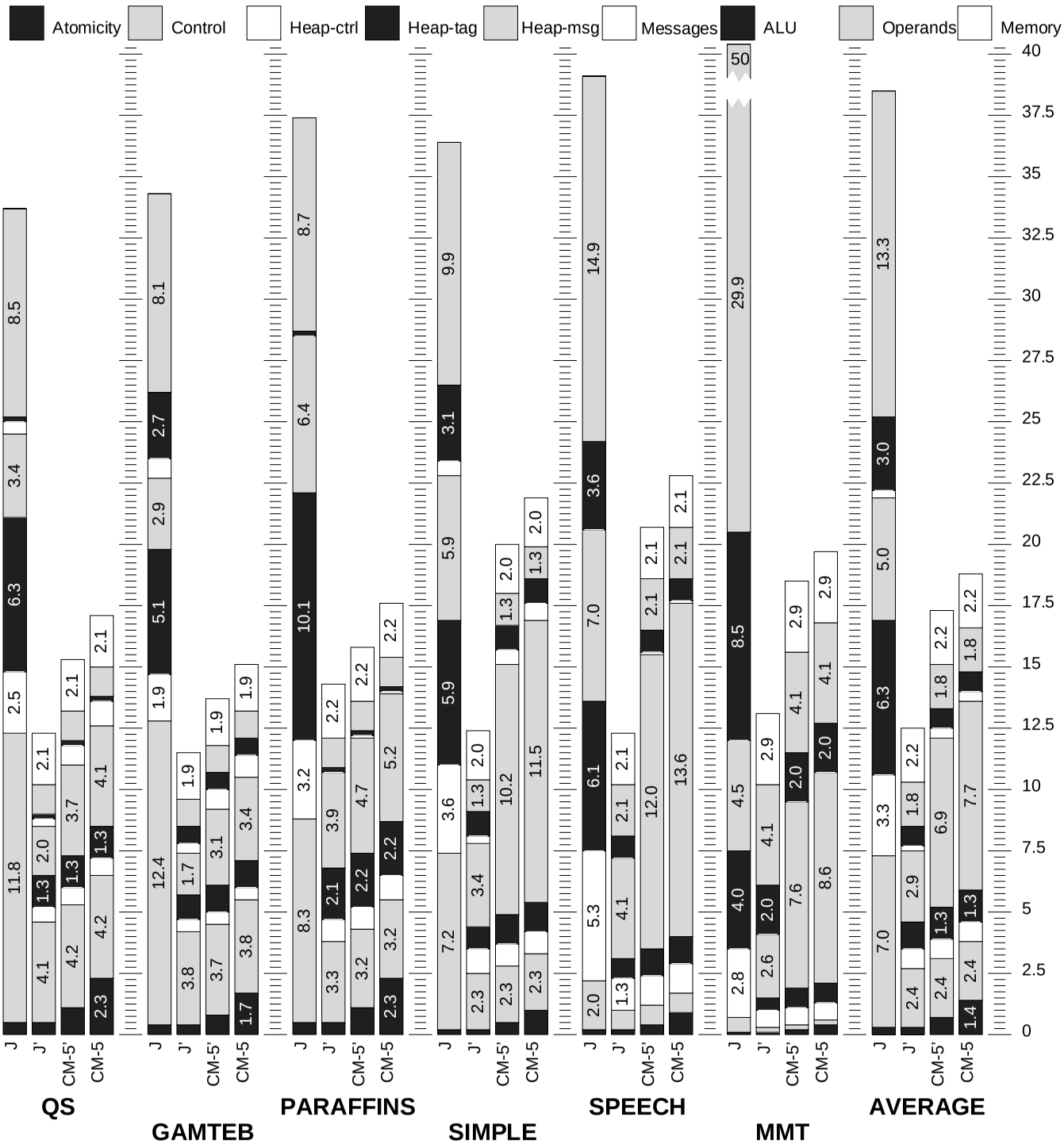

Figure 2: *Relative performance of the J-Machine, the CM-5 and their hypothetical variations on the sample dynamic instruction mix. The variations are used in the study to compensate for differences in engineering which do not offer any new architectural insight. Since the J-Machine does not have a cache, the memory penalty is factored into the other segments.*

cycle time of 62.5 ns and is unpipelined. Since the novel mechanisms of the J-Machine do not fundamentally impact cycle time or pipelining, we ignore these differences. We scale DRAM access time accordingly and measure it in terms of processor cycles. We also assume the MDP has the same control pipeline as the Sparc; in particular, we introduce annulling branches and branch delay slots into the MDP.

Networks and load balancing: In the analysis, we assume the performance characteristics of the network to be identical in the two machines and to affect each processor in a similar way. Similarly, we assume idle time resulting from inadequate parallelism or improper load balancing to affect the two machines equally.

Floating point arithmetic: On the Sparc, floating point operations run on a co-processor and take roughly seven cycles, unless overlapped with integer execution. The MDP has no built-in floating-point hardware, so software routines, typically taking 12–28 cycles, are used. For this analysis, we assume that the J′-Machine has the same floating point support as the Sparc. For all arithmetic-logic operations, operands in the frame must be brought into registers by explicit load instructions (2 cycles for 32-bit operands, 3 cycles for 64-bit), and results must be stored back (3 cycles for 32-bit, 4 cycles for 64-bit).

Storage hierarchy: Both machines have a 3-level memory hierarchy — registers, SRAM, and DRAM. However, the two faster levels differ significantly. The MDP has eight general-purpose or address registers for each of three priority levels. The Sparc has overlapping register windows of 32 registers each. The MDP has on-chip SRAM mapped into the bottom of the address space. One additional cycle is required to read the on-chip SRAM and 6 for off-chip DRAM, which we treat as 2 and 12, respectively, since we have doubled the MDP's clock rate. The Sparc has off-chip SRAM organized as a 64 Kbyte direct-mapped unified data and instruction cache. Cache hits causes a one-cycle processor stall, while refills from DRAM take 20 cycles. We assume that the J′-Machine has as many registers and the same memory system as the Sparc. A cache miss rate of 5% is used.

Message size: The TAM `send` and `receive` instructions do not place a limit on the number of words transferred. Although most messages are small, arguments to a code-block may involve a few tens of data elements. On the J-Machine, these can be delivered in a single message to a single inlet. The CM-5 limits the message size to five 32-bit words; thus, large messages are broken into smaller messages which are each sent to different inlets (to handle possible reordering of the smaller messages), incurring additional overhead. The CM-5′ is assumed to be able to send larger packets, 16 words, which is sufficient to hold the biggest message generated in our benchmarks.

Polling: Due to the current implementation of the network interface in the CM-5 the two data networks must be polled separately, which doubles the cost of polls. In the future both data networks will be polled simultaneously, which is what we assume for the CM-5′.

Correcting for the memory system and lack of floating point results in a significant improvement of the J′-Machine performance. Observe this improvement affects all the categories in Figure 2. After the correction, the time spent in arithmetic operations and memory access is nearly identical on the two machines. More interestingly, adding a cache and registers results in a more uniform memory system and increases processor state which not only speeds up ALU operations but allows for significant optimizations of all TAM instructions. This is particularly noticeable in how similar the cost of control becomes between the J′-Machine and the CM-5′.

The message, control, and heap components are the remaining factors and demonstrate significant differences between the machines. These are examined in detail below to explain precisely how these costs arise.

4 Processor/Network Coupling

This section examines the cost of sending and receiving messages and relates it to architectural features in the two machines. The J-Machine provides extensive message support with an intimate coupling between the processor and network, whereas the CM-5 has a memory-mapped network interface connected to the MBUS on each node.

4.1 TAM requirements

The *messages* segment in Figure 2 reflects the cost of passing arguments and return values for function calls, and of initializing loop constants and forwarding iteration variables for parallel loops. In TAM a message is formed and issued with the `send` instruction, which sends a number of data values to an inlet of a potentially remote frame. The message is received by a `receive` instruction in the destination inlet which extracts the data from the message and stores it into the frame. The adjacent segment of the *heap* segment reflects the cost of the message component of global data structure accesses, represented by `ifetch` and `istore` instructions. Note that a heap fetch requires two messages, a request to the node containing the accessed element and a response with the data.

4.2 Hardware support

On the J-Machine, the SEND instruction appends one or two 32-bit values to the message currently being composed. The first word of the message contains the destination address, and a bit in the SEND instruction indicates the end of

the message and completes its injection into the network. Overflow of the output buffer causes a fault. The CM-5 operates similarly with two exceptions: the message data must be stored into the outgoing fixed length FIFO queue of the memory-mapped network interface, and the network interface status register must be checked after each message to verify its successful injection. If the network is backed-up, the CM-5 network interface simply discards additional messages, and the program must retry the SEND until the status register indicates success.

Hardware support for message reception differs substantially on the two machines. The J-Machine implements asynchronous message reception by directly storing messages into an on-chip queue and dispatching to the code indicated by the first word of the message at the head of the queue, *i.e.*, the inlet. The inlet may load the message data one word at a time into registers from there. On the CM-5 message arrival is detected by software polling. When a message has arrived, software loads the first word of the message into a register and dispatches on it. The message data is received through additional loads from the network interface.

While an asynchronous message reception model appears most natural on the J-Machine, an alternate synchronous implementation is also possible: the message dispatch mechanism can be kept generally disabled except for short periods during which the network is essentially polled. This approach maintains the advantage of fast hardware dispatch but lets the compiler control when the computation can be interrupted by inlets. This will be shown to be a useful property in the following section. Similarly, an asynchronous message reception model can be implemented on the CM-5 using interrupts: on message arrival the network interface signals an interrupt, causing the Sparc to trap to the kernel. The kernel forwards the interrupt to the user process by creating a stack frame for the inlet and returning to it.

4.3 Implementation costs

Table 1 shows the cost of sending and receiving a message using synchronous and asynchronous reception on both machines. The start-up cost for sending a message to a remote processor is 3.5 times as high on the CM-5′ as on the J′-Machine, mainly due to the status register check. The per-word cost is twice as high on the CM-5′, due to the cost of stores across the MBUS. Periodic polling is required in the synchronous reception models. The start-up cost for receiving a message includes the dispatch and the return to the interrupted computation. It is roughly twice as high on the CM-5′ with synchronous reception. However, the extremely high `receive` start-up cost for the asynchronous reception on the CM-5′ shows that the user-level interrupt which comes with the hardware dispatch is a dramatic improvement. The 100 cycles for the kernel trap on the CM-5′ is an approximation and assumes that, on average, two to three messages are received back-to-back per interrupt. Notice that the asynchronous model also increases the cost of local messages, in that local messages must be atomic with respect to remote messages. A remote `ifetch` involves two messages (request and reply), and a remote `istore` involves a single message.

Given the high message cost on the CM-5′, it is advantageous to treat local messages as a special case in software if their frequency is non-negligible, *i.e.*, to exploit `send` to local frames and `ifetch` and `istore` to local structures. The required destination check costs two cycles per message. The rightmost section of Table 1 shows the resulting costs for local messages. This optimization was tried on the J-Machine and found to hurt performance, so the general SEND mechanism is used for all messages. Local messages on the CM-5′ are cheaper than on the J′-Machine because the data does not need to be moved in and out of the network interface. The local `ifetch` and `istore` entries show that optimizing for local messages can enable further optimizations: the access portion of the `ifetch` is performed directly in the thread and if the data is present the threads which use it can be enabled cheaply, as discussed in Section 6.

4.4 Discussion

The implementation costs highlight three key points: network access cost is far higher on the CM-5′ than on the J′-Machine, receiving is more expensive than sending on both machines, and optimizing for local messages can be valuable. On Simple, where 93% of the messages are remote, the CM-5′ spends three times as much time on messages as the J′-Machine. However, on Gamteb where 30% of the messages are local the CM-5′ spends only twice as much time.

The high network interface access cost on the CM-5′ is due to the node architecture which connects the network interface chip to the MBUS. As a result, all loads and stores take 7 cycles each (8 cycles for double-word loads or stores). Further, checking the status register after every send accounts for $1/3$ of the send cost. On the other hand, the J-Machine effectively connects the network interface to the ALU bus. The J-Machine faults if the outgoing send buffers overflow, so, while a fault handler is necessary to retry, the check is free. Asynchronous message reception is prohibitively expensive on the CM-5′ due to poor support for user-level interrupts in the Sparc.

Several factors cause message reception to be more expensive than sending. Sending is synchronous to the computation whereas reception is logically asynchronous. Thus values to be sent are typically available in registers, whereas values received must be stored in memory until the waiting

TAM operation	Cycles					
	J′-Machine		CM-5′			
	All Messages		Remote Messages		Local Messages	
	sync.	async.	sync.	async.	sync.	async.
Send N-word message	$7+N$	$7+N$	$25+4\lceil\frac{N}{2}\rceil$	$25+4\lceil\frac{N}{2}\rceil$	$2+N$	$4+N$
Receive N-word message	$5+7\lceil\frac{N}{2}\rceil$	$5+7\lceil\frac{N}{2}\rceil$	$13+12\lceil\frac{N}{2}\rceil$	$\approx 100+12\lceil\frac{N}{2}\rceil$	$2+4\lceil\frac{N}{2}\rceil$	$4+4\lceil\frac{N}{2}\rceil$
Poll	4	—	9	—	—	—
Ifetch message	24	24	93	≈ 190	16 or 4†	18 or 4†
Istore message	15	15	60	≈ 160	2 or 0†	4 or 2†

†: Cost when the element is present for an `ifetch`, or empty for an `istore`.

Table 1: *Local cost for sending and receiving messages.*

computation can be scheduled.[2] Reception involves a dispatch, while sending does not. Finally, stores to the CM-5′ network interface can take advantage of the write buffer, while loads see the full latency. On the MDP, the send buffer can be written to at the rate of two words per cycle through the SEND instruction, while reading a word from the message queue takes 3 normalized cycles, the same as any on-chip memory access.

An intimate processor/network coupling reduces the cost of actual message transmission, but we must also account for the cost of dynamic scheduling introduced by messages, since they must be integrated with the rest of the program. Optimizing for local messages raised this issue, which the following sections address in more detail.

5 Dynamic Scheduling and Control Cost

This section examines the cost of control and shows how it is influenced by a combination of hardware mechanisms: prioritized scheduling and dispatch, multiple contexts, and atomic operations. While in a more conventional setting, we would be concerned only with the costs of jumps and branches, in the context of fine-grained parallel programs, the flow of control in the program is closely tied to the asynchronous arrival of messages, so we must also be concerned with the costs of dynamic scheduling and synchronization.

5.1 TAM requirements

The bottom three segments in Figure 2 show the relative cost of control. In TAM control is realized with the `fork`, `switch`, `post`, and `swap` instructions. Basic control flow is described by the `fork` instruction, which attempts to enable a thread in the current frame, and `switch`, which is a conditional `fork` to one of two threads. If a thread is synchronizing, it has an associated synchronization counter in the frame. `Fork` will decrement the counter and, if the counter becomes zero, enable the thread. When a `fork` succeeds then the thread address is pushed onto the LCV, the structure that holds the list of threads that are ready to run for the current frame. At the end of every thread is a `stop` instruction, which pops the next enabled thread and transfers control to it.

In addition, threads can be enabled from inlets by a `post` instruction. Whereas `fork` enables a thread for the currently running frame, `post` enables a thread for the frame associated with the message, whether it is currently running, idle, or ready. A `post` of a synchronizing thread requires decrementing the synchronization counter, just as `fork` does. If the `post` enables the thread, then it must check the state of the frame. If the frame is idle, it is made ready and placed on the frame queue. Finally, the `swap` instruction schedules the next frame from the ready queue.

Observe that inlets and threads cooperate in determining the flow of computation, but inlets may preempt threads to handle incoming messages. The two levels of scheduling share the synchronization counters, the continuation vector, and the frame ready queue. Thus, the implementation must guarantee that either thread and frame scheduling operations (*e.g.*, `fork`, `stop`, and `swap`) are atomic with respect to `post` instructions or that threads and inlets use distinct resources.

5.2 Hardware support

Recall that the J-Machine provides distinct priority levels for threads and inlets with separate register sets, and upon message arrival the hardware dispatches to the inlet, unless interrupts have been explicitly disabled. Since there are no read-modify-write operations on memory, disabling interrupts becomes the primary means of ensuring atomicity. The CM-5 has a single user level, uses a single register set, and, as described in the previous section, uses a synchronous model for receiving messages.

5.3 Implementation

The implementation of the `fork`, `stop`, and `swap` instructions is basically the same on both machines. The

[2] Moving computation into the inlet, as in a message-driven model, does not help because it replaces memory accesses for storing message data with memory accesses for retrieving local operands.

translator specializes most forks into fall throughs and branches, which eliminates the corresponding `stop` at the end of the thread. The remaining forks translate into a push onto the LCV. A pointer to the top of the LCV is kept in a register. The top portion of Table 2 shows the cost of the specific code sequences, depending on the position of the `fork` in the thread and whether the destination thread is synchronizing. Taking the dynamic frequencies of these cases into account, the average cost of a `fork` is about 8 cycles on both the CM-5′ and J′-Machine. Nonetheless, forks account for roughly half of the cost of control operations. Both machines implement synchronization counters as locations in the frame which must be initialized to the entry count of their associated thread. This incurs the cost of a memory write.

The instruction sequence executed for `post` depends on whether or not the target thread is synchronizing and on the state of the frame. When the frame is idle or ready, the CV pointer is contained in the frame, not a register. Also, if the frame is idle it must be placed on the ready queue. However, if the thread is being posted to the running frame, it can simply be pushed onto the LCV, like a fork. On the J′-Machine, the cost of a `post` to a running frame is higher because the register holding the pointer to the top of the LCV is stored in the low-priority (thread) context, causing a slight penalty for high-priority (inlet) code access. Similarly, determining if the `post` is for the running frame incurs the cost of accessing the other register context.

5.4 CM-5: Compiler-controlled message reception

The CM-5 translator inserts polls into the threads to allow for message reception. The poll incurs a cost of 9 cycles; however, by placing the polls in the appropriate places, it ensures that `fork`, `stop`, and `swap` run atomically relative to `post`. Eliminating the cost of polling by implementing an interrupt-based approach on the CM-5 is impractical for fine-grained parallelism. This is due to the high cost of an interrupt, about the same as 10 polls.

The cost of a `post` is reduced further in inlets that handle an `ifetch` response. If the element being fetched is local and present (see Section 6), the response inlet is inlined into the thread. This eliminates all the inlet overhead and as a result turns the `post` into a `fork`. The result is that the average `post` instruction takes between 9 and 13 cycles depending on the application program.

5.5 J-Machine: Asynchronous message reception

The J-Machine supports two-level scheduling directly, with threads interrupted by the automatic dispatch to an inlet. The challenge with this approach is ensuring that the `fork`, `switch`, and `swap` instructions run atomically.

TAM operation	J′′ cycles	J′ cycles	CM-5′ cycles
Fork a thread			
Fall through	0	0	0
Branch to thread			
unsynchronizing	1	1	1
successful sync.	4	4	4
unsuccessful sync.	13	13	13
Push thread onto CV			
unsynchronizing	5	5	5
successful sync.	10	10	10
unsuccessful sync.	7	7	7
Switch one of two threads	fork+2	fork+2	fork+2
Stop (pop thread from CV)	5	5	5
Poll	0	4	9
Initialize sync. counter	4	4	4
Post a thread from inlet			
Idle frame			
unsynchronizing	19	19	19
successful sync.	24	24	23
unsuccessful sync.	11	7	7
Ready frame			
unsynchronizing	15	15	14
successful sync.	20	20	19
unsuccessful sync.	11	7	7
Running frame			
unsynchronizing	31	10	8
successful sync.	31	15	12
unsuccessful sync.	27	7	7

Table 2: *Cost of TAM synchronization and scheduling instructions. J′′-Machine represents the asynchronous J′-Machine, discussed in Section 5.5. For most operations, several different versions reflect the various compiler optimizations or runtime conditions, such as whether a* `fork` *can be combined with a* `stop` *into a branch, whether the target thread is synchronizing, and whether the synchronization was successful or not.*

The first approach we adopted was to restrict the use of shared resources, i.e., the synchronization counters and CV pointers. In this approach, each inlet contains code to check whether the inlet frame is the same as the currently-running frame. If so, the synchronization counter and the thread address are pushed onto a special stack which does not interfere with thread execution. When the CV is empty, these posts are processed. Message interrupts are disabled when the special stack is being cleared. Swapping frames is made atomic by explicitly disabling interrupts.

This approach had the advantage of leaving interrupts disabled a minimal amount of time, shortening the waiting time of incoming messages, meaning that requests are serviced more quickly and the queue is less likely to overflow.

However, the cost of a `post` to a running frame (including the subsequent processing) was considered unacceptably high, as shown in Table 2. The cost of an unsuccessful `post` is also higher since the status of the frame must be checked before the synchronization counter is decremented and tested.

By leaving interrupts disabled for a significant proportion of thread execution, the J-Machine can almost be thought of as using polling instead of being fully message-driven. Setting or resetting the interrupt flag on the MDP takes two one-cycle instructions, which makes the J-Machine "poll" instruction 4 cycles.

5.6 Discussion

Control in fine-grained parallel programs involves the integration of asynchronous events with the control flow internal to the computation. In TAM, the two are closely related as the compiler generates the code for both scheduling levels. With this kind of coupling, the compiler can partition the available registers by convention; hardware partitioning into distinct contexts can restrict how registers can be used and can prevent certain optimizations.

The close relationship between the two levels also requires that asynchronous scheduling be complemented by efficient atomic operations on shared resources. Polling for messages avoids this issue and is acceptable for fine-grained parallelism. For instance, in our benchmark programs, polling accounts, on average, for 4% of execution time. It may, however, constitute unnecessary overhead for coarser-grained computation.

In hindsight, it appears that instead of focusing resources on multiple register contexts, it would be more advantageous to provide support for atomicity and fine-grained control operations, like `fork`, which contribute as much as 40% to the cost of running a program.

6 Heap access cost

The I-structure memory model used in TAM requires split-phase access to synchronizing data structures in the global heap. Implementing this memory model has multiple facets: (i) to access a remote heap location involves generating a request message, (ii) to perform synchronization on memory *elements*, each is augmented by a few tag bits which must be checked and updated on every access, and (iii) remote and suspended accesses must be delivered to the computation when they complete. The costs of the first and third parts were described in Sections 4 and 5, respectively. This section focuses on the middle portion and considers whether J-Machine hardware support for tags is beneficial.

6.1 Requirements

In TAM, the heap consists of 64-bit elements, each with a small tag. The tag indicates whether the element is empty, holds data, points to a list of waiting (deferred) readers, or holds a thunk which must be evaluated to yield the data value. The exact state transitions follow Id90 I-structure semantics and permit synchronization on a per-element basis. The discussion in this section focuses on I-structures, but it applies equally to other global heap structures (*e.g.*, M-structures). Three issues arise in implementing I-structure operations: representing the presence bits, checking and updating their state on every access, and maintaining the lists of deferred readers.

6.2 Hardware support

The J-Machine provides tagged memory in hardware to support dynamic typing and dynamic synchronization. Each 32-bit memory word is augmented with 4 tag bits, and instructions may trap on certain tag values. TAM does not require dynamic typing, so it cannot demonstrate the benefit of this use of tags. (A significant component of the high-level compilation process from Id90 to TAM is type inference and resolution of overloading to eliminate the need for dynamic typing.) Tags are used to support the I-structure state transitions, without binding the specific semantics of I-structures in hardware.

6.3 Implementation

On the J-Machine, each I-structure element uses two words and one four-bit tag which will trap on access to futures (empty locations), transferring control to a handler that enqueues deferred reads in 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 occurs. When an I-structure is allocated each element's tag must be set to empty, which requires one store per element.

On the CM-5, the tags are stored in a memory area disjoint from the data area. One tag byte is allocated for each 8-byte I-structure element, which allows 8 tags to be cleared at once. All I-structure accesses must explicitly check the tag byte before reading or writing the data. Deferred reads are handled similarly as on the J-Machine.

6.4 Discussion

Table 3 shows the costs of I-structure heap accesses in detail. The simple case of reading a present data element is where hardware tags exhibit some advantage. On the CM-5, the tag check is half of the access cost in this case. With tags, the trap occurs on deferred reads and all stores. Without tags, these cases require only a branch. So the deciding factor is the frequency of the simple case. We find that this differs radically on different programs. On Simple

TAM operation	Cycles	
	J′-Machine	CM-5′
I-fetch		
Data present	6	9
Defer	48	41
I-store		
Cell empty	14	15
Deferred readers	17	22
I-structure allocate (N words)		
Local/remote policy	—	7
Allocate	18	18
Clear tags	$5N$	$.75N$

Table 3: *Access to data structures with synchronization on a per-element basis. The costs shown reflect the memory access including checking and updating the tags. I-structure allocation includes initializing all tags to empty.*

there are more than seven ifetches per istore, and only 3% of the ifetches are deferred. On Gamteb, there are only 1.6 ifetches per istore, and 24% of the ifetches are deferred.

The other difference between the two implementations is the often neglected time to allocate an I-structure, which includes initializing all tags to empty. On the J-Machine, each element must be set separately to the future-tagged value denoting an empty unrequested element. On the CM-5, the tags of eight elements can be initialized using a single store-double instruction.

Returning to the cost breakdown in Figure 2 we see that the cost of actually accessing the heap is only a fraction of the total cost. Once the cost of the message component and the control component are included, it becomes apparent that the primary factor in determining the utility of the hardware mechanisms supporting the global heap is the ratio of local to remote accesses. On remote access, the message handling overhead diminishes the impact of fast tag checking. While treating the local access specially in software reduces both the message and control overhead, it incurs the cost of the tag check. However, when at least 30% of accesses are local, the reduction in control overhead balances the difference in cost of sending messages on the CM-5′.

7 Summary

We have compared the performance of the J-Machine, a recent experimental architecture with several novel mechanisms that support fine-grained parallelism, and the CM-5, a recent commercial architecture using conventional Sparc processors, on fine-grained parallel programs written in Id90. A complete quantitative comparison in this regime is very difficult because there are so many variables that can influence performance and there is little consensus on what constitutes a representative workload. We follow a method of analysis similar to the Abstract Machine Characterization Model used to evaluate a wide range of conventional machines and benchmarks [12]. We normalize for software effects, including programming language, programming style, and high-level compiler optimizations by using a common low-level representation of each program in terms of a Threaded Abstract Machine. Machine-specific optimizations are realized in compiling the TAM code to each machine. Examination of the generated code yields the cost for each TAM primitive. An instrumented version of the program is run to produce roughly a hundred dynamic statistics. These are combined with the machine cost coefficients to obtain the average cycles per TAM instruction.

We look forward to a much broader set of studies following a similar methodology using additional machines and additional language frameworks. Clearly it should be possible to evaluate proposals such as *T [11] in this framework. Although other parallel language implementations may differ from TAM in many ways, the primary ingredients are likely to be similar: message exchange, remote references, synchronization, control flow, and dynamic scheduling. We do anticipate that the granularity of parallelism will have a significant impact on the evaluation. Coarser grained models will not stress the message handling and dynamic scheduling as heavily.

In this comparison we found that traditional architectural issues, such as the average memory access time and the floating point performance had a substantial impact on the performance of the experimental architecture. After correcting for these factors, the fast message send and receive are a clear gain, accounting for as much as a 50% improvement in some programs. Treating local messages as a special case in software compensated for larger message overhead in many programs because it also reduced the control overhead of dynamic scheduling; however, the resulting system is more sensitive to variations in the remote reference frequency. The fast message dispatch mechanism was of modest value in the absence of adequate atomic operations on resources that are shared between the primary computation and the message handlers which operate on its behalf. Multiple disjoint register sets were not of particular value, since code was generated for the different scheduling levels from a single program. The compiler could simply partition the register file, which allows a tight integration of the two levels. The utility of tagged memory was undercut by the high cost of initialization.

Our measurements suggest some directions in the design of parallel computers:

- The network interface should be integrated with the processor register file or cache.
- Fast dispatch to user-level message handlers significantly reduces communication overhead. However, careful attention must be paid to atomicity and sharing between the message handlers and the on-going computation.
- Fine-grained parallel programs are control intensive. When the communication and memory requirements are adequately addressed, this stands out as the primary avenue for further advancement.
- While individual mechanisms need to be efficient in isolation, the interactions among the mechanisms must be carefully considered.

The general observation is that in evaluating novel architectures it is essential to carry the implementation of programming languages to completion, as that is the only way to perceive the completeness and the synergy between the various mechanisms and to determine the relative importance of each component.

Acknowledgments

We are grateful to the anonymous referees for their valuable comments. We would also like to thank Fred Chong, Richard Lethin, and Nate Osgood for their comments on earlier versions of this paper. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Funding at MIT was provided in part by the Defense Advanced Research Projects Agency under contracts N00014-87K-0825, F19628-92-C-0045, and N00014-91-J-1698 and in part by a National Science Foundation Presidential Young Investigator Award, Grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T. Ellen Spertus is supported by a NSF Graduate Fellowship. 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. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172.

References

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

[2] 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. ESCOM (Leider), Trento, Italy, September 1988.

[3] D. H. Bailey et al. The NAS Parallel Benchmarks — Summary and Preliminary Results. In *Proc. Supercomputing '91*, November 1991.

[4] 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.

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

[6] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991. (Also available as Technical Report UCB/CSD 91/591, CS Div., University of California at Berkeley).

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

[8] S. C. Goldstein. Implementation of a Threaded Abstract Machine on Sequential and Multiprocessors. Master's thesis, Computer Science Division — EECS, U.C. Berkeley, 1993. (In preparation, to appear as UCB/CSD Technical Report).

[9] J. Gustafson, G. Montry, and Benner R. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9, 1988.

[10] R. S. Nikhil. ID Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, 1991.

[11] 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. Sci., 545 Tech. Square, Cambridge, MA, January 1991.

[12] R. H. Saavedra-Barrera and A. J. Smith. Benchmarking and The Abstract Machine Characterization Model. Technical Report UCB/CSD 90/607, U.C. Berkeley, Computer Science Div., November 1990.

[13] 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).

[14] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[15] E. Spertus. Execution of Dataflow Programs on General-Purpose Hardware. Master's thesis, Department of EECS, Massachusetts Institute of Technology, 545 Tech. Square, Cambridge, MA, August 1992. To be expanded and released as MIT AI Lab Technical Report 1380.

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