Trading Off Overhead and Cache Performance in Fine-Grained Computing

Ellen Spertus* and William J. Dally
ellens@ai.mit.edu, billd@ai.mit.edu
Laboratory for Computer Science and
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

A major challenge in fine-grained computing is achieving locality without excessive scheduling overhead. We built two J-Machine implementations of a fine-grained programming model, the Berkeley Threaded Abstract Machine. One implementation takes an Active Messages approach, maintaining a scheduling hierarchy in software in order to improve data cache performance. Another approach relies on the J-Machine's message queues and fast task switch, lowering the control costs at the expense of data locality. Our analysis measures the costs and benefits of each approach, for a variety of programs and cache configurations. The hardware-buffered implementation is especially strong in direct-mapped caches, where it achieves substantially better instruction cache performance. The Active Messages implementation is strongest when miss penalties are high and for the finest-grained programs.

1 Introduction

Multithreading allows processors to hide the long, unpredictable latencies that occur in dynamically-scheduled fine-grained parallel programs. These latencies are due to both communication and synchronization among parallel computations. Counteracting the benefits of multithreading is the cost of context switching, which includes both direct overhead and the indirect cost of impaired cache performance [MB91]. Two different approaches to lowering the costs of frequent context switches are bringing the programming model closer to the architecture (as is done by Active Messages [vECGS92]) and bringing the architecture closer to the programming model's needs (as is done by the MIT J-Machine [DFK+92]). Ideally, both approaches should be explored in combination.

We compare two implementations of the same programming system, varying the scheduling hierarchy: (1) The Active Messages [vECGS92] implementation, based directly on Culler's Threaded Abstract Machine (TAM) [CSS+91] and von Eicken's Active Messages [vECGS92], manages tasks explicitly with the goal of improving data locality. (2) The *message-driven* implementation flattens TAM's scheduling hierarchy by exploiting the J-Machine's message queue, requiring fewer overhead instructions.

*Author's current address: Microsoft Research, 1 Microsoft Way, Redmond, WA 98052.

We measure the costs and benefits of the two systems, quantitatively evaluating the claim [vECGS92] that using the J-Machine's message queue as a task queue hurts cache performance too much to be useful. While the message-driven approach generally incurs a higher miss rate, its lower number of total memory accesses and substantially reduced overhead more than compensate in many cases, demonstrating that buffered messaging can improve on the Active Messages approach. While we focus on TAM and Active Messages, our work applies to all fine-grained computation occurring on systems with caches by quantifying trade-offs between data locality and control overhead.

1.1 Background

1.1.1 Active Messages

Active Messages is a set of communication mechanisms that improves the performance of large-scale multiprocessors by eliminating the need for buffering incoming messages. This is accomplished by requiring message handlers to be short, restricting them to (1) storing message words (and any control information) into user-allocated memory or (2) replying immediately to a simple request that does not require storage [vECGS92].

The key difference between programming models based on Active Messages and message-driven processing is where the computation is performed. In purely message-driven systems, the computation is performed in the message handlers, i.e., directly in response to incoming messages.

In Active Messages, the computation is performed in the background, and the handlers merely incorporate incoming data into the computation. Because message handlers are short and are executed at higher priority than the computation, hardware message buffering is not needed. The message handlers and operating system work together to schedule tasks, generating larger computations than would occur in a purely message-driven approach. Specifically, tasks using the same context are scheduled to run together, before switching to a different context. Coordinating tasks from a single context creates larger-grained computation, leading to better memory usage (for both caches and registers) and less frequent context-switching [vECGS92, CSS+91].

1.1.2 The MIT J-Machine

The MIT J-Machine, a massively-parallel computer built using Message-Driven Processors (MDPs), was designed to support fine-grained parallel processing. Its mechanisms include two complete priority levels, each with its own register set and large (4 Kbyte) message queue. When a message arrives at the high-priority queue, low-priority computation is preempted. Message reception does not interrupt execution of a same-priority task; dispatch occurs when the task suspends [D+87, DFK+92]. The hardware message buffering has a negligible effect on system cost.†

The J-Machine provides a superset of the base hardware required by Active Messages. Specifically, because of the large message queues with automatic buffering, messages need not be serviced immediately. The designers of Active Messages considered this feature counter-productive, claiming that the relatively short run lengths of a message-driven approach significantly impair data locality by preventing efficient use of registers and cache memory [vECGS92, p. 261]. We evaluate this claim, finding that under some circumstances, a message-driven approach provides a substantial improvement.

1.1.3 TAM

The Berkeley Threaded Abstract Machine (TAM) [CSS+91] is a fine-grained parallel execution model that is used as a back-end for the implicitly-parallel functional language Id [Nik91]. An Id codeblock is compiled into a set of *inlets* and *threads*, each made up of a sequence of instructions. Inlets are short message handlers that receive arguments from outside the codeblock, and threads are sequences of code corresponding to the body of the codeblock. Operations of unbounded latency, such as global reads, are split-phased: the request is initiated in one thread, and the reply is received in an inlet and used in one or more other threads. Each thread has an *entry count* indicating the number of inlets and threads in the same codeblock that must run before it. Inlets initiate threads through the *post* instruction and threads through *fork*.

When a codeblock is invoked, a *frame* is allocated for storage of arguments, local variables, entry counts, and a *remote continuation vector* (RCV), which contains a list of threads that are ready to run. When one of the codeblock's inlets is executed, it typically writes the incoming value(s) into the frame and posts a dependent thread. If the thread is *non-synchronizing* — that is, if it does not have an explicit entry count — a pointer to the thread is placed in the RCV, indicating that the thread may run. If the thread is *synchronizing*, the associated entry count is decremented, and a pointer to the thread is placed in the RCV only if the count has reached zero. A "non-synchronizing" thread has an implicit entry count of one, because it is placed in a continuation vector on the first attempt. When a fork occurs at the end of a thread, it is converted by the compiler into a branch when possible.

When a frame is activated, the frame's continuation vector is then considered the *local continuation vector* (LCV), and a specially-designated entry thread is executed. Any threads that fork other threads within the codeblock do so by placing them in the LCV, or, if the fork appears at the end of the thread, branching directly. Threads are executed from the LCV until none remain, after which the exit thread is executed. The set of threads executed in a single frame activation is called a *quantum*. Typically, activation is comprised of multiple quanta, with a quantum ending when needed data is not yet available.

Entry and exit threads were intended to take advantage of the locality created by multiple threads in a scheduling quantum, allowing time to be saved by having an entry thread load commonly-used frame slots into registers where they would remain during the entire scheduling quantum. At the end of the quantum, the exit thread would store them into frame memory. Entry and exit threads have not been implemented in the TAM compiler, however, and are not included in our analysis. Details on compiling Id programs to run on TAM are available elsewhere [Sch91, TCS92].

2 Two TAM Implementations for the J-Machine

To explore the benefits of Active Messages on a computer with the J-Machine's ability to buffer messages, we implemented two TAM back-ends. TAM is designed to be implemented on top of Active Messages. A separate study examines how TAM's performance is affected by other J-Machine mechanisms, such as tagged memory, automatic dispatch on messages, and tight processor-network coupling [SGS+93]. This section describes the two different J-Machine implementations, which are summarized in Table 1.

2.1 Active Messages Implementation

The Active Messages (AM) implementation of TAM runs short message handlers at high priority and computation at low priority, in the Active Messages style. Threads associated with the same frame are scheduled sequentially. In this implementation, an argument to a codeblock is sent to an inlet, which runs at high priority. The inlet places the argument in the frame and posts a thread. If the

†On the J-Machine, messages are buffered directly into the top level of the memory hierarchy. The processor control finite state machine includes states that transfer messages from the network output register into memory and pointer registers that address the memory during these transfers; these states and registers account for a tiny fraction of the chip area. The major cost in buffering messages is the use of on-chip SRAM space and bandwidth by arriving messages; however, even under software control, cache space and memory bandwidth is required to buffer most arriving data. If the network interface is attached to a lower level of the memory hierarchy, as with the CM-5 [Thi92], even more scarce memory bandwidth is consumed reading incoming message data out of the memory-mapped registers. Thus, the only advantage to not buffering messages is avoiding consuming cache space and bandwidth for any incoming message data that can be handled without being written to memory.

TAM Mechanism	AM Implementation	MD Implementation
inlet	high priority message handler	low priority message handler
post from inlet	place thread in RCV	jump directly to thread
activation of frame	low priority library routine	n/a
threads	low priority code	low priority code
fork from thread	jump or push onto LCV	jump or push onto LCV
system routines	high priority message handlers	high priority message handlers

Table 1: Mapping of TAM Constructs to the J-Machine

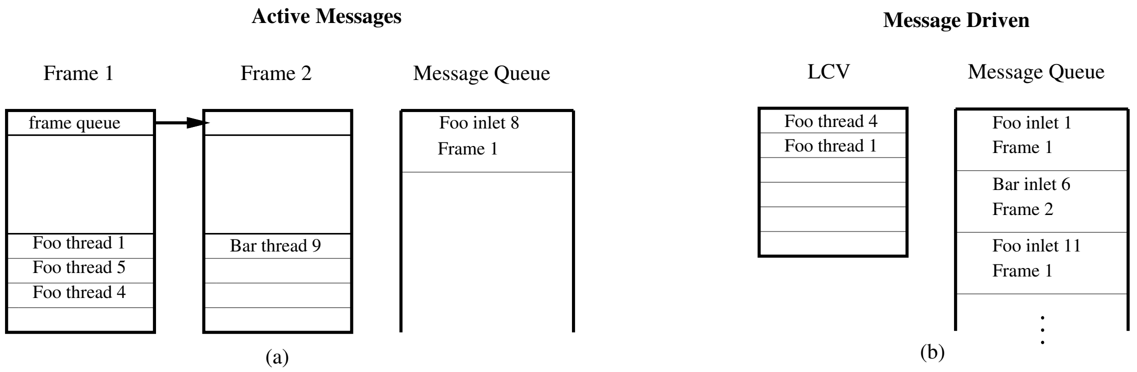

Figure 1: A comparison of the scheduling hierarchies for the two implementations. In the AM implementation (a), messages in the queue are executed at the highest priority, so the inlet would run next, followed by all the enabled threads corresponding to Frame 1, after which Frame 2 would be come active. In the MD implementation (b), inlets are executed at the same priority as threads, so none of the inlets would be executed until the LCV is emptied. Then, each inlet would be processed, along with any thread it posts (and the thread's descendants).

thread is now ready to run, a pointer to the thread is placed in the RCV. When the processor is otherwise idle, a background process chooses a frame whose RCV is non-empty, sets the LCV pointer to the RCV, and branches to its entry thread. When a thread finishes, the address of the next thread is popped from the LCV. When one thread forks another, the child thread's entry count is decremented. If the entry count has reached zero or if the thread is non-synchronizing, a pointer to the thread is pushed on the LCV. The last item in the LCV is the address of the system code to swap in a new frame.

An atomicity problem arises due to allowing inlets to interrupt threads [Spe92]. Specifically, consider the case where the TAM "fork t" instruction is interrupted by an inlet that includes the instruction "post t". If the thread and the inlet belong to the same codeblock activation, the synchronization counter for thread t may be improperly decremented. Similarly, the LCV and the pointer to the top of the LCV are places of contention. In order to ensure atomic access of data structures accessible from both threads and inlets, interrupts are disabled during control operations in thread bodies. In Section 3.2, we discuss the effects of different policies toward enabling interrupts.

2.2 Message-Driven Implementation

The Message-Driven (MD) implementation uses the message queues as task queues instead of servicing messages immediately. Inlets contain branches directly to threads, eliminating the need for the RCV. Because control can be transferred directly from an inlet to a thread, both run at low priority. The only code that runs at high priority is that to service system calls, such as allocating frames or accessing global data structures. As in the AM implementation, if a thread forks multiple threads, control is transferred to one and the others are pushed on the LCV. When a thread that does not end with a fork completes, the next thread address is popped from the LCV.

Unlike in the AM implementation, atomicity is not a problem because inlets run at the same priority as threads and do not interrupt them.

The MD implementation's partial scheduling hierarchy ensures that all threads in the LCV are executed before switching frames. While one could totally eliminate scheduling hierarchy by placing threads in the message queue instead of in a continuation vector, there would be no purpose in doing so, since this would sacrifice data and control locality without improving overhead.

2.3 Comparison of Implementations

The MDP mechanisms for the two implementations are summarized in Table 1. The order of thread and inlet execution for a sample message sequence is contrasted in Figure 1. In the AM implementation, shown in subfigure (a), each frame has a list of threads in its RCV. These are run through entirely for one frame (along with any threads they spawn) and then for another. Arrivals in the message queue are processed immediately, interrupting thread execution. In the MD implementation, shown in subfigure (b), there is only one continuation vector, for the currently-running frame. The messages in the queue are not processed until the LCV has been emptied. Then, the first message is processed, executing an inlet that pushes any threads it posts on the LCV, which will then get cleared out before the next message is processed.

While both implementations yield the same results, their dynamic behaviors differ. If two inlet messages for the same frame arrive at about the same time, under the AM implementation, one will run, then the other, followed by any threads they fork. Under the MD implementation, the first inlet will run, followed by any threads that it posts, with the second inlet running after all the enabled threads within the frame complete.

The MD implementation has four negative consequences:

1. Since inlets are not executed at high priority, the message queue has a greater likelihood of overflowing.

2. Due to the shorter quanta, entry and exit threads (if used) would run more often, and more quanta would be started.

3. The shorter quanta prevent the data cache benefits of grouping together threads from the same frame.

We do not address the first concern in this paper, only running programs that fit in the message queue. We verified that substantial problems could be solved without using all the memory available for message queues. We also disregard the second issue, since entry and exit threads have not been and are not planned to be implemented. Also, starting a quantum in the MD approach is cheap, requiring no more time than a thread-to-thread control transfer. We focus on the third point: how the cache differences and instruction counts of the two systems affect total run times.

The major benefit of the MD implementation is reduced time to post from an inlet, enqueue a frame into the global list, and begin an activation. Nikhil made similar observations independently, criticizing TAM for its atomicity problems, frame memory traffic, and scheduling overhead, which he found to be of greater significance than the improvement in locality [Nik93].

An additional benefit of the MD implementation is that, because inlets pass control directly to threads instead of placing them into a continuation vector, a bigger region of code is open to conventional optimization. For example, consider the following inlet and thread, as they might be implemented on the MDP:

```
     Inlet 0:
(I1)  reg0 ← message.argument
(I2)  frame.5 ← reg0
(I3)  post thread 1
     Thread 1:
(T1)  reg0 ← frame.5
      ...
(Tn)  stop
```

If thread 1 is non-synchronizing and if only inlet 0 posts or forks thread 1, the reload of the register in line T1 can be eliminated. Additionally, the code for the thread can be placed immediately after the inlet, eliminating the need for line I3. If no other threads use frame slot 5, line I2 can be removed. Ordinarily, the stop statement in line Tn is implemented as a pop of the LCV into the instruction register. If thread 1 contains no pushes onto the LCV, then the LCV is known to be empty, and the stop can be converted to a suspend instruction. Even if only some of these conditions are met, a subset of these optimizations can be performed.

It is possible to combine the two approaches, using the message-driven approach for short threads and the Active Messages approach for long threads, as is done with Optimistic Active Messages [KWW+94]. In this study, however, our goal is to understand the differences in behavior of the two pure systems.

3 Analysis

In this section, we compare the differences in the dynamic behavior of the code produced by the two implementations, including relative numbers of data and instruction accesses and cache performance. Although several 512-node J-Machines are operational, an instruction simulator was used to produce more detailed statistics, specifically on memory access and granularity. Our systems can run on multiple processors, but we examine uniprocessor results almost exclusively, factoring out one uniprocessor anomaly to better predict multiprocessor performance (Section 3.2).

The programs used in the analysis and theiar arguments are: *matrix multiply* (mmt) 50, which multiplies two matrices of floating-point numbers and sums the elements of the product; *quicksort* (qs) 100, which sorts an array of random integers; *discrete time warp* (dtw) 10, a speech-processing application that performs operations on matrices of floating-point numbers; *paraffins* 13, which enumerates the distinct isomers of paraffins [AHN88]; *wavefront* 40, which computes successive matrices in which each element depends on a function of north and west values of the previous and current matrix; and *selection sort* (ss) 100, which sorts an array of integers that are originally in reverse order. The number of instructions executed ranged from hundreds of thousands to tens of millions of machine instructions.

3.1 Counts of instructions and memory accesses

The primary data access benefits of the MD implementation are:

- eliminating the remote continuation vector and
- directly accessing data from the message queue instead of storing it in the frame.

In the executions of paraffins and quicksort, the MD implementation contains more reads and writes of system memory than the AM implementation because the CV is in system rather than user memory. The total number of data accesses is still smaller for the MD implementation.

The main instruction count benefits of the MD implementation are:

- eliminating the calls to library routines to post threads and manage the queue of inactive frames,
- eliminating data stores/loads due to the control locality of inlet and thread execution, and
- eliminating pops of the empty CV.

For analysis, memory was divided into system and user regions. System code includes the operating system and library, including the floating-point library. System data structures are comprised of the incoming message queues, operating system globals, and the LCV. User code consists of the threads and inlets unique to each program. On average, the MD implementation yields 86% of the reads, 87% of the writes, and 77% of the fetches produced by the AM implementation.

In the remainder of this section, we quantitatively expresses the differences in dynamic behavior in the code produced by the two implementations. We now take into account the differences in cache performance, showing the net effect of the trade-offs in instruction counts and locality.

3.2 Measures of Granularity

A useful metric of granularity is *threads per quantum* [CSS+91], which indicates how many threads from a given codeblock are executed before the LCV is emptied. Consider the case where a quantum using frame f is completed and is immediately followed by the arrival of an inlet that posts to frame f. In this case, one quantum involving f can be immediately followed by another. We define a *superquantum* to be composed of adjacent quanta acting on the same frame. The average number of threads per superquantum for a program will always be greater than or equal to the average number of threads per quantum. We contend that the superquantum is more useful than the quantum as a measure of locality, in the absence of entry and exit threads, since it is more directly related to cache performance.

Table 2 shows the average number of threads and instructions per quantum and superquantum for all the programs running on a single processor for each system. As expected, the AM implementation has higher numbers of instructions and threads per quantum or superquantum, almost without exception. It was surprising how much larger superquanta were than quanta for wavefront and selection-sort, the latter in part because it makes only 3 procedure calls in its entire execution, leading to high locality for frame memory. With the exception of matrix multiply (MMT), whose average number of instructions per thread (IPT) is much larger than any other program, instructions per superquantum (IPS) grows as one moves downward in the table. We will return to these metrics in Section 3.3, which discusses cache performance.

We discovered an interesting uniprocessor anomaly when analyzing the large difference between the sizes of quanta and superquanta for certain programs. Our AM implementation only briefly enables interrupts at the top of each thread. Another possibility, which we call the *enabled* implementation, allows interrupts whenever possible (i.e., except during CV access). As Figure 2 shows, the enabled implementation allows a local I-structure fetch to be serviced immediately, resulting in greater quantum size. In the AM implementation, the I-structure fetch might not be serviced until after the quantum, decreasing granularity. The Berkeley team inlined fetches of local I-structures, so this effect was not a direct concern. While performance of the enabled implementation is superior to that of the AM implementation on a single processor, the difference between these two systems is negligible when I-structure accesses are remote. The unenabled AM implementation on a uniprocessor, which we use, more accurately models multiprocessor behavior of either of these systems than does the enabled implementation on a uniprocessor.

3.3 Performance including cache effects

We fed instructions and traces generated by the instruction simulator into a cache simulator. In all cases, we specified separate instruction and write-back data caches with replacement of the least-recently-used element. We simulated 1-, 2-, and 4-way set-associativity with block sizes varying from 8 to 64 bytes. We show data for 64-byte blocks, the size at which both systems performed best. Instructions were assumed to uniformly take one cycle, not counting memory access time. Because the number of data and code accesses differ between the two implementations, the absolute numbers of cycles, not miss percentages, are compared. Throughout this section, we use as a metric the ratio of the number of total cycles (includ-

Program	TPQ		TPS		IPT		IPQ		IPS	
	MD	AM	MD	AM	MD	AM	MD	AM	MD	AM
MMT	3.6	3.6	4.2	4.2	84	90	302	326	349	373
QS	3.8	5.7	4.5	5.7	16.7	20.8	63	119	76	119
DTW	4.4	6.0	5.3	6.0	22.6	26.5	100	159	119	159
Paraffins	4.1	8.6	6.8	8.7	23.8	29.2	96	250	161	253
Wavefront	3.1	3.3	43.9	65.2	50.7	54.0	157	176	2223	3520
SS	4.7	4.8	6432	6892	13.1	18.3	62	87	83961	126104

Table 2: A comparison of threads per quantum (TPQ) and superquantum (TPS), and instructions per thread (IPT), quantum (IPQ), and superquantum (IPS) for the Message-Driven (MD) and Active Messages (AM) implementations. The programs are organized so that TPS increases.

```
Thread A1                     Thread B1
(A1) Enable interrupts        (B1) Enable interrupts
(A2) Disable interrupts       (B2) Initiate I-fetch
(A3) Initiate I-fetch         (B3) Disable interrupts
(A4) Pop CV                   (B4) Pop CV
          (a)                              (b)
```

Figure 2: In subfigure (a), illustrating the (unenabled) AM implementation, interrupts are enabled briefly at the top of a thread. The I-fetch is not serviced during the lifetime of the thread. In subfigure (b), illustrating the enabled AM implementation, interrupts are only disabled for CV access. If the I-structure fetch is local, it will occur between steps B2 and B3 in the enabled implementation, followed by the inlet to receive the data, allowing another thread to be placed on the CV, extending the quantum.

Program	TPQ		TPS		MD/AM		
	MD	AM	MD	AM	12	24	48
mmt	3.60	3.62	4.16	4.15	1.03	1.20	1.54
qs	3.79	5.73	4.53	5.73	.98	1.13	1.38
dtw	4.44	6.00	5.28	6.00	.97	1.12	1.39
paraffins	4.05	8.55	6.79	8.65	.87	.92	.99
wavefront	3.09	3.27	43.87	65.22	.87	.86	.87
ss	4.75	4.75	6432	6892	.61	.61	.62

Table 3: A comparison of threads per quantum and superquantum for the two implementations to the ratios of the cycles taken under the MD and AM implementations in 8192-byte 4-way set-associative caches with varying miss costs. Programs are organized so that threads per superquantum increases and the cycle ratios generally decrease.

ing miss penalties) of the MD implementation to that of the AM implementation.

Figure 3 shows the ratios of the geometric means of the test programs, with varying cache sizes and miss penalties. For all caches, the MD implementation outperforms the AM implementation when the miss cost is 12 or 24 cycles or the cache is large. (This is true of the geometric mean but not of all the test programs, as will be discussed later.) When the miss cost is 48 cycles, the MD implementation outperforms the AM implementation if a direct-mapped cache is used, and there is no clear winner with 2- and 4-way set-associative caches.

There is little difference between the ratios of the geometric means for 2- and 4-way set-associative caches, but there is for direct-mapped caches. In the next sections, we will discuss the behavior with 4-way set-associative and direct-mapped caches.

3.3.1 4-way set-associative caches

The ratio of total cycles used in the MD to the AM implementation is lowest with small and large caches, as was the case in Figure 3. With small caches, both systems have high miss rates, so the number of memory accesses is the primary factor. With large caches, there are few misses, so after compulsory misses are taken, the number of memory accesses dominates. As the TAM model predicts, the two-level scheduling strategy yields fewer data cache misses on medium-sized caches. This is an important cache size range to consider, since relative cache performance is less important at the extremes, when the working set fits either completely or not at all in the cache.

While most programs run more quickly in the AM than the MD implementation for the largest miss penalty considered, three of the programs still perform best under the MD implementation. Conversely, for the smallest miss penalty, the AM implementation outperforms the MD implementation for one program and does about as well on two others. This behavior can be understood by examining the granularities of each of the programs. Table 3 shows the relation between threads and instructions per superquantum (Table 2) and the cycle ratio. As threads per superquantum increases, the cycle ratio decreases; i.e., the more work in each

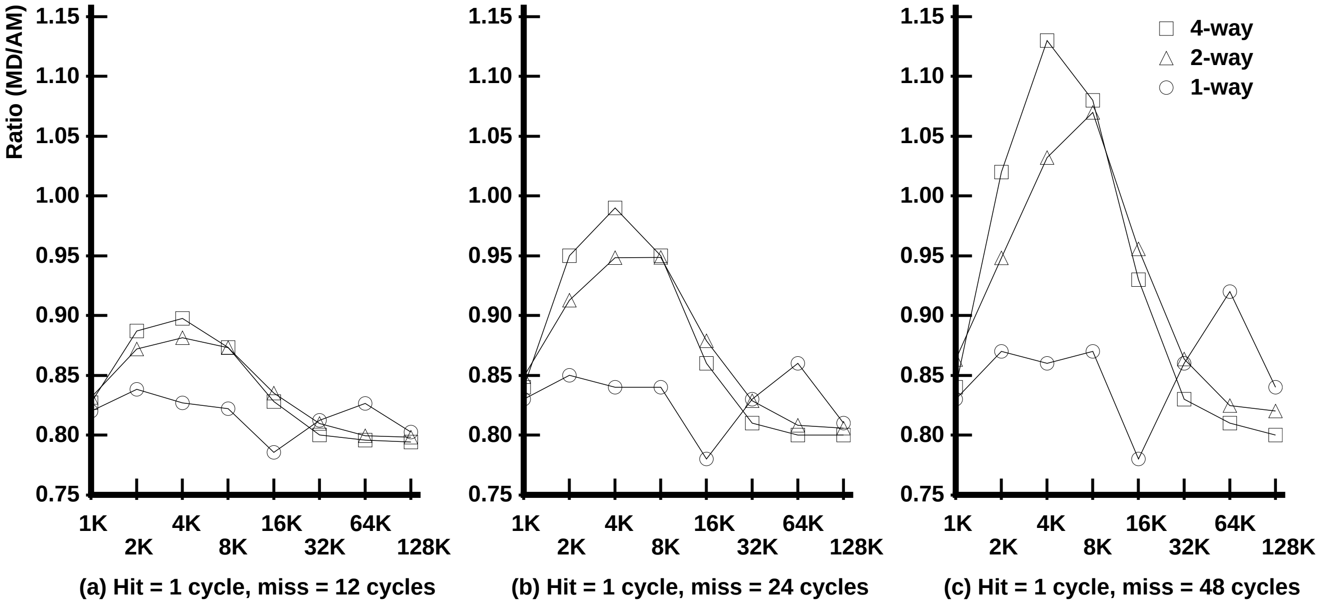

Figure 3: The geometric means of the ratios of the total time taken for the set of test programs in the MD to the AM implementation given separate data and instruction caches. The miss penalties are 12, 24, and 48, respectively, in the three graphs.

superquantum, the better the MD implementation's relative performance. Thus, the MD implementation is better when locality is easier to find, and the AM implementation is better on finer-grained programs where there is little locality to exploit. Superquanta were more useful than quanta (Table 2) in quantifying granularity.

3.3.2 Direct-mapped caches

While 4-way set-associative caches yielded the best performance for both systems, we also examined direct-mapped caches because they are easier to build and have lower latency. Figure 5 shows the relative performance of each program and their geometric mean for miss costs of 12, 24, and 48 cycles. The order of the ratios is again closely related to the relative threads per superquantum. These graphs exhibit more "noise" due to the conflicts characteristic of direct-mapped caches, so the geometric mean shows more regularity than any of the individual programs. While the curves representing the geometric mean average out the noise and are qualitatively similar among the three graphs, the shape is different from what we saw with 4-way set-associative caches (Figure 4). The reason for the dip in the 8–16 K range, indicating superior MD implementation performance, is relatively poor instruction cache performance for the AM implementation, a predictable consequence of the lesser control locality.

As Subfigure 5a shows, when the miss cost is 12 cycles, all programs do better in the MD than the AM implementation. When the miss cost increases to 24 cycles, as in subfigure (b), matrix multiply and discrete time warp perform better in the AM than MD implementation for some cache sizes. At 48 cycles, subfigure (c), the benefits of the AM implementation for matrix multiply, discrete time warp, and quicksort are pronounced, but the geometric mean indicates overall superior performance in the MD implementation. To remove the effects of the outlier, selection sort, Figure 6 shows the geometric means of all the programs except for it. The MD implementation still performs better than the AM implementation for miss costs of 12 and 24 cycles, although less dramatically so. With a miss cost of 48 cycles, the geometric mean for the AM implementation is sometimes slightly superior.

3.4 Summary

The cache benefits of TAM's two-level scheduling hierarchy have been overstated. For the finest-grained test programs, the two-level scheduling hierarchy of the AM implementation had superior performance for medium-sized caches and for high miss penalties. The MD implementation performed well at all cache configurations for less finely-grained programs. Threads per superquantum proved a better measure of program granularity than did threads per quantum.

4 Conclusions

We have performed the first quantitative study of the cache benefits of TAM's two-level scheduling hierarchy and Active Messages' decision to use short message handlers rather than message-driven

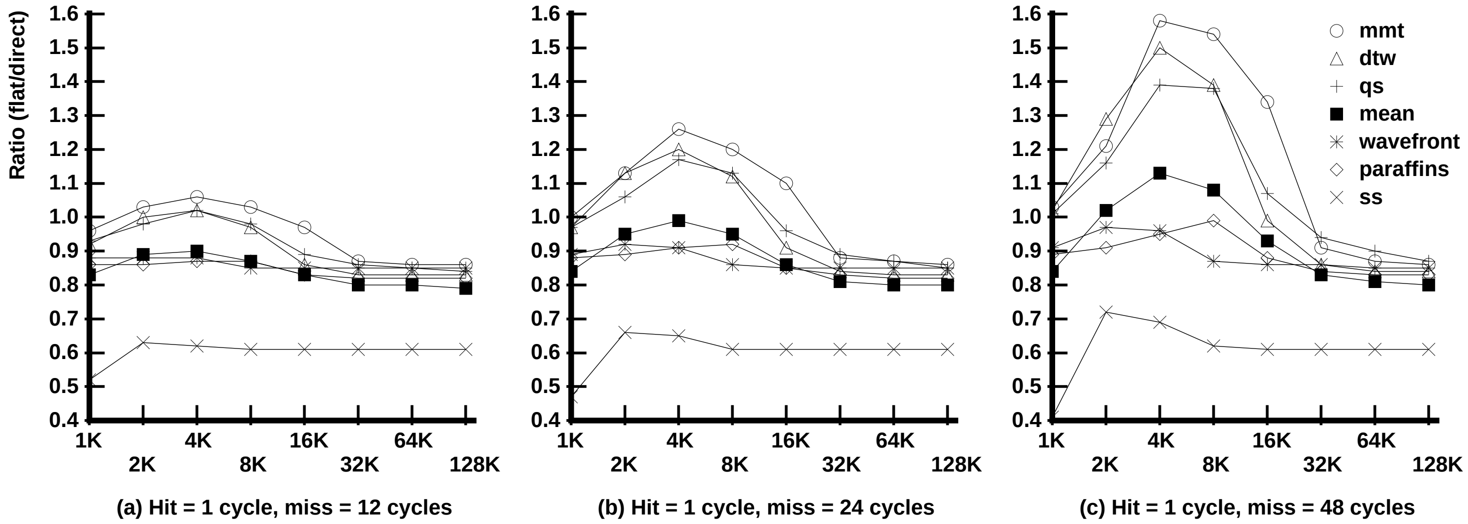

Figure 4: This figure shows the ratio of the total cycles taken in the flat implementation vs. the direct implementation for separate **4-way set-associative** data and instruction caches of varying sizes. The miss penalties are 12, 24, and 48 cycles in the three graphs, respectively. The curve with the filled-in squares in each graph indicates the geometric mean.

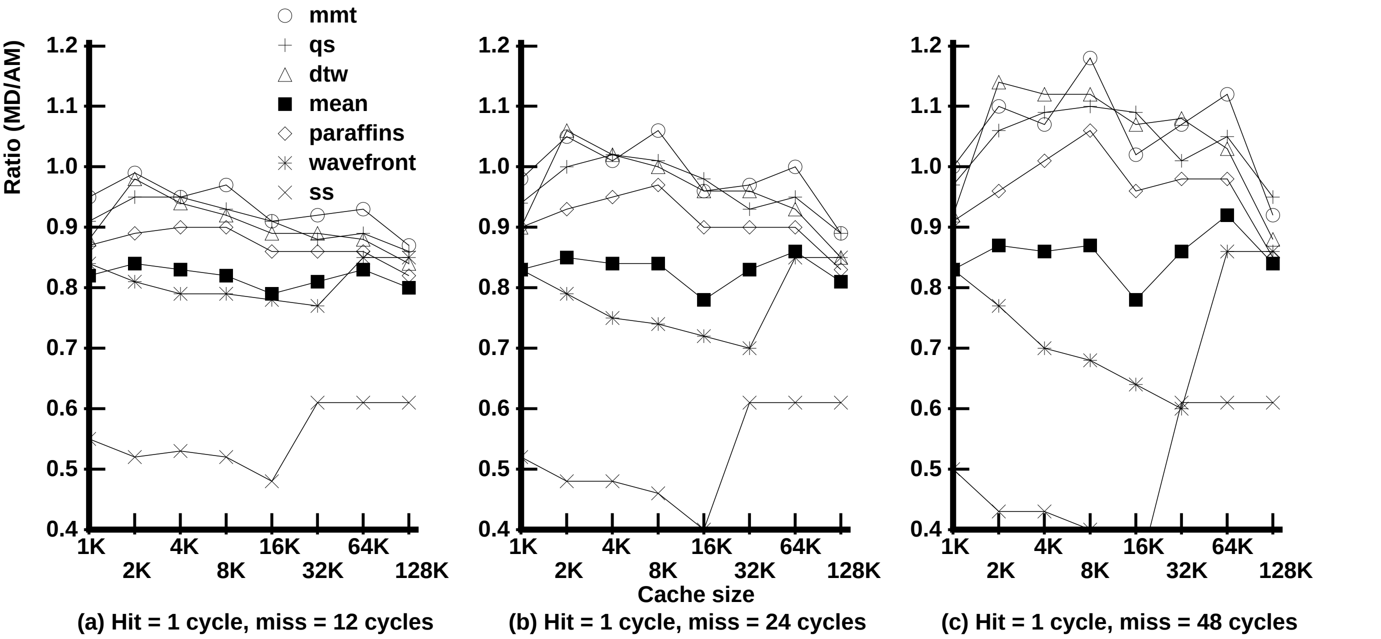

Figure 5: This figure shows the ratio of the total cycles taken in the MD implementation vs. the AM implementation for separate **direct-mapped** data and instruction caches of varying sizes. The miss penalty is 12, 24, and 48 cycles in the three graphs, respectively. The curve with the filled-in squares in each graph indicates the geometric mean.

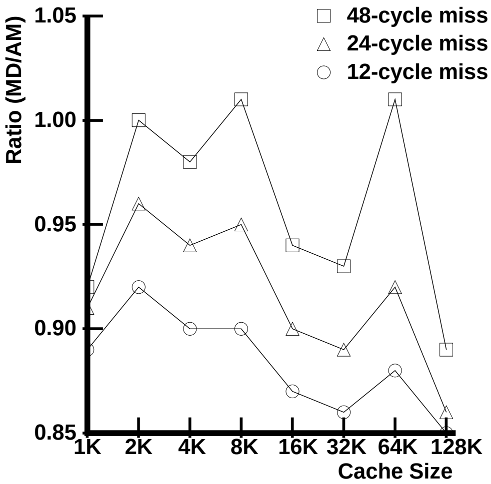

Figure 6: The geometric means of the ratio (MD/AM) of the total cycles taken in all programs except selection-sort for direct-mapped caches.

computation. The significance of this work extends to all fine-grained computation occurring on systems with caches. Specifically, we have measured in which cases the locality benefits of Active Messages outweigh their additional control cost. The message-driven implementation is superior not only for caches so large or so small that the absolute number of accesses dominates but when the MD implementation has higher miss rates but significantly lower total accesses. The MD implementation is especially strong in direct-mapped caches, where its control locality leads to better instruction cache performance. The AM implementation is strongest when miss penalties are high and for the finest-grained programs. While our results were only for uniprocessors, our isolation of a uniprocessor anomaly (Section 3.2) gives reason to believe our work would extend to multiple processors, although further research needs to be done.

We have extended the threads per quantum metric to threads per superquantum and found the latter a more useful measure of granularity. Not only does it more accurately describe how long a data set remains in use, but it proved a good predictor of the relative benefits of exploiting control and data locality. It also helped us pinpoint the granularity differences resulting from different interrupt enabling policies.

More generally, we have shown that the J-Machine's buffered message queues, an inexpensive mechanism, can provide a real benefit over the Active Messages approach. The benefit of hardware dispatch has been shown elsewhere [SGS+93]. The net lesson is that hardware mechanisms for fine-grained parallelism can significantly improve performance. Additionally, it is beneficial for the message queue to be available so new compilation techniques can be explored and implemented without requiring a new machine.

Our results depend heavily on cache miss penalties and cache sizes. In the future, we can expect both the miss penalty and size of caches to grow. While higher penalties cause data locality to increase in importance, larger caches with their lower miss rates allow taking advantage of control locality. Thus, neither of the systems would be clearly superior on future caches, and the same trade-offs remain.

Our results also depend on program granularity. Id/TAM programs are very finely-grained, with threads containing tens of instructions and quanta a few hundred. From the better performance of the message-driven approach on the coarser-grained benchmark programs, we can conclude that its relative performance will be even stronger for programming systems with coarser granularity, such as [REFS].

Acknowledgments

We are grateful to David Culler, Seth Goldstein, Klaus Schauser, and Thorsten von Eicken for sharing their systems and expertise. We are also grateful to the MIT Computation Structures Group and Concurrent VLSI Architecture group for the compiler and simulator infrastructure they provided. We received valuable comments on this work and its presentation from the above people and Fred Chong, Michael Ernst, Bob Gruber, Rishiyur Nikhil, Nate Osgood, Bjarne Steensgaard, Debby Wallach, and Daniel Weise.

This work was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and by a National Science Foundation Graduate Fellowship. This paper was written while the first author was employed by Microsoft Research.

References

[AHN88] Arvind, S. K. Heller, and R. S. Nikhil. Programming generality and parallel computers. In *Proceedings of the* 4^{th} *International Symposium on Biological and Artificial Intelligence Systems*, pages 255–286, Trento, Italy, September 1988. ESCOM (Leider).

[CSS+91] 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).

[D+87] William J. Dally et al. Architecture of a message-driven processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, June 1987.

[DFK+92] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[KWW+94] M. Frans Kaashoek, William E. Weihl, Deborah A. Wallach, Wilson C. Hsieh, and Kirk L. Johnson. Optimistic active messages: Structuring systems for high-performance communication. In *Sixth SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991. DEC.

[Nik91] R. S. Nikhil. Id (version 90.1) reference manual. CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1991.

[Nik93] Rishiyur S. Nikhil. A multithreaded implementation of id using p-risc graphs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, 1993.

[Sch91] Klaus Erik Schauser. Compiling dataflow into threads. Master's thesis, Computer Science Division, University of California at Berkeley, 1991.

[SGS+93] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the j-machine and the cm-5. In *Proceedings of the International Symposium on Computer Architecture*, pages 302–313, May 1993.

[Spe92] 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 Technical Report 1380.

[TCS92] K. R. Traub, D. E. Culler, and K. E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. *ACM LISP Pointers*, 5(1):324–334, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming.

[Thi92] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, January 1992.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.