When does Dedicated Protocol Processing Make Sense?

Babak Falsafi and David A. Wood

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
wwt@cs.wisc.edu

Abstract

Distributed-memory parallel computers and networks of workstations (NOWs) both rely on efficient communication over increasingly high-speed networks. Software communication protocols—from flow-control and reliable delivery to multicasting and coherent distributed shared memory—are often the performance bottleneck. Several current and proposed parallel systems—e.g., the Intel Paragon—address this problem by dedicating one general-purpose processor (in a multiprocessor node) specifically for protocol processing. This operating system convention reduces communication latency and increases effective bandwidth, but also reduces the peak performance since the dedicated processor no longer performs ‘‘useful’’ computation.

In this paper, we study a network of multiprocessor workstations and ask the question: ‘‘when does it make sense to dedicate a processor specifically for protocol processing?” We compare three protocol processing policies: *Single*, the baseline case with one processor that does everything; *Fixed*, which uses a dedicated protocol processor; and *Floating*, where all processors perform both computation and protocol processing.

We use a simple analytic model of a general request/reply protocol to illustrate the trade-offs between the policies. The model shows that: i) adding a dedicated protocol processor to a uniprocessor node is unlikely to be cost-effective and even less likely to outperform the Floating policy; ii) a dedicated processor is more advantageous for light-weight protocols (e.g., active messages) than for heavy-weight protocols (e.g., TCP/IP), iii) the Fixed policy becomes advantageous when communication becomes the bottleneck, as when multiple compute processors and multithreading saturate the resource. The break-even point between Fixed and Floating is a function of the number of processors, protocol overheads, and application parallelism.

We then evaluate these policies in the context of a fine-grain user-level distributed shared memory system. We present preliminary measurements from a dedicated network of Sun SparcStation-20s connected by a Myrinet network. The measured performance on four nodes—each with up to four processors—of three hybrid shared-memory parallel applications confirm the intuitive results from the model.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 Introduction

Distributed-memory parallel computers have become the supercomputers of the 1990s, providing impressive performance on many large and important applications. Networks of workstations (NOWs) promise to exploit economies-of-scale to make large-scale parallel computing cost-effective enough for every day use [4]. Both types of systems rely heavily on efficient communication over high-speed networks. While the underlying network hardware keeps improving rapidly, the overhead of software communications protocols—ranging from flow-control and reliable delivery to multicasting and coherent distributed shared memory—has increasingly become a bottleneck [18].

To address this problem, many distributed-memory parallel machines employ a dedicated protocol processor to off-load the primary (computation) processor(s). For example, the Meiko CS-2 [17], IBM SP-2 [8], and proposed Stanford FLASH [15] all use embedded processors to accelerate communications performance. By reducing the frequency of system calls, interrupts, locking, and cache pollution, these processors reduce communication latency and increase effective bandwidth.

A variation on this approach exploits the growing availability of bus-based shared-memory multiprocessors. The Intel Paragon [13], MIT StarT-NG [5], and Wisconsin T-Zero [21] systems all dedicate one processor of a multiprocessor node—by operating system convention—specifically for protocol processing.

Unfortunately, while a dedicated protocol processor can improve communications performance, it provides little benefit for compute-bound programs. These applications would rather use the dedicated processor for computation. In a recent experiment, researchers at Sandia demonstrated that using the Paragon's protocol processor for computation (via a low-level cross-call mechanism under SUNMOS) nearly doubled the performance of Linpack [14].

In this paper, we study a network of multiprocessor workstations being used as a parallel machine and ask the question: *"when does it make sense to dedicate one processor in each node specifically for protocol processing?"* As with previous multiprocessor thread-scheduling studies [28,29,25], the central question is when do the overheads encountered in practice outweigh the theoretical advantages of processor sharing. In this study, we examine three scheduling policies for protocol processing:

- *Single*, the baseline case where one processor performs all communication and computation,
- *Fixed*, where one processor in a multiprocessor node is dedicated for protocol processing, and
- *Floating*, where all processors perform computation and alternate acting as protocol processor.

We present a simple model—using mean occupancies and response times—for a general request/reply protocol that shows that:

1. Adding a dedicated protocol processor to a uniprocessor node is unlikely to cost-effectively improve performance over Single, and is even less likely to perform better than Floating,
2. The Fixed policy benefits light-weight protocols (e.g., active messages) more than heavy-weight protocols (e.g., TCP/IP).

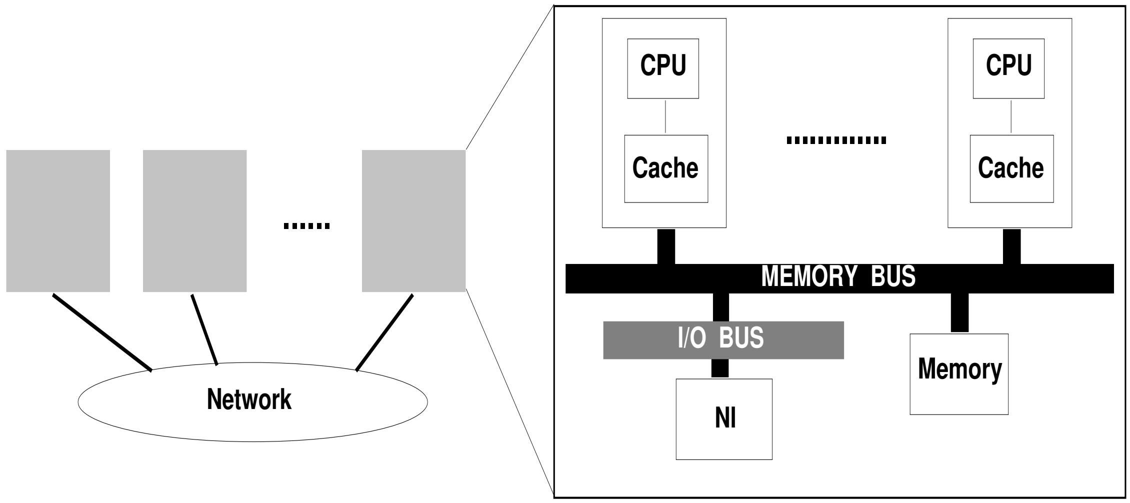

FIGURE 1. Network of Multiprocessor Workstations

3. The Fixed policy performs better than Floating when communication becomes the bottleneck, which is more likely to happen with multiprocessor nodes and/or multithreading. The breakeven point is a function of the number of processors, protocol overheads, and application parallelism.

We then present preliminary measurements from an implementation on four nodes of a network of Sun SparcStation-20s connected by a Myrinet network[1] running the light-weight Illinois Fast Message protocol [20]. We vary the number of processors per node from one to four as well as the number of threads per processor. We examine the performance of three hybrid shared-memory applications [11] running on a fine-grain distributed shared memory system [23].

Measurements confirm the breakeven point predicted by the analytic model. Fixed performs better than Float for communication-intensive codes that saturate the protocol processor. Fixed also performs better for computation-intensive codes once the node's internal bus nears saturation; Fixed reduces bus contention because the protocol processing state remains resident in the dedicated processor's cache.

The next section summarizes the system architecture assumed in the remainder of the paper. Section 3 then describes the three protocol processing alternatives in more detail. Section 4 presents our simple analytic model to present the intuition behind the policy performance. Section 5 describes the relevant details of our distributed shared memory implementation and Section 6 presents preliminary measurement results. Finally, Section 7 concludes the paper.

2 System Architecture

Figure 1 illustrates the general class of parallel machines—a dedicated network of multiprocessor workstations—that we study in this paper. Each node of this system is a commodity multiprocessor workstation, consisting of one to four processors and memory connected by a bus. A snooping

1. Note to reviewers: We expect to have results on at least 8 nodes (and possibly as many as 32 nodes) for the final version of this paper.

cache-coherence protocol keeps the caches *within* a node consistent. A network interface device connects the node to a low-latency, high-bandwidth network; this device may either interface via the node's standard I/O bus or directly to the high-speed memory or graphics bus. Section 5 provides more details of our current implementation.

Each node runs a standard commodity operating system but also runs higher-level software that manages them collectively as a single parallel machine [3,10]. Parallel applications are invoked by starting a single process on each of the nodes (i.e., the SPMD programming model); multiprocessing within a node is handled locally. In this paper, we assume that space sharing—were the nodes are logically allocated to separate parallel tasks—is the only form of sharing. More general time sharing is of course possible, but is beyond the scope of this paper.

A distributed shared memory system extends the coherent shared memory abstraction beyond the processors in a single node. This study assumes a fine-grain distributed shared memory system based on the Tempest interface [22]. This system allocates shared memory at the page granularity, like many other DSM systems, but maintains coherence at a finer grain (e.g., a cache block). Coherence is enforced via a fine-grain access control mechanism analogous to the ubiquitous page-level protection mechanism. While the results of this paper are largely independent of whether this mechanism is implemented in hardware or software, the results in Section 6 come from a software technique [23].

High-performance communication is performed via an active message abstraction [30]. Active messages are essentially very light-weight RPCs that are optimized for the case where processing nodes are co-scheduled; that is, where the destination node already has the correct context for the RPC. Message arrivals either cause interrupts or the processor(s) may poll the network interface to eliminate the interrupt overhead. We assume Tempest active message semantics, which reduces the need for synchronization by requiring sequential execution of handlers within a node.

Tempest is a user-level interface, so the distributed shared memory protocol actions run in user-level software. Active message arrivals, fine-grain access control violations (also called block access faults), and page faults are all vectored to their appropriate user-level handlers.

3 Protocol Processing Policies

In general, protocol processing consists of running the user and system software needed to manage the communication between cooperating nodes. In this study, we focus on communication in parallel applications using the Tempest primitives, either directly or through a user-level distributed shared memory system. Protocol processing includes both user-level block access fault and message handlers plus the low-level messaging protocols that ensure reliable message delivery. Tempest handlers are defined to be atomic and mutually exclusive, i.e. each node may have at most one handler running at a time. This effectively limits each node to having a single processor executing protocol events; regardless of the policy we say that such a processor is *acting* as protocol processor.

In the remainder of this section, we briefly describe each of the three protocol processing policies: Single, Fixed, and Floating

3.1 Single Processor

Single is the baseline policy that applies only to a single processor per node. Under this policy, the single processor performs all protocol processing as well as all computation. Implementing active messages under this policy either requires interrupts or periodic polling in the compute thread. Otherwise, the active message response time could be unbounded, severely degrading application performance. Unfortunately, delivering user-level interrupts is slow in most current operating systems; Thekkath and Levy showed that a simple exception (which requires a similar path through the kernel) takes at least 60 to 200 microseconds for a round-trip [26]. The alternative is periodic polling, which requires instrumenting the computation thread to periodically check for messages. This can be done either via a compiler [30] or by directly editing the executable file [16]. This approach requires a trade-off between latency and overhead: frequent polls decrease message latency but increase overhead.

3.2 Fixed Protocol Processor

The Fixed policy dedicates one processor of a multiprocessor node to perform only protocol processing. By always polling the network when otherwise idle, the protocol processor eliminates the need for message interrupts or polling by the compute processor(s). In addition to decreasing the total overhead, the Fixed policy invokes message handlers more quickly, reducing the round-trip latency and protocol processor occupancy. Lower occupancy allows the dedicated protocol processor to sustain a higher bandwidth of protocol requests than under the Single policy. Secondary factors also help improve performance. The protocol processor's caches are not polluted by compute threads, and should thus have lower miss ratios [18,19].

3.3 Floating Protocol Processor

The disadvantage of dedicating a protocol processor is that may waste cycles that could have productively contributed to the computation. The Floating policy addresses this dilemma by using all processors to perform computation; however, when one becomes idle (e.g., due to waiting for a remote request or synchronization operation) it assumes the role of protocol processor. Since all processors may be computing, either interrupts or periodic polling is still required to ensure timely active message handling. On the other hand, once a processor assumes the role of protocol processor, handler dispatch is nearly as efficient as in Fixed. Thus when communication is infrequent, Floating can use all processors for computation. When communication becomes more frequent, it degenerates to Fixed.

4 A Simple Analytic Model

In this section we develop a simple analytic model to provide intuition about the trade-offs between the three protocol processing policies. We use the model to address two questions about a parallel computing on a network of workstations:

- When is it cost-effective to add a second processor to serve as a dedicated protocol processor?
- When is it more cost-effective to use that second processor as a second computation processor (the Floating policy) rather than a dedicated protocol processor (the Fixed policy)?

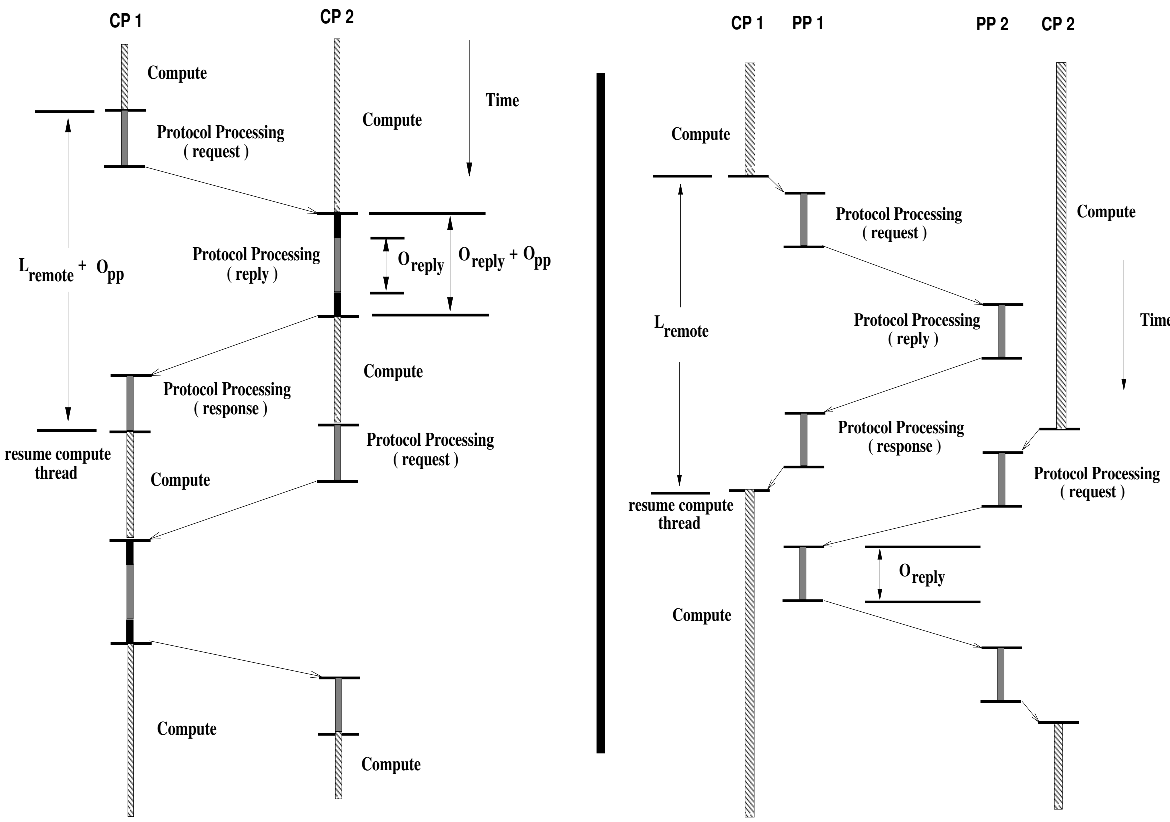

FIGURE 2. Request/reply protocol on Single (left) and Fixed (right)

We base the model on a simple request/reply protocol, a general paradigm employed by many parallel computing paradigms. Figure 2 (left) illustrates the case of only one processor per node (Single). The (compute) processor CP_1 sends a request to the destination node CP_2. The destination node interrupts the local compute thread, invokes the protocol handler, which sends the appropriate reply. Finally, the reply arrives back at CP_1 where a handler takes appropriate action and resumes the computation thread.

Figure 2 (right) illustrates the same remote request/reply, but for a system with dedicated protocol processors (PP_1 and PP_2). The protocol processor PP_1 reduces the overhead on the requesting node by eliminating two context switches and their resulting cache pollution. The protocol processor PP_2 eliminates the interrupt overhead on the destination node as well as overlapping the protocol processing with the computation processor.

Our model estimates the performance of these three systems as functions of the following variables:

M = number of remote requests per processor,
C = mean computation time between remote requests (i.e., the inter-request time),
L_{remote} = remote request/reply latency *with* a dedicated protocol processor,
O_{reply} = overhead of protocol processing on the destination node,
O_{pp} = overhead *saved by* a dedicated protocol processor.

M and C are characteristics of the application and L_{remote} and O_{reply} depend upon the application and network protocols and implementations. O_{pp} depends upon message delivery and context

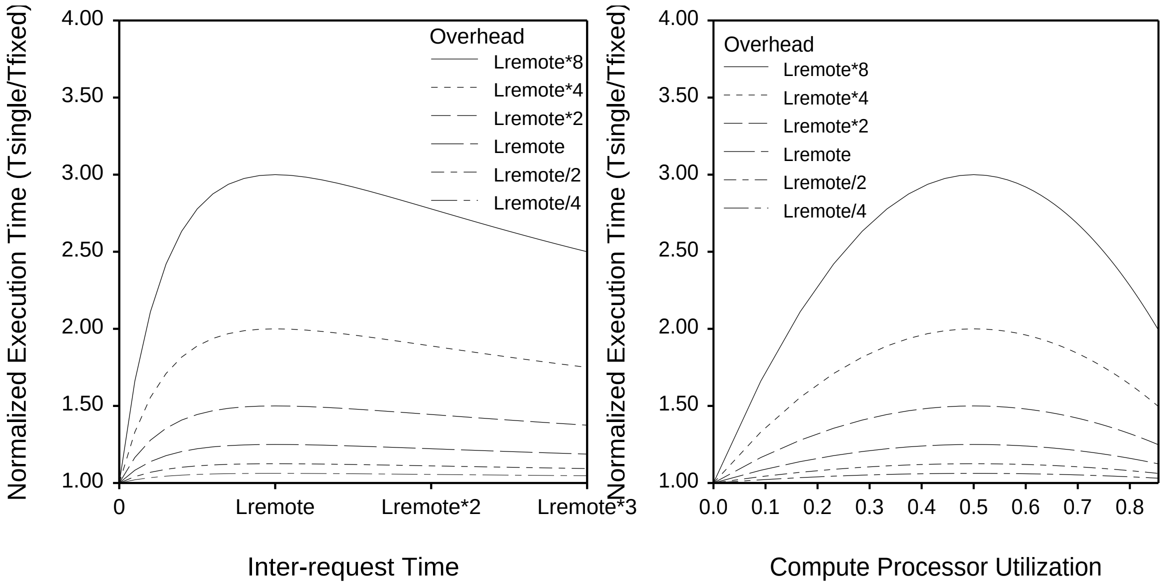

FIGURE 3. Normalized execution time *(T_{single}/T_{fixed})*
The left figure plots normalized execution time versus the computation time between requests; the right figure versus compute processor utilization $C/(C+L_{remote})$.

switch overheads and is defined such that the request/reply latency in the Single system is $L_{remote} + O_{pp}$.

We then estimate the execution time for the three systems as:

$$T_{fixed} = M\,(C + L_{remote})$$
$$T_{single} = M\,(C + L_{remote} + (2\,O_{pp} + O_{reply})\,C\,/\,(\,C + L_{remote}))$$
$$T_{floating} = T_{single}\,/\,2$$

The T_{single} equation assumes that all processors act as destination node for M requests, each incurring overhead of $O_{pp} + O_{reply}$ unless it happens to arrive while the processor is idle waiting for its own remote request (we approximate this probability by $C/(C+L_{remote})$). Similarly, remote latencies are O_{pp} longer unless the remote processor is idle. $T_{floating}$ optimistically assumes the second processor executes half the instructions and issues half the remote requests. These simple estimates obviously do not model contention, load imbalance, synchronization, or cache interference and thus underestimate actual runtimes. More subtle omissions include that Fixed reduces overheads on the requesting node and that the higher overheads of Single and Float lead to a greater susceptibility to queuing delays.

Figure 3 (left) plots the normalized execution times ($T_{single}\,/\,T_{fixed}$) as the mean inter-request time C increases. This graph illustrates the intuitive result that communication-intensive programs (small C) benefit more from a dedicated protocol processor than computation-intensive programs (large C). Similarly, as the program becomes communication-bound, the compute processor becomes idle and acts like a protocol processor, eliminating the extra overhead. Not surprisingly, Fixed performs best when communication and computation are perfectly balanced ($C = L_{remote}$), since both communication and computation processors are equally utilized. The advantage of Fixed decreases as the overhead saved by the protocol processor decreases (small $2\,O_{pp} + O_{reply}$).

Adding a dedicated protocol processor is only cost-effective if the performance gain exceeds the cost increment [31]. Thus if a two-processor workstation node costs one third more than a uniprocessor node, the dedicated protocol processor must improve performance by 33% to be cost-effective. The standard is even higher when we consider the alternative Floating policy. Our simple first order model suggests that the Fixed policy must perform 100% better than the Single policy to be more cost-effective than Floating. Figure 3 (left) shows that this is only true when the overheads eliminated by the protocol processor are significantly larger than the round-trip latency.

If a program becomes too communication intensive, then parallel processing becomes impractical, regardless of the protocol processing policy. Figure 3 (right) plots the normalized execution times against the effective compute processor utilization under the Fixed policy ($C / (C + L_{remote})$). This graph shows that Fixed performs better than (and hence is more cost-effective than) Floating only when the utilization is moderate (e.g., more than 20% but less than 80%) and the overhead ($2O_{pp} + O_{reply}$) is at least eight times the remote latency L_{remote}. The latter is likely only with very light-weight protocols and very fast network interfaces or very slow interrupt overheads.

The model also illustrates a surprising (to us) result: the Fixed policy is more beneficial for light-weight protocols (e.g., active messages) than for heavy-weight protocols (e.g., TCP/IP). This runs counter to the common intuition that a dedicated protocol processor helps off-load heavy-weight protocols from the computation processor. The result follows from the observation that Fixed does best when the extra overhead of Single ($2O_{pp} + O_{reply}$) is much greater than the round-trip latency L_{remote}. But since $O_{reply} < L_{remote}$, this is only true when $2O_{pp} >> L_{remote}$; in other words, when the overhead of interrupts and context switches is much greater than the round-trip latency. This result suggests that dedicated protocol processors may become more attractive as interrupt latencies go up (due to faster processors) and network latencies go down (due to faster links and better network interfaces).

Multiple Compute Processors and Multithreading

The Fixed policy becomes more attractive with multiple compute processors per node. Sharing a protocol processor among multiple compute processors amortizes its cost, decreasing the performance improvement needed to be cost-effective. However, sharing also increases the likelihood for contention, since the protocol processor will be more highly utilized. Furthermore, the Floating policy also benefits from more processors, because the likelihood that at least one is idle—and thus acting as protocol processor—increases.

During program phases that exhibit high communication, Floating approximates the Fixed policy. However, the overhead of Floating is always somewhat higher than Fixed, since the acting protocol processor must at a minimum decide when to return to computation. When a processor does resume computation, there are additional synchronization and cache pollution effects when another processor becomes acting protocol processor. The higher overheads lead to a greater probability of queueing delays under Floating, and when saturated this policy will produce lower bandwidth than the Fixed policy.

Multithreading also shifts the balance toward the Fixed policy. For example, in multithreaded distributed shared memory systems [2,12], a remote miss causes the (compute) processor to switch to executing a new thread. If the context switch overhead (T_{cs}) is less than the remote miss latency, than multithreading should increase processor utilization [1]. Under the Fixed policy, the compute processors' utilization can grow as high as $C / (C + T_{cs})$, if the application has sufficient parallelism and the protocol processor does not saturate. The protocol processor becomes the bottleneck when the average overhead of processing a remote miss ($O_{request} + O_{reply}$) exceeds the mean computation time plus context switch overhead ($C + T_{cs}$). Protocol processor saturation is even more likely when multithreading is combined with multiple compute processors. However, since the Fixed policy minimizes the overheads of remote accesses, the saturation point will be higher (better) for Fixed than for Floating. Conversely, if multithreading does not saturate the protocol processor, then Floating is likely to be the better alternative.

In summary, our simple model illustrates that the Floating policy is likely to be much better than the Fixed policy unless multiple processors, multithreading, or both cause the application to become communication intensive[1]. Of course, our simple model omits too many details to provide more than intuition. Synchronization overheads, contention, and more complex protocol events each affect the trade-offs in subtle ways. In the remainder of this paper we describe the implementation and evaluation of the three policies in a distributed shared memory system on a network of workstations.

5 Implementation on a NOW

We have implemented the three protocol processing policies in a distributed shared memory system running on a network of multiprocessor workstations. Each processing node is an unmodified SPARCStation 20 with 4 66MHz HyperSparc processors, each with a 16 Kbyte instruction cache backed by a 256K direct-mapped unified cache. The SS-20s are connected via Myrinet adaptors on the SBus (the standard I/O bus) and an 8-port Myrinet switch. The remainder of this section describes the relevant details of the threads package, fine-grain distributed shared memory system, messaging layer, and protocol processing implementations.

Threads

Our implementation is based on a locally-developed light-weight user-level thread package. The thread package creates a Solaris LWP (kernel thread) per processor and non-preemptively runs the (user) threads on them. The thread package provides intra-node (local) spin-lock, semaphore and barrier operations. Threads sleep on a semaphore `wait` or at an incomplete barrier; semaphore `signal` and barrier completion wake them up. A thread scheduler is invoked either when a thread goes to sleep or explicitly wakes up another thread. The scheduler used for this paper uses a single thread queue per node, and hence provides no processor affinity. We experimented with a scheduling policy that bound threads to processors—to reduce cache pollution and eliminate synchronization in the scheduler—but the load imbalances made it run slower for our applications. Threads also use only a single SPARC register window to facilitate fast context switching; applications are compiled using GCC 2.6.3 with the `-mflat` argument to use a flat (24 integer + 8 global) register file and eliminate `save` and `restore` instructions.

Fine-grain access control

Fine-grain access control provides the foundation of the distributed shared-memory system used in this study [23]. The Tempest interface defines a set of mechanisms to control sharing a region of memory at subpage granularities. These mechanisms allow user-level software to manipulate logical tags on 32-byte blocks of memory. These tags have possible values of *Invalid*, *Readonly* and

1. Prefetching will also shift the balance towards Fixed but is not evaluated in this study.

Writable; read accesses to a Readonly or Writable block and Write accesses to a Writable block proceed as normal. Read or Write accesses to an Invalid block or Write accesses to a Readonly block incur block access faults. These faults invoke a user-registered handler much like the UNIX `signal` interface.

The study in this paper uses a software implementation of fine-grain access control similar to that in Blizzard-S [23]. We use EEL [16], a tool for executing editable files, to instrument each shared memory operations with tag lookup code that enforces the Tempest access control semantics. The instrumentation is wrapped around set and clear operations on a cached memory location to guarantee the tag check and the instrumented memory operation together appear atomic. On a block access fault, the control is transferred to a stub that saves the global registers and the condition codes and submits a handler dispatch request to the protocol processor.

This all-software approach causes code and time dilation to the instrumented performance, and is not intended as a high-performance implementation. However, the results of this study are largely independent of this implementation decision.

Messaging Layer

On this system, the Tempest active message interface is implemented on top of the Illinois FM library, which provides low-latency communication through the Myrinet switch [20]. The library implements an active message interface and guarantees delivery of messages. The FM interface provides `send` calls to inject messages and an `extract` call to receive any pending messages and dispatches the corresponding handlers. The FM library provides no support for interrupts, so all nodes must periodically poll the Myrinet interface card via the extract call.

Tempest defines two types of handlers to be dispatched: message handlers, and block access fault handlers. The dispatch code calls the message library, which in turn invokes message handlers on availability of messages. The code also sweeps an array of pending block access faults. When a thread takes a block access fault, it enters the handler dispatch information into the array entry corresponding to its processor id. When multiple threads fault on the same block, the dispatch code runs a single handler on the behalf of all threads. Faulting threads first block until the dispatch code accepts and processes their request, and then perform a wait on a semaphore associated with the block address. The thread package subsequently schedules other threads waiting to run. Up on a reply message, a message handler calls a resume routine which performs a signal on the semaphore and wakes up all threads waiting for the block.

In the Fixed policy, the dedicated protocol processor executes the dispatch code in a tight-loop. The remaining processors use the normal thread scheduler to run compute threads.

The Floating policy runs compute threads on all the processors in a node. We use EEL to instrument the back edges of loops to periodically call the dispatch code. As in the case of block access faults, the transfer of control from the thread to the dispatch code happens through a stub which saves the global registers and condition codes. Polling the interface card involves accessing the I/O bus and can not be performed efficiently. The saving and restoring state through the stub also introduces overhead. We therefore, opt for calling the dispatch code every N iterations of the loop (N is currently set to 128).

In order to prevent multiple processors from polling simultaneously, we implement a round-robin polling ownership policy. When a thread returns from a call to the dispatch code, it transfers the

Name	Input Data Set
em3d	8160 nodes, degree 5, 5% remote, distance span 2, 10 iterations
gauss	672 x 672 matrix
tomcatv	768 x 768 matrices, 10 iterations

TABLE 1. Benchmark Input Parameters

ownership to another processor in a round-robin fashion. When a processor blocks waiting for runnable threads, it enters a non-instrumented tight-loop that waits for the processor to become the poll owner and then directly calls the dispatch code. The latter has the advantage of polling frequently (not periodically) and saves the overhead of going through the stub.

6 Performance on a NOW

This section presents preliminary measured performance of our distributed shared memory system running on the three policies on our network of workstations. Section 6.1 presents microbenchmark results that measure the best-case latencies and overheads of the implementation. The remaining sections analyze the performance of the applications in turn.

Table 1 lists the applications and input parameters used in this study. All programs are Tempest-compliant applications and communicate using a combination of active messages and transparent shared memory. Coherence is maintained using a sequentially-consistent protocol with 64-byte blocks. The applications use PARMACS directives [6] to create a process per node, allocate shared memory, and synchronize between nodes. Each PARMACS process is multithreaded using our locally-developed thread package. Threads share the same address space and therefore can share the remote data.

6.1 Microbenchmark Results

Table 2 presents the results from a simple microbenchmark experiment to determine the latency and overhead of remote misses in our distributed shared memory system. The latency numbers were computed by counting the time for one processor to perform 10,000 remote misses in a tight “miss” loop with the home node spinning in a tight compute loop. Overhead numbers were computed by saturating the requester (responder) and calculating the time required to process each request (response).

Table 2 shows that the remote miss latency increases from 113 us under the Fixed policy to 168 us under the Single policy, or 49%. This increase is largely due to higher overheads to dispatch the protocol handlers on both the requesting and responding nodes, which increase by 44% and 59%, respectively. The Other category includes network latency, which should be the same for both models, and miss detection. Miss detection is the time from when the access control lookup detects the block access fault until it is dispatched by the protocol processor. Since under Single the same processor acts as both protocol and compute processor, this overhead is obviously larger.

	Fixed	Single	%increase
Round-trip remote miss time	113 us (100%)	168 us (100%)	49%
Requester Overhead	54 us (48%)	78 us (46%)	44%
Home Node Overhead	27 us (24%)	43 us (26%)	59%
Other (includes network latency and miss detection)	32 us (28%)	47 us (28%)	47%

TABLE 2. Remote miss latency and overheads for Fixed and Single

Round-trip miss calculated with requester in tight miss loop and requester in tight compute loop. Overheads calculated via microbenchmarks that saturate requester (responder) and calculate time per request (response).

6.2 Tomcatv

`Tomcatv` is a parallel version of the well-known SPEC benchmark [24]. The program performs an iterative stencil computation on a pair of matrices allocated in shared memory and five local matrices. Work is partitioned by assigning a block of rows to each thread. On each iteration, the threads make three passes over the rows they own: update matrix, calculate locally maximum residual value, adjust values based on globally maximum residual. A synchronization step is required between the second and third passes to calculate the globally maximum residual. All communication in this benchmark occurs through transparent shared memory and is dominated by near-neighbor sharing of bordering rows due to the stencil nature of the computation.

Tomcatv is compute-intensive because most of the data is either allocated locally, or remains cached in memory for the duration of the program. Because the computation is a stencil, communication occurs primarily on the border rows between two neighboring nodes. For our data sets, Single spends up to 85% of the time computing. Using our model from Section 4 and our overhead estimates for Single from the microbenchmarks, we expect that adding a dedicated protocol processor will not significantly improve performance. Figure 4 (left) shows the performance of Single, Fixed, and Floating for one thread per CPU; as predicted, the results show that Fixed with two processors runs less than 10% than Single.

Since threads are assigned complete rows and only border rows are actively shared, only two threads actually incur remote misses and participate in communication; all other threads simply compute on data that is always cached locally. As a result, communication and computation may be overlapped between at most two threads on each node, making tomcatv an unlikely candidate for multithreading. Figure 4 (right) corroborates this result; running two threads per processor does not result in speedups. The program could be restructured to allow better overlap, but this is unlikely to be beneficial since the computation to communication ratio is so high.

Conversely, adding compute processors should substantially improves performance because the application spends a large fraction of its time in parallelizable local computation. This also allows communication to occur in parallel, but, as with multithreading, should not be a significant factor. Figure 4, illustrates that Floating with 2 processors is a nearly a factor 2 faster than Single. How-

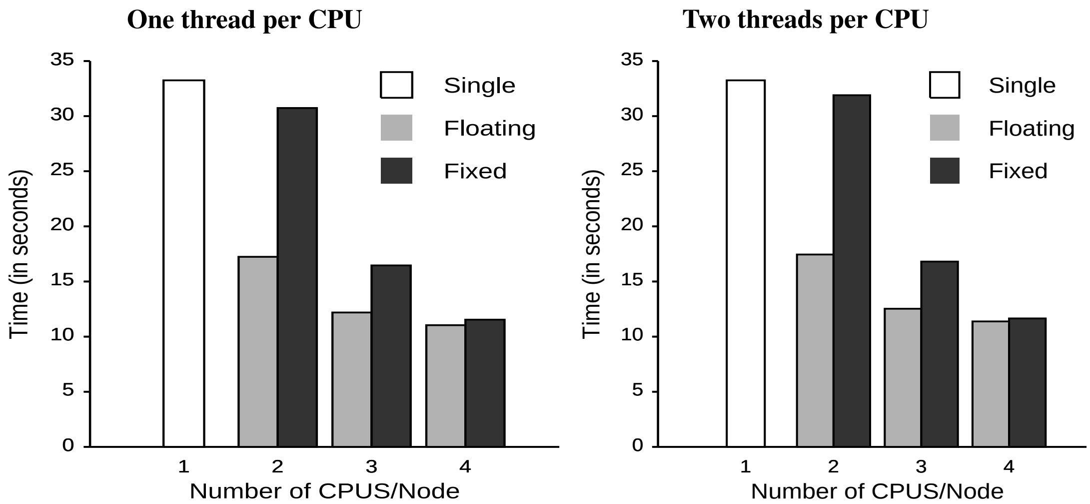

FIGURE 4. Tomcatv performance

ever, since the processor cache is not large enough to hold the entire data set, the bus eventually becomes a bottleneck with enough compute processors. Our results show that with up to three processors, Floating outperforms Fixed because it uses one more processor for computation. Once the computing processors push the bus to its limit, Fixed improves performance slightly by minimizing the bus traffic required to process protocol actions (since the protocol code and state are not flushed from the protocol processors cache by the computation).

6.3 Em3d

`Em3d` models propagation of electromagnetic waves through objects in three dimensions [9]. The program iterates over a bipartite graph made up of directed edges between nodes representing electric and magnetic fields (E and H nodes, respectively). The graph is partitioned over all threads so that each thread gets an equal share of the computation. In the first of two phases, threads allocate and initialize the nodes they own, and create edges between them and other nodes in the graph. In the second phase, each thread iterates over its sets of E and H nodes, first computing new E values as a weighted sum of the its neighboring H nodes, followed by an analogous step to update the H nodes. In this implementation, the program's build phase uses remote store operations similar to the original Split-C version of the code. The second phase uses transparent shared memory for communicating data among the threads.

In em3d, communication occurs when an edge connects two graph nodes allocated on different processor nodes. Since a thread spends only 1.7 microseconds per edge to compute the new value of a graph node, even if only 5% of the edges are to remote nodes, the computation per remote edge is only 34 microseconds. If each remote edge causes a remote miss (as in this version of the program), even with the best-case round-trip miss time of 113 microseconds the processor will only be 23% processor utilized. With low utilization, Single spends most of time acting as a protocol processor, not incurring the overhead of switching between computation and protocol processing. Consequently, adding a dedicated protocol processor should not result in a significant performance boost. Figure 5 indicates that indeed the extra protocol processor in Fixed improves the performance by only 16% over Single.

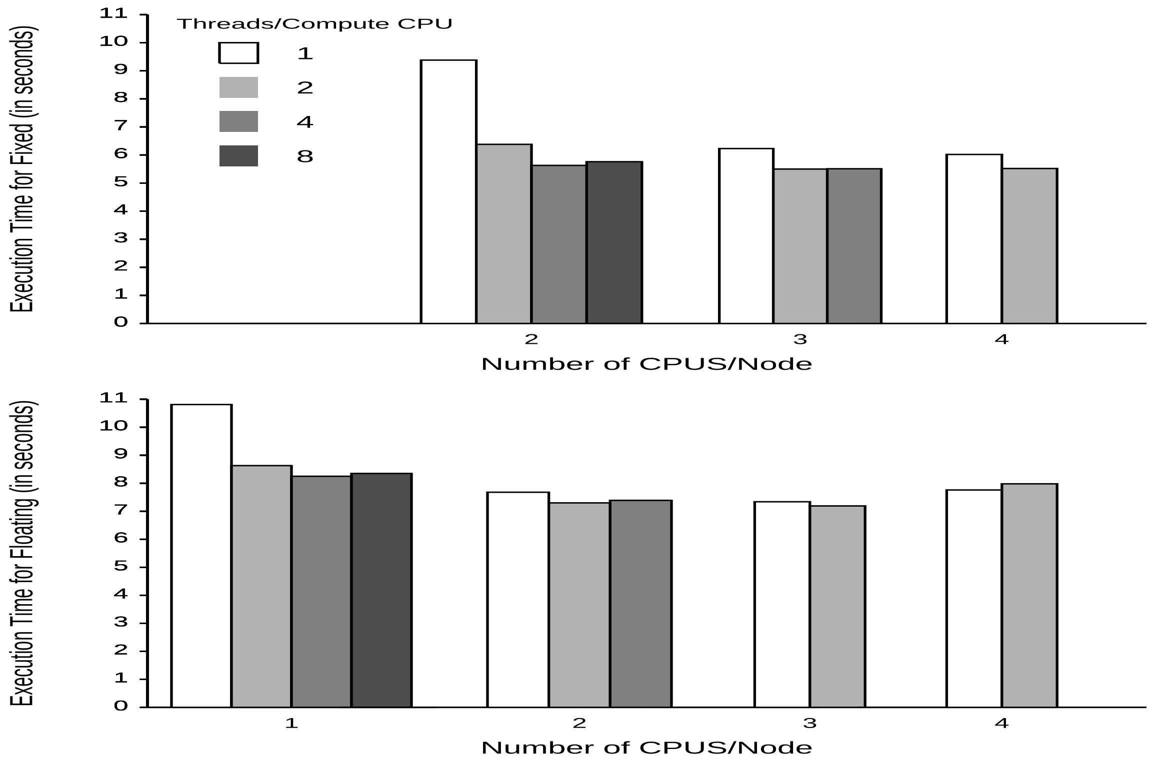

FIGURE 5. Fixed (top) and Floating (bottom) performance for Em3d

We can improve em3d's performance by overlapping multiple requests for remote edges. The higher request rate allows for better utilization of protocol processing bandwidth, until the bandwidth reaches a saturation point either on the requester or on the responder side. With our data set for em3d, remote edges from one (machine) node are uniformly distributed over the other nodes. Therefore, there are no hot spots on the protocol traffic and all protocol processors saturate at the same peak bandwidth.

There are two ways to exploit the available protocol processing bandwidth. Multithreading increases the request rate by switching between threads, as long as the switch time is lower than the round-trip time of a miss. Adding processors similarly allows for higher number of simultaneous requests, increasing the request rate. Figure 5 indicates that both multithreading and multiprocessing can utilize the available bandwidth effectively [27]. That is, the request bandwidth is not limited by the switch time. Saturation for both Floating and Fixed occurs quite rapidly as we increase the number of threads and/or processors; this follows from the relatively high protocol processing overheads of our remote miss handlers. Fixed is 30% faster than Floating at the saturation point. Fixed can sustain higher peak bandwidths because Floating incurs the extra overhead of switching between computation and protocol processing.

6.4 Gauss

`Gauss` is a kernel that uses Gaussian elimination to solve a linear system of equations. The rows of the coefficient matrix are evenly distributed to the threads, which execute the computation in three phases. In the first (unmeasured) phase, threads initialize their part of the matrix with pseudo-random numbers. In the second phase, threads solving the equations one column at a time, computing

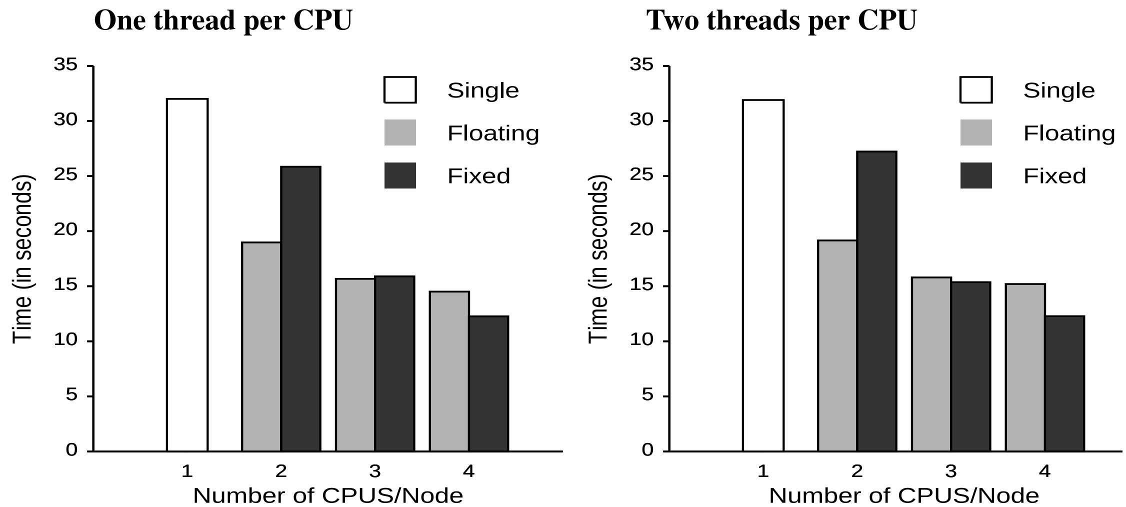

FIGURE 6. Gauss performance

a maximum pivot per column. Computed maximums are locally reduced among the threads on a node to obtain a per node local maximum. A recursive halving algorithm—implemented using Tempest's active messages—is used to calculate a global maximum. All threads then read the row of the maximum pivot from shared memory and use it for the next iteration. In the third phase, the threads iterate through the columns starting from the last, and for each column globally compute a value, participate in a barrier synchronization and subtract the computed value from all their rows. The last phase exhibits load imbalance and does not speedup as the number of threads increases. However, it does not account for a large fraction of the execution time [7].

Reading the maximum pivot row is a potential bottleneck since all threads read the row before computation begins. Thus the faster the protocol processor can deliver the data, the faster the program runs. In addition, "broadcasting" the pivot and the reduction are the only forms of communication; since they are both synchronous operations, communication cannot be overlapped with computation and gauss will not benefit from multithreading.

Figure 6 indicates that Fixed improves performance over Single by 28%. The two-threaded configurations (right) do not exhibit any improvements over the single-threaded configurations (left). Both Floating and Fixed effectively exploit the parallelism available in the computation up to 3 processors, after which the broadcast of the pivot row accounts for a large fraction of the running time. Floating initially outperforms Fixed benefiting from the extra compute processor. With the broadcast eventually dominating the running time, Fixed sustains a higher bandwidth of protocol operations delivering the maximum pivot row at a faster rate.

7 Summary and Conclusions

In this paper, we examined how the protocol processing should be scheduled on a network of multiprocessor workstations used as a parallel computer. Previous systems such as the Intel Paragon have dedicated a processor specifically for protocol processing. We compare this Fixed policy with the alternative policy of using all processors for both computation and communication.

We use a simple analytic model to illustrate the trade-offs between the policies. The model shows that: i) adding a dedicated protocol processor to a uniprocessor node is unlikely to be cost-effective and even less likely to outperform the Floating policy; ii) a dedicated processor is more advantageous for light-weight protocols (e.g., active messages) than for heavy-weight protocols (e.g., TCP/IP), iii) the Fixed policy becomes more advantageous when multiple compute processors and multithreading increase the bandwidth per node. The breakeven point depends upon the number of processors, overhead of invoking protocol operations, and application parallelism.

Finally, we presented preliminary measurements from a distributed shared memory system running on a dedicated network of Sun SparcStation-20s connected by a Myrinet network. The measured performance on four nodes showed that Fixed outperformed Floating only for the very communication intensive program em3d or when there were four processors per node. The three applications clearly illustrate the predicted breakeven point between Fixed and Floating.

References

[1] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. of the 17th Int'll Symposium on Computer Architecture*, pages 104–114, June 1990.

[3] Tom Anderson. NOW: Distributed Supercomputing on a Network of Workstations, September 1993. Presentation at 1993 Fall ARPA HPC Software PI's meeting.

[4] Tom Anderson, David Culler, and David Patterson. A Case for Networks of Workstations: NOW. Technical report, Computer Science Division (EECS), University of California at Berkeley, July 1994.

[5] Boon Seong Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. CSG Memo 354, MIT, February 1994.

[6] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfieldand Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., 1987.

[7] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[8] et al. Craig Stunkel. The SP2 High-Performance Switch. *IBM System Journal*, 34(2), 1995. to appear.

[9] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[10] J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–174, March-April 1993.

[11] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[12] Vincent Freech, David W. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *First USENIX Symposium on Operating Systems Design and Implementation*, pages 201–214, November 1994.

[13] Intel Corporation. Paragon Technical Summary. Intel Supercomputer Systems Division, 1993.

[14] Randy Katz. ASPLOSVI Keynote Address, October 1994.

[15] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[16] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995. To Appear.

[17] Meiko World Inc. Computing Surface 2: Overview Documentation Set, 1993.

[18] Alan Montz, David Mosberger, Sean O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A Communications-Oriented Operating System. Technical Report TR 94-02, Department of Computer Science, University of Arizona, 1994.

[19] David Mosberger, Larry L. Peterson, and Sean O'Malley. Protocol Latency: Mips and Reality. Technical Report TR 95-02, Department of Computer Science, University of Arizona, 1995.

[20] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Myrinet Fast Messages, March 1995.

[21] Steve Reinhardt, Robert Pfile, and David A. Wood. T-Zero: Prototyping distributed shared-memory on a network of workstations. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1995.

[22] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[23] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* , pages 297–307, October 1994.

[24] SPEC. SPEC Benchmark Suite Release 1.0, Winter 1990.

[25] Mark S. Squillante and Edward D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 4(2):131–143, April 1990.

[26] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.

[27] Radhika Thekkath and Susan J. Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, San Jose, California, 1994.

[28] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Me mory Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles (SOSP)*, pages 159–166, December 1989.

[29] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP)*, pages 26–40, October 1991.

[30] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[31] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, Feb 1995.