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

Ellen Spertus
William J. Dally

MIT Artifical Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

{ellens,billd}@ai.mit.edu
617-253-7710, 617-253-6043

Seth Copen Goldstein
Klaus Erik Schauser
Thorsten von Eicken
David E. Culler

Computer Science Division — EECS
University of California
Berkeley, CA 94720

{tam}@cs.berkeley.edu
510-643-7572

**Abstract**

Experimental and commercial parallel machines have matured to a point where it is possible to quantify the performance enhancement due to the novel mechanisms supporting fine-grain parallel programs in the experimental machines. The MIT J-Machine provides a register-level message send, autonomous message receive and dispatch, a prioritized scheduler with multiple contexts, and tagged memory with fast traps. The Thinking Machines CM-5 is essentially a collection of conventional processors with user-level messages. However, it is difficult to establish a meaningful workload for a comparison that adequately accounts for compiler optimizations which specifically address the lack of available mechanisms.

This paper uses an abstract machine approach in a relative evaluation of the J-Machine and CM-5. High-level parallel programs are translated by a single optimizing compiler to a fine-grained abstract parallel machine, TAM. The final compilation step is unique to each architecture and optimizes for specifics of the architecture. Determining the cost of each primitive on the machines weighted by their dynamic frequency in parallel programs, provides a quantitative analysis of the effectiveness of the mechanisms.

The analysis addresses the individual mechanisms and their synergy. Efficient processor/network coupling proves valuable. The J-Machine message dispatch is reduced in value by the lack of support for atomic operations necerssary for the scheduling levels to co-operate. Multiple hardware contexts are of small value when the contexts co-operate and the compiler can partition the register set. Tagged memory provides little gain. The performance of the overall system is strongly influenced by the performance of the memory system. Fine-grain parallel programs are control intensive.

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

**Abstract**

Experimental and commercial parallel machines have matured to a point where it is possible to quantify the performance enhancement due to the novel mechanisms supporting fine-grain parallel programs in the experimental machines. The MIT J-Machine provides a register-level message send, autonomous message receive and dispatch, a prioritized scheduler with multiple contexts, and tagged memory with fast traps. The Thinking Machines CM-5 is essentially a collection of conventional processors with user-level messages. However, it is difficult to establish a meaningful workload for a comparison that adequately accounts for compiler optimizations which specifically address the lack of available mechanisms.

This paper uses an abstract machine approach in a relative evaluation of the J-Machine and CM-5. High-level parallel programs are translated by a single optimizing compiler to a fine-grained abstract parallel machine, TAM. The final compilation step is unique to each architecture and optimizes for specifics of the architecture. Determining the cost of each primitive on the machines weighted by their dynamic frequency in parallel programs, provides a quantitative analysis of the effectiveness of the mechanisms.

The analysis addresses the individual mechanisms and their synergy. Efficient processor/network coupling proves valuable. The J-Machine message dispatch is reduced in value by the lack of support for atomic operations necerssary for the scheduling levels to co-operate. Multiple hardware contexts are of small value when the contexts co-operate and the compiler can partition the register set. Tagged memory provides little gain. The performance of the overall system is strongly influenced by the performance of the memory system. Fine-grain parallel programs are control intensive.

# 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[17] focuses on Id90[15], J-Machine[10, 11] on CST[13], Alewife[1] on Mul-T, CM-5[20] on Fortran90, and Dash[14] 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, all of the machines are universal enough to support any of the other language paradigms, and all of the machines 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 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 which they are intended to evaluate[19, 12] or specifically avoid emerging languages which novel mechanism may bring into practical reach[4]. 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, and does not 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 powerful compilation system exists. The compiler performs a variety of high-level optimizations in translating the language down to code for a simple abstract machine, TAM[8]. 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 on a suite of programs. The J-Machine essentially provides direct hardware support for every aspect of TAM, although 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.[1]

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 significant impact on execution time for the two machines, but for which conventional intuition (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 that interact in support of dynamic scheduling related to asynchronous message arrival. 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

This section provides background for the study. The two machines are described briefly; more details are presented later as the study focuses on particular aspects. An overview of the Threaded Abstract Machine is presented to define the requirements placed on the machines and TAM-level dynamic measurements are provided for the set of benchmark programs used in the study.

### 2.1 CM-5

The CM-5 is a massively-parallel MIMD computer based on the Sparc processor, interconnected in two identical disjoint "hypertree"[2] 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. (In some CM-5 installations the node also contains four 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-word by 36-bit static memory, a closely integrated network interface, a packet

[1] To further control for experimental bias, some of the authors have worked extensively on the J-Machine and others have worked extensively on the CM-5. We have carefully examined each others implementations to make sure that neither is missing an implementation trick.

[2] More precisely, a Benés network with four-to-one concentration, also described as an incomplete fat tree.

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, symbols, 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 256-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. When a message is executed, its first word is loaded into the instruction pointer to start execution of a message handler, and a pointer to the base of the message is loaded into an address register so that subsequent words may be accessed. The handler runs until an explicit **suspend** instruction is executed, at which time control passes to the next message in the queue of same priority, or the next lower priority context is resumed.

### 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 differences with "thread packages" are that it is 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 of unbounded latency occurs between them.

The TAM execution model centers on the *activation frame*, which is the analog of a stack frame for parallel calls, as suggested by Figure 1. To invoke a code-block, a frame is allocated on a processor, 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 executed. To detect the completion of antecedents, a synchronization counter 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 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, its enabled threads are executed until the CV 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.

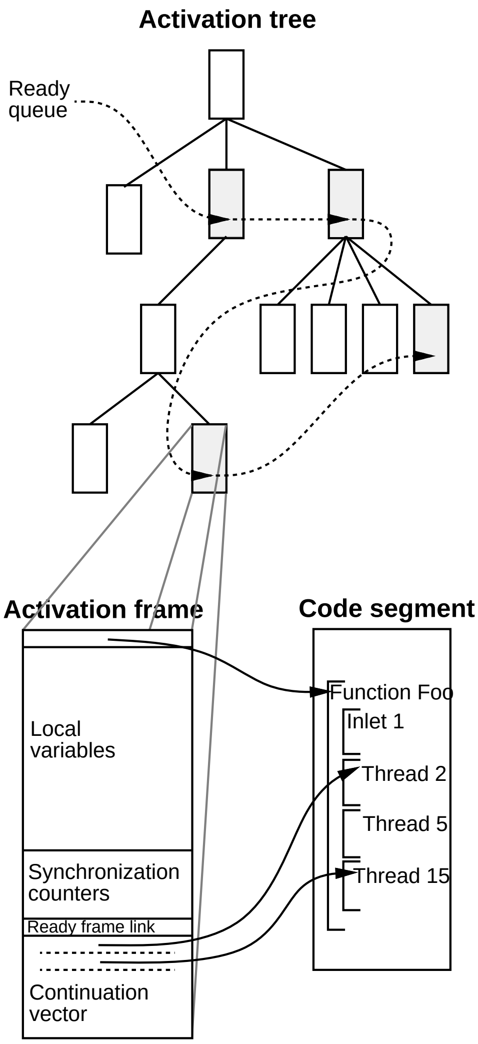


Figure 1: *TAM activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame holds local variables, counters used to synchronize threads and inlets, and the continuation vector which contains the addresses of all currently enabled threads of the activation. On each processor, all frames holding enabled threads are linked into a ready queue.*

Global data structures in TAM provide synchronization on a per-element basis to support I-structure and M-structure semantics [5, 15]. In particular, reads of empty elements are deferred until the corresponding write occurs. I-structures support write-once semantics while M-structures allow multiple writes to an element. Accesses to the data structures are split-phase and send a request message to the processor containing the data. The remote processor services the request while local computation proceeds, and responds with a message that is received by an inlet. Accesses to I-structures are performed via special instructions: `ifetch` reads an element and 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 (TL0 V2.1), instructions are primarily three address, where the operands are constants, registers, or frame locations. TL0 registers and frame slots are statically typed into integers, floats, various pointers, and generals. Generals are sufficiently large to contain any 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 TL0 registers, although the compiler tries to use them as efficiently as possible. The translator from TL0 to a target machine is responsible for mapping TL0 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 remote requests or calls, dynamic synchronization of producers and consumers, and split-phase remote operations allowing computation to overlap 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[9]. 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 that maintain a set of bins, holding a linked list of available memory blocks of fixed size.

The J-Machine implementation of TAM 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, running inlets to handle the messages does not interfere with thread execution. As shown in Figure 2, 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 TL0 instructions. The remaining general-purpose register is used to hold one TL0 register. All other TL0 registers are mapped to the base of on-chip memory, a region that can be addressed easily. Frames are stored off chip in DRAM.

A similar approach can be 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. As shown in Figure 2, the CM-5 implementation uses 32 registers divided into three classes: *global registers* which hold frequently-used values, *TL0 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 TL0 variables as possible in the TL0 registers and spills the rest into the frame. 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).

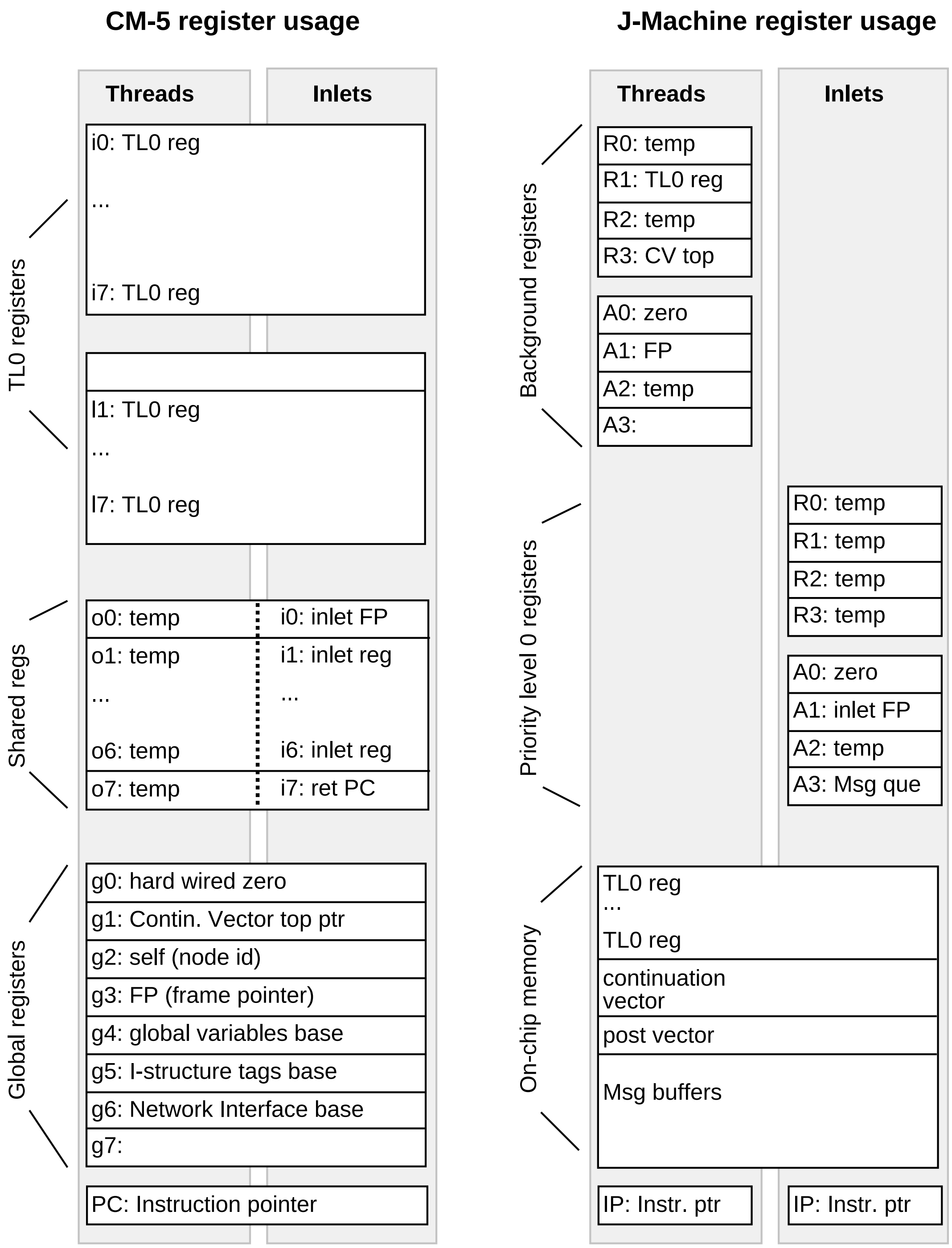


Figure 2: *Register set usage. A single register window is used on the CM-5 to provide registers for global values, TL0 variables, temporaries, and inlets. The J-Machine uses two distinct register sets for threads and inlets with very few registers each.*

## 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. These are grouped into the basic instruction categories in Figure 3. *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.

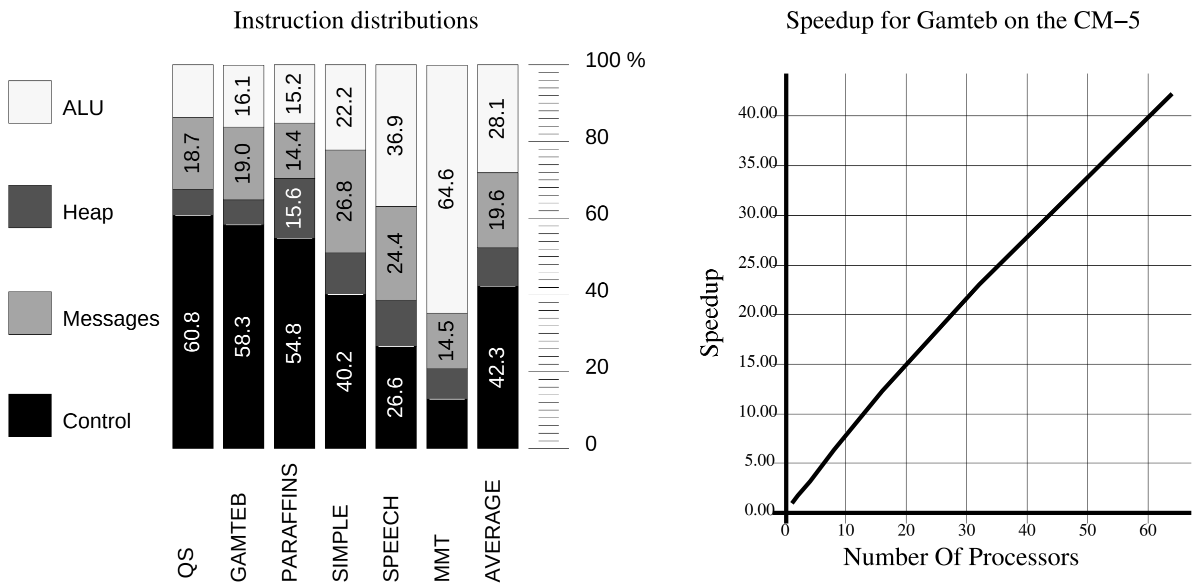


Figure 3: *Instruction mix statistics for the benchmark programs and speed-up obtained on gamteb with up to 64 CM-5 nodes.*

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 [6]. It is highly recursive with many conditionals. *Paraffins* [3] enumerates the distinct isomers of paraffins. *Simple* is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id90 [7, 2]. *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 final column shows the arithmetic mean of the distributions. Figure 3 also shows the speed-up obtained on gamteb with up to 64 processors.

The programs toward the left of Figure 3 represent fine-grain 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.

# 3 Baseline Architectural Issues

By associating a cost with the implementation of each TAM primitive we can obtain an estimate of how time is spent on each machine from the dynamic statistics and isolate the contribution of specific mechanisms. However, there are several significant engineering differences between the study machines including cycle time, 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 contributions. 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. Also, many design choices were based on speculation of the requirements of fine-grained parallel programs, because it was impossible to obtain adequate empirical data at the time. 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 artifactual differences.

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 4 and explained below. The metric used is cycles per TL0 instruction (CPT). The bars show the contribution to the CPT resulting from each of the classes of TL0 instruction. Two new bars have been introduced to highlight important implementation issues. The Operand bar at the top reflects the memory access penalty in bringing data into registers for ALU instructions. The Atomicity bar 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 bars, because these have to do with conventional processor efficiency. The remaining sections examine the lower bars, which involve issues that are unique to fine-grained parallelism.

## 3.1 Cycle time

The Cypress Sparc processor in the CM-5 has a cycle time of 30ns while the cycle time of the MDP in the J-Machine is 62.5ns. The Sparc is a full custom design with a four stage pipeline and product-level engineering investment. The MDP is implemented from standard-cells and its instruction execution is unpipelined. This results in differences in execution time on the two machines, but does not offer any architectural insight. (In fact, neither clock rate is impressive by current standards.) We see no reason why the novel mechanisms in the J-Machine fundamentally impact the cycle time. Certain aspects of the J-Machine instruction set frustrate pipelining, but these do not appear to be particularly valuable in hindsight, so we will eliminate them below. With this assumption, we conduct the study entirely in units of cycles, assumed to be the same on the two machines, with SRAM and DRAM access times expressed as fixed multiples of cycles.

It should be noted that we are looking at the efficiency of the processor in carrying out a fixed workload. In the analysis, we assume the performance characteristics of the network itself to be identical in the two

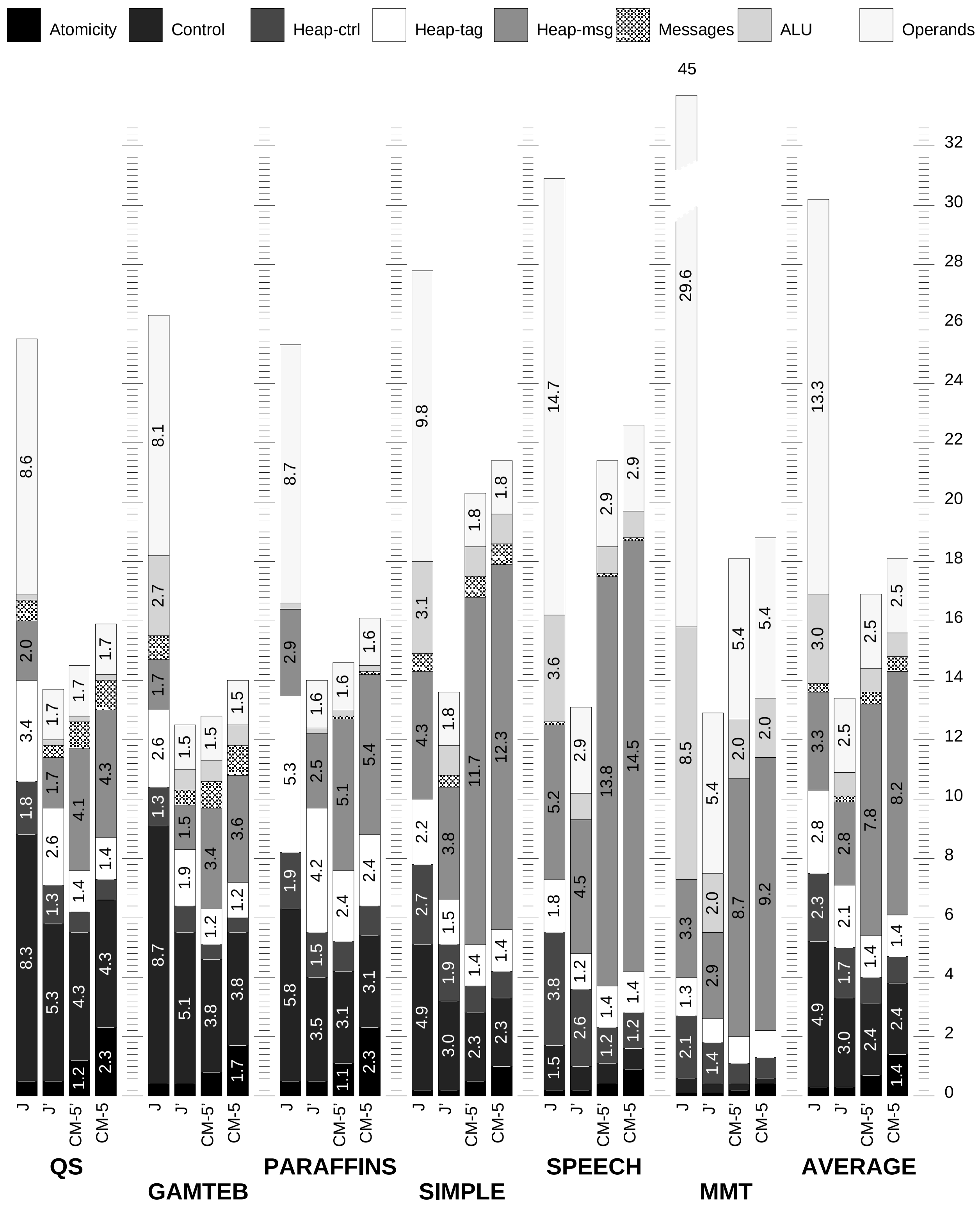


Figure 4: *Relative performance of the J-Machine, the CM-5 and their hypothetical variations on the sample dynamic instruction mix. The variations are used in the study to compensate for artifactual differences in engineering which do not offer any new architectural insight.*

machines and to effect the processor in a similar way. Similarly, we assume idle time resulting from inadequate parallelism or improper load balancing to affect the two machines to the same extent.

### 3.2 Floating point

Most TL0 arithmetic and logical operations map into a single machine instruction on the Sparc and the MDP, once the operands have been properly positioned. A few TL0 instructions such as `abs, max,` and `min` require short instruction sequences. Integer divide, and multiply on the Sparc, are implemented by moving the operands into machine registers and calling a library routine. However, the MDP has no built-in floating-point hardware, so software routines are used for floating-point arithmetic. MDP single-precision floating-point libraries typically take 12–28 cycles. It has a user-defined tag type to represent floating-point numbers, but since Id90 is statically typed, no benefit is gained from this mechanism. On the Sparc, floating point operations execute on a co-processor and take roughly seven cycles, unless overlapped with integer execution.

The lack of floating point or co-processor interface on the J-Machine is motivated partly by the desire to have a very small "jelly bean" node and partly by the desire not to spend time on that well-understood aspect of computer engineering. In current technology, the FPU would fit on-chip, and the result presumably still qualifies as a commodity part. The J′-Machine is assumed to have floating point performance identical to that of the Sparc.

### 3.3 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 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 cycle is required to read the on-chip SRAM and 12 for off-chip DRAM.[3] 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.

Since the SRAM on the J-Machine is managed explicitly, it can only be used to hold explicitly specified information, whereas data is moved into the CM-5 cache on demand. Most of the J-Machine on-chip memory holds library routines and program code. (We assume for the analysis that all the program fits on-chip, although this is optimistic.) A few frequently used data objects are also stored on chip: the continuation vector of the resident frame, TL0 registers that could not be mapped to J-Machine registers, and the message queue. However, frame accesses are off-chip to DRAM. This results in a significant memory access penalty, as shown by the top bar in Figure 4. With a reasonable amount of additional engineering, the J-Machine SRAM could be organized as a cache. Thus, the J′-Machine is assumed to have a memory structure identical to the Sparc. Both are assumed to have a 5% miss rate on memory accesses. The J′-Machine is also assumed to have a large register set, like the Sparc, so values that are entirely local to a thread can be kept in registers.

[3] Actually the J-Machine can access DRAM in 6 cycles, but we have assumed that the processor clock rate is doubled, so we must double the number of cycles to DRAM.

### 3.4 Message Size

The TL0 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. However, the CM-5 limits the message size to five 32-bit words. Thus, large messages must be broken into smaller messages and reconstructed via multiple inlets. The use of multiple inlets overcomes the lack of in-order delivery on a stream of point-to-point packets. As a result, each packet in a long message incurs the overhead of specifying the destination frame, and inlet number.

This limitation is due to space constraints in the network interface chip at the time it was designed; the network is capable of delivering larger packets and current gate-array densities would permit larger FIFO queues in the interface chip. Thus, the CM-5′ is assumed to be able to send larger packets, say 16 words. This is sufficient to hold the biggest message generated in our benchmarks.

### 3.5 Polling

The current implementation of the CM-5 has a bug in the network interface which affects the cost of polling the network: the two data networks must be polled individually, which doubles the cost of polls. We are assuming that both data networks can be polled simultaneously in the CM-5′.

### 3.6 Discussion

Correcting for the memory system and lack of floating point results in a significant improvement of the J-Machine performance. After the correction, the time spent in arithmetic operations and memory access is very close on the two machines. The message, control, and heap components become the dominant 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 MBUS on each node.

### 4.1 TAM requirements

The *messages* bar in Figure 4 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 TL0 these messages are formed using a `send` instruction which sends a number of data values to an inlet of a potentially remote frame. In the inlet a `receive` instruction extracts the data from the message and stores it into the frame. The adjacent segment of the *heap* bar reflects the cost of 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 causes 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 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 copying messages into an on-chip queue, and dispatching to 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 and store it into frame locations from there. On the CM-5 the network interface status register is polled periodically and 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, followed by stores into the frame.

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 small windows 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. 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, where $N$ is the number of words in the message. The start-up cost for sending a message to a remote processor is four times higher on CM-5′, 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

| TAM operation | Cycles | | | |
|---|---|---|---|---|
| | J′-Machine | | CM-5′ | |
| | sync. | async. | sync. | async. |
| Remote messages | | | | |
| Send $N$-word message | $7{+}N$ | $7{+}N$ | $27{+}4\lceil\frac{N}{2}\rceil$ | $27{+}4\lceil\frac{N}{2}\rceil$ |
| Poll | 4 | — | 9 | — |
| Receive $N$-word message | $6{+}6\lceil\frac{N}{2}\rceil$ | $6{+}6\lceil\frac{N}{2}\rceil$ | $14{+}12\lceil\frac{N}{2}\rceil$ | $100^{\dagger}{+}12\lceil\frac{N}{2}\rceil$ |
| Ifetch messages | 21 | 21 | 93 | $190^{\dagger}$ |
| Istore message | 12 | 12 | 60 | $160^{\dagger}$ |
| Local messages | | | | |
| Send $N$-word message | $7{+}N$ | $7{+}N$ | $4{+}N$ | $6{+}N$ |
| Receive $N$-word message | $6{+}6\lceil\frac{N}{2}\rceil$ | $6{+}6\lceil\frac{N}{2}\rceil$ | $2{+}4\lceil\frac{N}{2}\rceil$ | $4{+}4\lceil\frac{N}{2}\rceil$ |
| Ifetch messages (present) | 21 | 21 | 6 | 8 |
| Ifetch messages (deferred) | 21 | 21 | 12 | 14 |
| Istore message | 12 | 12 | 2 | 4 |

†: approximate values.

Table 1: *Cost for sending and receiving messages.*

are received back-to-back per interrupt. 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 specialize code generation to exploit `send`s to local frames and `ifetch`es and `istore`s to local structures. The required destination check costs two cycles per message. The lower 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 due to two factors: the data does not need to be moved in and out of the network interface, and the dispatch is a simple **call**. The local `ifetch` and `istore` entries show that optimizing for local messages can enable further optimizations: the access portion of the `istore` 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 J-Machine spends only 4.1 CPT on messages where the CM-5′ spends 13 CPT. This is 30% and 64% of the execution time, respectively. However, on Gamteb where 30% of the messages are local the J-Machine spends 2 CPT on messages where the CM-5′ spends 4.3 CPT, which is 16% and 33% of the execution time, respectively.

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). The J-Machine effectively connects the network interface to the ALU bus.

Also, checking the status register after every send accounts for $1/3$ of the send cost. 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 computation can be scheduled[4]. Reception involves a dispatch, while send does not. Finally, **store**s to the CM-5′ network interface can take advantage of the write buffer, while **load**s see the full latency; J′-Machine **send**s use both register file read ports whereas only one write port is available for **receive**s.

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 program control. 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. In a more conventional setting, we would be concerned only with the costs of jumps and branches. In the context of fine-grain parallel programs, the flow of control in the program is closely tied to the asynchonous arrival of messages, so we must also be concerned with the costs of dynamic scheduling and synchronization.

## 5.1 TAM Requirements

Program control flow in TAM 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 enable the thread if the counter becomes zero. When a `fork` succeeds then the thread address is pushed onto the local continuation vector (LCV) which 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 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 like `fork`. 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. The `swap` instruction schedules schedules the next frame from the ready queue.

Observe that inlets and threads co-operate in determining the flow of computation, but inlets may preempt

[4]Moving computation into the inlet, as in a message driven model, does not help because it replaces memory accesses to message data by memory accesses to local operands.

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`) be atomic with respect to `post` instructions or that threads and inlets use the distinct resources.

### 5.2 Hardware support

Recall, the J-Machine provides distinct priority levels for threads and inlets with separate register sets and hardware dispatch to the inlet on message arrival, unless interrupts have been explicitly disabled. This is the primary means of ensuring atomicity, there are no read-modify-write operations on memory. 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 `fork`, `stop`, and the `swap` instructions is basically the same on the two machines. The translator specializes most forks into fall throughs and branches, eliminating the `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 the CM-5′ and 9 on the J′-Machine. Nonetheless, forks account for roughly half of the cost of control operations on both machines, as indicated in Figure 5. 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 implementation of `post` is similar to that of `fork` and specialized to the target threads. However, it is somewhat more expensive because the inlet has a pointer to the frame in a register, but the pointer to the top of the the CV is in the frame itself. Additionally, if the frame is idle it is 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.

### 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` execute atomically relative to `post`. The implementation of `post` is straightforward, as described above, and has the costs shown in Table 2. The net result of specializing the post instruction is that the most common case, posting to a running frame, costs only 2 cycles more than a fork instruction. For example, in Gamteb the posts to a running frame account for 51% of the posts. This cost is reduced further for posts appearing 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

| TL0 operation | J″-Machine cycles | J′-Machine cycles | CM-5′ cycles |
|---|---|---|---|
| Fork a thread | | | |
| Fall through | 0 | 0 | 0 |
| Branch to thread | | | |
| unsynchronizing | 2 | 2 | 1 |
| successful sync. | 6 | 6 | 4 |
| unsuccessful sync. | 13 | 13 | 12 |
| Push thread onto CV | | | |
| unsynchronizing | 4 | 4 | 5 |
| successful sync. | 12 | 12 | 10 |
| unsuccessful sync. | 9 | 9 | 7 |
| Switch one of two threads | fork+4 | fork+4 | fork+2 |
| Stop and pop thread from CV | 5 | 5 | 5 |
| Initialize sync. counter | 4 | 4 | 4 |
| Post a thread from inlet | | | |
| Idle frame | | | |
| unsynchronizing | 38 | 36 | 19 |
| successful sync. | 47 | 41.5 | 24 |
| unsuccessful sync. | 16.5 | 16.5 | 7 |
| Ready frame | | | |
| unsynchronizing | 25 | 25 | 14 |
| successful sync. | 34 | 30.5 | 20 |
| unsuccessful sync. | 16.5 | 16.5 | 7 |
| Running frame | | | |
| unsynchronizing | 31 | 15.5 | 8 |
| successful sync. | 40 | 21.5 | 12 |
| unsuccessful sync. | 40 | 16.5 | 7 |
| Swap to next frame | $66 + 4t$ | 54 | $24 + 4t$ |

Table 2: *Cost of TL0 synchronization and scheduling instructions.* `Fork` *enables a thread from another thread,* `switch` *tests a boolean and enables one of two threads, and* `post` *enables a thread from an inlet. All three may first synchronize by decrementing a counter.* `Stop` *terminates the current thread and starts the next one. When no more threads are enabled in the running frame,* `Swap` *switches to the next one, copying each of the waiting threads $t$ into the CV. For most operations several different versions reflecting the various compiler optimizations. For instance, depending on 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.*

turns the post into a fork. The net result is that a post instruction takes between 9 to 12 cycles depending on the application program.

Reducing the cost of polling by implementing an interrupt based approach on the CM-5 is impractical for fine-grain parallelism. This is due to the high cost of an interrupt, about the same as 10 polls, and the 30 cycle cost of an atomic exchange instruction.

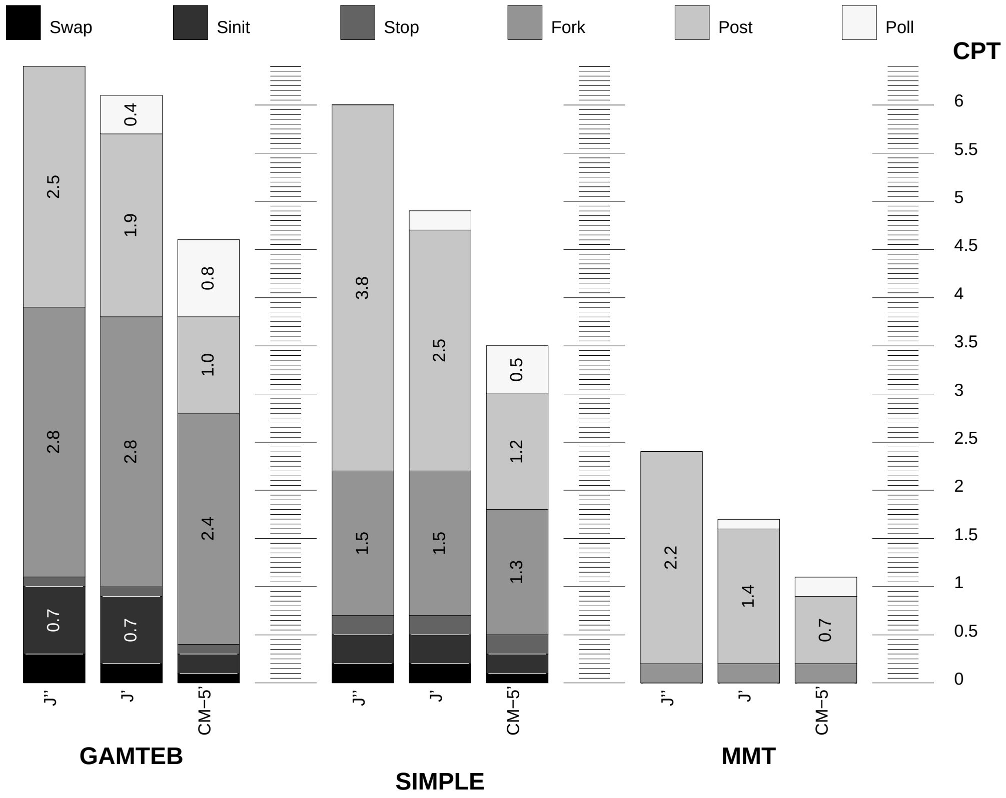


Figure 5: *This bar graph shows the contribution of control instructions to the cost of an average TAM instruction on each of the three architectures for our benchmark programs. In this figure, J′′-Machine represents the asynchonous J′-Machine. For J′-Machine and* CM-5′ *this figure presents the details of the control bar in Figure 4.*

### 5.5 J-machine: Asynchronous message reception

The J-machine implements the 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 execute atomically. 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 number 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.

While this approach has the benefit of leaving interrupts disabled for a minimal amount of time and of quick execution of the `fork` instruction, the cost of a `post` (including the subsequent processing) was unacceptably high at about 30 cycles. As can be seen in Figure 5, this approach is the most expensive for all the programs.

The approach used in producing Figure 4 is to ensure atomic execution of the TAM scheduling instructions by briefly enabling interrupts at the top of the thread. This brings the cost of the average post instruction down to 20 cycles. Observe that this simulates polling by permitting interrupts only during a small window. Setting or resetting the interrupt flag on the MDP takes two one-cycle instructions; which makes the J-Machine"poll" instruction 4 cycles. However, the optimization of using the LCV pointer for posts that occur to the running frame is hindered by the multiple register contexts as there is a 2 cycle penalty for accessing the thread registers from inlets.

### 5.6 Discussion

Control in fine-grain parallel programs involves the integration of asynchronous events with the control flow internal to the computation. In TAM these two aspects are closely related, with the compiler generating code for both scheduling levels. With this kind of coupling, the compiler can partition the available registers by convention; hardware partitioning into distinct contexts only restricts how the registers can be used, and can prevent certain optimizations. This obervation would not hold, however, if the code running at the different scheduling levels came from unrelated sources, such as the operating system.

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, but one must make a conservative estimate of how many polls to perform. Even when a frame is expecting no messages it must continue to poll the network. For fine-grain parallelism with frequent messages this may be acceptable, but for coarser grained computation the polls would constitute unnecessary overhead.

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

## 6 Heap access cost

The I-structure/M-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. Section 4 examined mechanisms for message exchange and Section 5 considered the dynamic scheduling involved in request completion. 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 or M-structure semantics and permit synchronization on a per-element basis. I-structures support write-once semantics while M-structures allow multiple writes to each element. The discussion in this section focuses on I-structures, but it applies equally

| TL0 operation | Cycles | |
|---|---|---|
| | J′-Machine | CM-5′ |
| I-fetch | | |
| Data present | 9 | 9 |
| Defer | 99 | 41 |
| I-store | | |
| Cell empty | 28 | 15 |
| Deferred readers | 29 | 22 |
| I-structure allocate ($N$ words) | | |
| Local/remote policy | — | 7 |
| Allocate | 41 | 15 |
| Clear tags | $6N$ | $N$ |

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. All data assumes the cached memory model.*

to 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. Reading an element traps if it is not present, and the fault handler implements the synchronization semantics. 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 occurs. When an I-structure is allocated, the tags of each element must be set to empty, which requires one store per element.

On the CM-5, one tag byte is allocated for each 8-byte I-structure element. The tags are stored in a memory area disjoint from the data area. A global register $R_{\text{izero}}$ 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 8 bytes starting at location $R_{\text{izero}} + 8N$. All I-structure accesses must explicitly check the tag byte before reading or writing the data, and deferred reads are handled similarly as on the J-Machine.

### 6.4 Discussion

Table 3 lists 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 there are more than seven I-fetchs per I-store and only 3% of the I-fetches are deferred. On Gamteb, there are only 1.6 I-fetches per I-store and 24% of the I-fetches 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 Section 3 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 apparant 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, but 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.

## 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[18]. 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 at *T[16] 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 reduces 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 was 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. Tagged memory was inconclusive, since some programs yielded a very high trap frequency.

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

- The network interface should be integrated with the register file or cache of a processor. Placing it on the main memory bus is too slow. Few high-performance off-the-shelf processors provide access to these points and a modified processor should be considered for fast network access.
- 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 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.

## References

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th Annual Int'l Symp. on Comp. Arch.*, pages 104–114, Seattle, WA, May 1990.

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

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

[5] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Conf. on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991.

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

[8] 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).

[9] 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).

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

[11] William J. Dally, Roy Davidson, J. A. Stuart Fiske, Greg Fyler, John S. Keen, Richard A. Lethin, Michael Noakes, and Peter R. Nuth. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, April 1992.

[12] 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.

[13] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Technical Report Technical Report 1321, MIT Artificial Intelligence Lab, 545 Tech. Square, Cambridge, MA, 1991.

[14] D. Lenoski, J. Laudon, K Gharachorloo, A Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, Sealttle, Washington, May 1990.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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