Scheduling Communication on an SMP Node Parallel Machine

Babak Falsafi and David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
{babak,david}@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 are often the performance bottleneck. Several current and proposed parallel systems address this problem by dedicating one general-purpose processor in a symmetric multiprocessor (SMP) node specifically for protocol processing. This scheduling convention reduces communication latency and increases effective bandwidth, but also reduces the peak performance since the dedicated processor no longer performs computation.

In this paper, we study a parallel machine with SMP nodes and compare two protocol processing policies: Fixed, which uses a dedicated protocol processor; and Floating, where all processors perform both computation and protocol processing. The results from synthetic microbenchmarks and five macrobenchmarks show that: i) a dedicated protocol processor benefits light-weight protocols much more than heavy-weight protocols; ii) Fixed improves performance over Floating when communication becomes the bottleneck, which is more likely when the application is very communication-intensive, overheads are very high, or there are multiple (i.e., more than two) processors per node; iii) a system with optimal cost-effectiveness is likely to include a dedicated protocol processor, at least for light-weight protocols.

1 Introduction

Parallel computers are emerging as the supercomputers of choice, exhibiting impressive performance on many classes of large and important applications. Commodity microprocessors form the core of computation in these machines, exploiting large sales volumes and rapid technology improvements to provide superior cost-performance [1]. Low-level communication in these machines is implemented in the form of messaging over high speed networks. Both applications programs and the system software employ a variety of protocols to schedule and coordinate communication and computation. These protocols range from low-level messaging functionality, such as checksumming, reliable delivery, and flow control, to high-level parallel programming abstractions, like coherent distributed shared memory.

Systems can implement these protocols in either hardware or software. Many researchers and vendors favor software implementations due to their flexibility [7], reduced manufacturing cost [17], shorter design times [16], and increased portability [10,25]. However, as the performance of network interface hardware improves, software protocol overheads begin to dominate end-to-end communication time [11].

To address this problem, many distributed-memory parallel machines employ an embedded processor to off-load the primary (computation) processor(s). For example, the Meiko CS-2, IBM SP-2, and proposed Stanford FLASH [13] and Wisconsin Typhoon [23] 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.

This paper studies an alternative approach which employs one of several processors on a symmetric multiprocessor (SMP) node for protocol processing. Small SMP systems—such as the recently-announced Intel Pentium Pro servers—are becoming widely available, making them attractive building blocks for parallel computers [16]. The Intel Paragon, and the proposed MIT StarT-NG [4] and Wisconsin Typhoon-0 [20]

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, an IBM graduate fellowship, and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. 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.

0-8186-7764-3/97 $10.00 © 1997 IEEE

systems all dedicate one processor of a multiprocessor node specifically for protocol processing.

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, Womble, et al., demonstrated that using the Paragon's protocol processor for computation (via a low-level cross-call mechanism under SUNMOS) improved performance on Linpack by more than 50% [28]. Similarly, others have shown that a dedicated protocol processor provides little benefit for systems with large communication latencies and overheads as in ATM [12] or HIPPI [6] networks.

In this paper, we ask the question: *"when does it make sense to dedicate one processor in each SMP node specifically for protocol processing?"* The central issue is when do the overheads eliminated by a dedicated protocol processor offset its lost contribution to computation? We address this question by examining the performance and cost-performance trade-offs of two scheduling policies:

- *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 limit our study by only considering SMP nodes interconnected using relatively simple network interfaces—similar in complexity to the Thinking Machines CM-5 NI [8]—where most protocol processing occurs on a regular processor. In contrast, other research has examined complex, powerful network interfaces that use embedded protocol processors to off-load protocol processing [13, 23]. While both simple and complex network interfaces are interesting, we focus on the former, lower-cost alternative.

We analyze the policies using two sets of experiments. In the first set, we use synthetic microbenchmarks to examine two simple request/reply protocols and show the following results:

1. A dedicated protocol processor benefits light-weight protocols (e.g., Split-C get/put [27]) much more than heavy-weight protocols (e.g., page-based DSM [2]). This is because the overheads saved by the Fixed policy represent a significant fraction of the light-weight protocol's total round-trip time.

2. Fixed reduces protocol processor *occupancy* [9]—i.e., the time it takes to handle a protocol event—and thus performs better than Floating when communication becomes the bottleneck. This is more likely when the application is very communication-intensive, protocol overheads are very high, or there are multiple (i.e., more than two) processors per node.

3. A system with optimal cost-effectiveness—in the number of processors per node—is likely to include a dedicated protocol processor when overheads are a significant component of protocol processor occupancy. This follows because the incremental cost of an additional processor is typically less than the relative performance increase provided by lower protocol processor occupancy.

In the second experiment, we examine the same policies using five shared-memory applications. These applications run on a fine-grain distributed shared-memory machine based on the Tempest interface [23]. Besides corroborating our findings from the first experiment, the results also show that communication patterns in some applications decrease the benefit of the Fixed policy. Under Floating, an idle processor acts much like a dedicated protocol processor, which occurs more frequently with bursty and synchronous communication.

The next section summarizes the system architecture and simulation methodology. Section 3 describes the two protocol processing alternatives in more detail. Section 4 briefly describes the cost model and system characteristics that affect the performance of the system under the policies. Section 5 and Section 6 present results from our microbenchmark and macrobenchmark experiments, respectively. Finally, Section 7 concludes the paper.

2 System Architecture

Figure 1 illustrates the general class of parallel machines that we study in this paper. Each node of this system is modeled after a SPARCStation 20 consisting of one or more 200 MHz HyperSparc processors, each with a 1M direct-mapped data cache, connected by 100 MHz split-transaction bus.[1] We assume perfect instruction cache performance but model contention at the memory bus accurately. A snooping cache-coherence protocol keeps the caches *within* a node consistent. A network interface device—similar to that on a Thinking Machines CM-5 [8]—with a pair of uncached memory-mapped send and receive queues resides on the memory bus and connects the node to a low-latency, high-bandwidth network. We assume a point-to-point network with a constant message latency of 100 cycles but model contention at the network interface device.

An operating system both provides local services and manages the nodes collectively as a single parallel machine [8,1]. Parallel applications follow the SPMD programming model. In this paper, we assume space sharing—where the nodes are logically allocated to separate parallel tasks. More general time sharing is of course possible, but is beyond the scope of this paper.

High performance communication is performed via an active message abstraction [27]. Message arrivals either cause interrupts or the processors poll the network interface to eliminate the interrupt overhead. A memory-mapped interrupt arbi-

[1] In this paper we use *cycle* to refer to processor cycles.

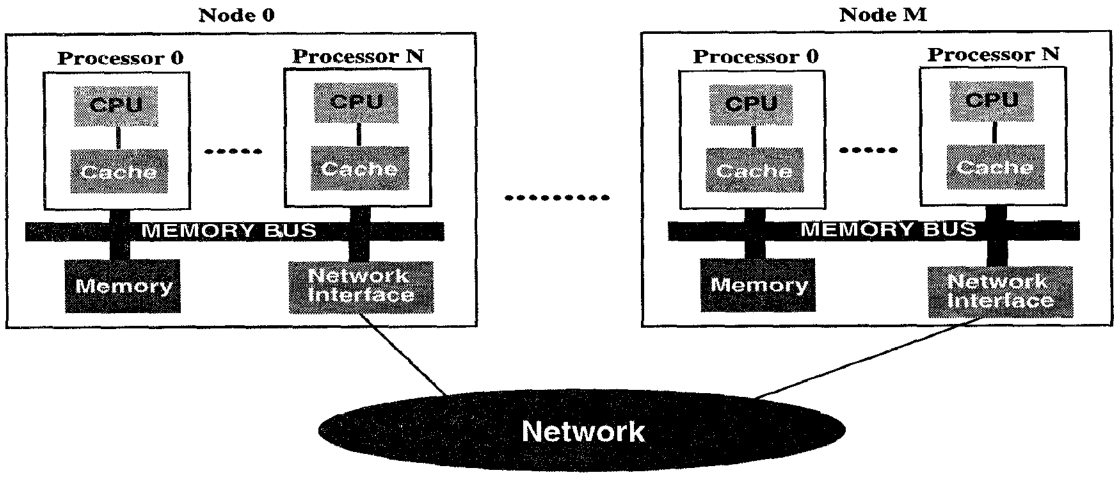

FIGURE 1. A multiprocessor node parallel machine

ter device located on the memory bus distributes interrupts among processors in a round-robin fashion. Unless stated otherwise, we assume an interrupt overhead of 200 cycles, characteristic of carefully tuned parallel computers [22]. We also assume Tempest active message semantics which reduces the need for synchronization by requiring sequential execution of handlers within each node [21].

A fine-grain software distributed shared-memory system extends the coherent shared-memory abstraction beyond a single node. This system is based on the Tempest interface and allocates shared memory at the page granularity, but maintains coherence via a fine-grain access control mechanism [25]. While the results of this paper are largely independent of whether the mechanism is implemented in hardware or software, we assume a hardware implementation via a Typhoon-1 (T1) board [20]. On each node, a T1 board performs a tag lookup to enforce the Tempest access control semantics on shared-memory loads and stores that miss in the cache. On a remote miss, the hardware provides handler dispatch information in a cacheable memory location. T1 also facilitates messaging by providing a cacheable control register for detecting message arrivals in order to eliminate poll traffic from the memory bus.

3 Protocol Processing Policies

In this paper, the term *protocol processing* refers to executing the user and system software needed to manage communication between cooperating nodes. For the distributed shared-memory system we focus on in this study, protocol processing includes executing remote miss handlers—invoked on a fine-grain access control exception—and active message handlers. Because of Tempest atomicity requirements, each node is limited to one processor executing protocol events at any one time. Regardless of the policy we say that this processor is *acting* as protocol processor.

In this study, we examine two scheduling policies for protocol processing: *Fixed* and *Floating*. In Section 4 we qualitatively analyze the cost-performance trade-offs between the different policies and identify application and system characteristics that affect these trade-offs.

Fixed. The Fixed policy dedicates one processor of a multiprocessor node to perform only protocol processing. The dedicated protocol processor executes all the remote miss and active message handlers. By always polling the network when otherwise idle, the protocol processor eliminates the need for message interrupts or polling by the compute processor(s).

Floating. The disadvantage of dedicating a protocol processor is that it may waste cycles that could have productively contributed to 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 are required to ensure timely handling of active messages. On the other hand, once a processor assumes the role of protocol processor, it acts much like a dedicated protocol processor. We use the term *Single* to refer to the special case of a single processor (per node) performing all protocol processing as well as all computation.

4 When does dedicated protocol processing make sense?

In this paper, we pose the question: "when does dedicated protocol processing make sense?" We address this question by evaluating when one of our two protocol processing policies performs better or is more cost-effective than the other. While there are many factors—including system software complexity, and protection [15]—we believe that performance and cost-performance are important.

To quantify cost-effectiveness, we use the simple cost model from Wood and Hill [30]. A change, e.g., adding a second processor, is cost-effective if and only if the increase in cost (or costup) is less than the increase in performance (or speedup). In this paper, we say a system is *cost-effective* if its cost-performance ratio is less than a uniprocessor node's. A system is *most cost-effective* if it achieves the lowest cost-performance ratio. Our simple cost model assumes that a processor represents 30% of the cost of a uniprocessor node.[1] Thus, a two-processor node and a five-processor node have costups of 1.3 and 2.2, respectively.

To answer "when" one policy is better than another, we examine which factors significantly affect performance. In the remainder of this section, we identify and qualitatively analyze four factors that have first-order effects.

Computation/Communication Ratio. Efficient protocol processing helps speed up communication. Compute-intensive applications—such as some dense matrix methods—require little communication. These applications, characterized by having large computation-to-communication ratios, perform well even with very heavy-weight protocols [2]. Thus such applications should not benefit from a dedicated protocol processor. Conversely, other applications have lower computation-to-communication ratios and may benefit from a dedicated protocol processor.

Protocol Processing Overhead. A dedicated protocol processor improves performance by eliminating two types of protocol processing overhead: *direct* and *indirect.* Direct overhead consists of the overheads incurred when a processor assumes or relinquishes the role of protocol processor. As such, it includes the overhead of disabling and enabling interrupts, accessing a lock—that ensures there is only a single (acting) protocol processor on a node—and checking when to relinquish being protocol processor. Direct overhead also includes the overhead of delivering and returning from an interrupt. Indirect overhead consists of cache interference between computation and protocol threads and migration of protocol processor lock [5], code [18] and data [24] among processors. A dedicated protocol processor always eliminates both direct and indirect overheads and becomes beneficial when the overheads it saves are large compared to overall communication time.

Protocol Weight. Protocol weight is a measure of the protocol's total execution time. It is a function of the protocol complexity and the speed of the network interface device. We characterize the weight by end-to-end communication time: heavy-weight protocols have larger end-to-end latencies than light-weight protocols. Protocol weight affects the policy trade-off because for heavy-weight protocols, the overheads saved by Fixed become an insignificant fraction of the overall communication time. Thus, a dedicated protocol processor should be more beneficial for light-weight protocols (e.g., active messages) than for heavy-weight protocols (e.g., page-based DSM). This runs counter to the common intuition that dedicating a protocol processor helps off-load heavy-weight protocols from the computation processor.

Number of Processors per Node. The number of processors per node has several effects on the policy trade-off. First, more processors increase the likelihood that at least one processor is idle (e.g., waiting for a protocol response). Under the Floating policy, such a processor acts as protocol processor, significantly reducing the direct overhead by eliminating interrupts. A dedicated protocol processor, however, saves all of the direct and indirect overhead which may improve performance in the presence of high bus contention. Second, more processors decrease the opportunity cost (in lost computation) of the dedicated processor. Third, by parallelizing the computation *within* a node, multiple compute processors decrease the *apparent* computation-to-communication ratio. This increases the demand for protocol processing which makes performance more sensitive to protocol processor *occupancy* [9]—i.e., the time it takes to handle protocol events. Finally, sharing a dedicated protocol processor among multiple compute processors amortizes its cost, decreasing the performance improvement needed to be cost-effective.

5 Microbenchmark Analysis

In this section we evaluate the Fixed and Floating policies using two simple synthetic benchmarks. We base our benchmarks on a simple request/reply protocol, similar to that employed by many parallel computing paradigms [5,10,2,23]. Figure 2 (left) illustrates the execution of such a protocol under the Fixed policy. The compute processor N1CP submits a request to the protocol processor N1PP, which in turn sends a message. At the destination node, protocol processor N2PP immediately invokes the protocol handler and sends the appropriate reply. Because of the dedicated protocol processor, compute processor N2CP proceeds uninterrupted. Finally, the reply arrives and the handler runs on N1PP, which then resumes the computation thread.

Figure 2 (right) illustrates the same remote request/reply, but for the Floating policy. The (compute) processor N1CP2 submits a request, becomes the protocol processor and sends a message. When the message arrives at node 2, all processors are busy computing. Thus, an interrupt is generated causing processor N2CP1 to act as protocol processor. The requesting processor incurs the overhead of two context switches (to and

[1] The incremental cost of an additional processor varies greatly depending on the processor, memory hierarchy, peripherals, and the overall system cost per node. In many cases the incremental cost may be less than 30% which will shift cost-performance in favor of Fixed.

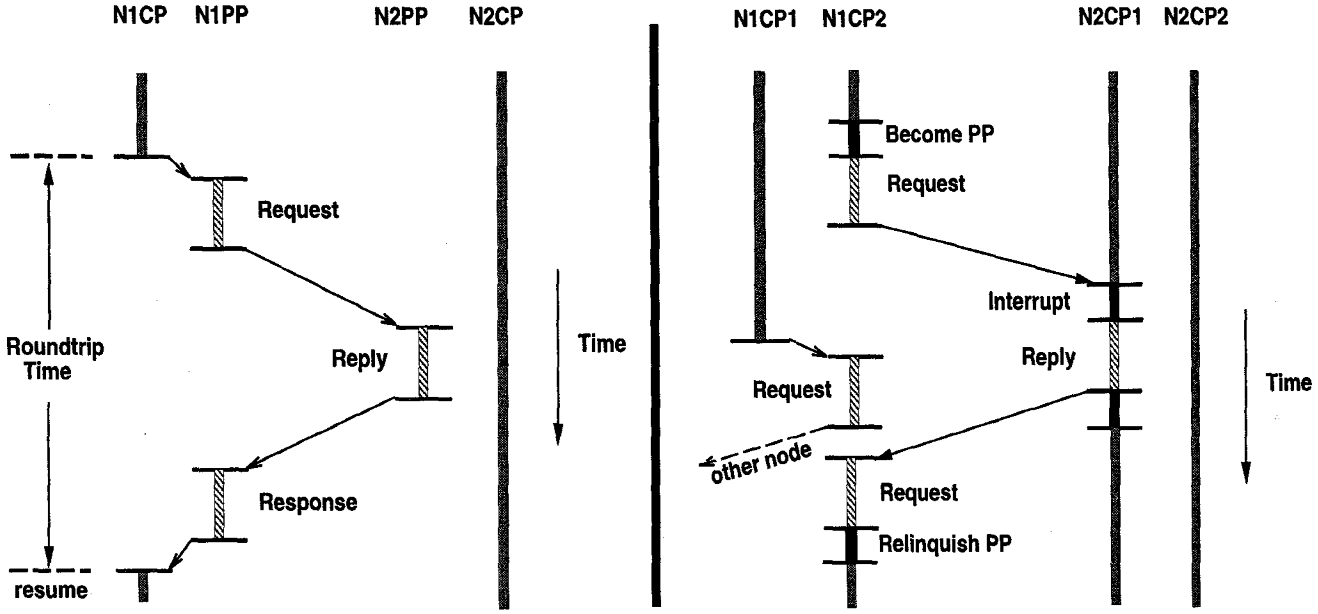

FIGURE 2. Request/reply protocol in Fixed (left) and Floating (right)

from the protocol thread) and the resulting cache pollution. The replying processor additionally incurs the overhead of delivering (and returning from) the interrupt. An idle processor acting as protocol processor (N1CP2) can immediately handle a request by another processor on the node (N1CP1), thereby eliminating the interrupt overhead.

Our benchmarks time the execution of a tight-loop running on a two-node machine. Each iteration alternates between computing and issuing a remote request using a simple request/reply protocol. To induce cache effects, computation is interleaved with uniformly random accesses to a (private) processor-specific segment of the address-space. The size of the segment is equal to the size of the processor cache. We let compute processor caches warm up before the start of measurements.

We experiment with two request/reply protocols with different protocol weights. Our *null-handler* protocol represents the lightest-weight protocol achievable in our simulated system. The protocol handlers do nothing but send the appropriate active message, i.e., the reply handler simply sends a null message back to the requester. Our *fetch-block* protocol is representative of the medium-weight protocols needed to support fine-grain distributed shared-memory systems [10,25]. We do not consider a heavy-weight protocol, e.g., page-based DSM, since prior work indicates that a dedicated protocol processor will be of little use [12,6]. The processors randomly request a 128-byte block of data from the private segment of a remote processor. In addition, the protocol handlers manipulate the memory block state in a protocol table. Both the data block transfer and accesses to protocol table contribute to cache pollution.

We define L_{min} to be minimum round-trip latency under the Fixed policy. Under our system assumptions, the protocol round-trip times are 3 μs for the null-handler protocol and 8 μs for the fetch-block protocol. We vary the following parameters in the experiment:

C = mean computation time between requests,

U = thread compute-utilization in the absence of protocol contention $(C/(C+L_{min}))$,

O_{int} = overhead of handling an interrupt.

We use an exponential random stream with mean C to generate computation times, and adjust C to derive various values for U. We vary O_{int} by delaying a thread upon an interrupt for a fixed number of cycles. The number of iterations in a loop is inversely proportional to the number of compute processors per node, e.g., Floating on a two-processor node and Fixed on a four-processor node execute half and one-third as many iterations as Single, respectively.

Figure 3 (left) compares the performance for the null-handler protocol in one and two-processor node machines. The figure plots execution times of Single and Floating normalized to Fixed as thread compute-utilization increases. Points above the horizontal line indicate that Floating (Single) performs worse than Fixed. The thick and thin lines depict high and low interrupt overheads, respectively. The graphs for Single (solid curves) illustrate the intuitive result that communication-intensive programs (small U) benefit more from a dedicated protocol processor than computation-intensive programs (large U). When the program becomes communication-bound ($C << L_{min}$), however, the compute processor in Single becomes idle and acts like a protocol processor, reducing the number of taken interrupts. The graphs also indicate that, when interrupt overhead is high, even a small number of interrupts severely impacts the execution time.

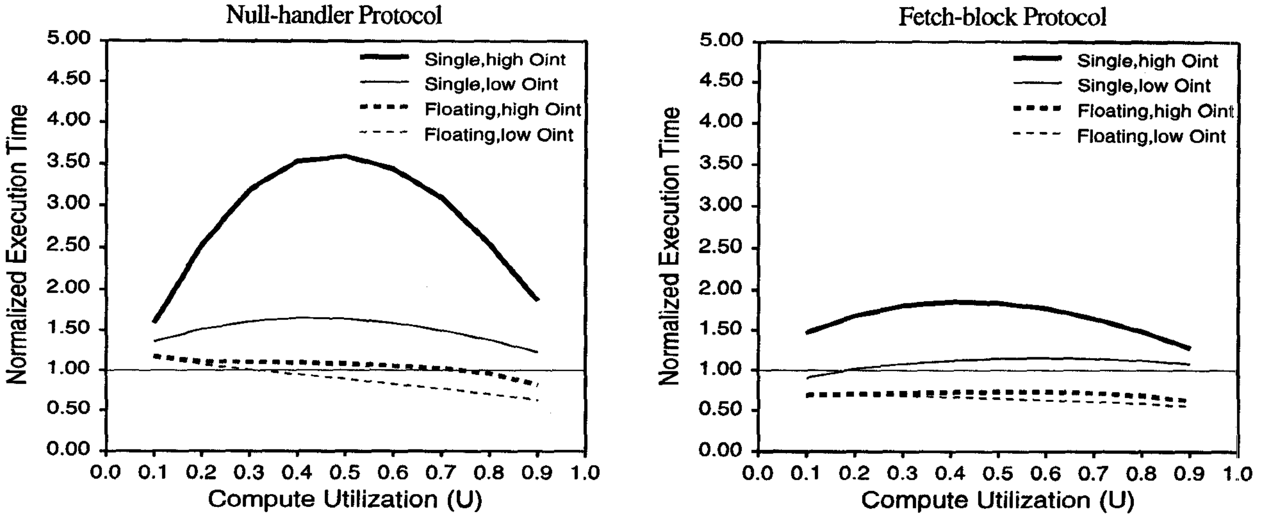

FIGURE 3. Relative performance of Fixed and Floating with varying interrupt overhead
The figures compare execution times of Fixed and Floating versus thread compute-utilization (*U*). The figures plot execution times of Single and two-processor Floating normalized to two-processor Fixed for two values of interrupt overhead (O_{int}); low and high O_{int} correspond to values of 200 cycles (1 μs) and 2000 cycles (10 μs) respectively. Values over the horizontal lines indicate better performance under the Fixed policy.

The dashed curves plot the normalized execution time for a two-processor node under the Floating policy. With high interrupt overheads, the Floating policy behaves like the Fixed policy; the two (compute) processors alternate acting as the protocol processor eliminating the interrupt overhead. Protocol processing migration, however, incurs indirect overhead, slightly reducing performance under Floating relative to Fixed. With low interrupt overheads, there is little benefit from a dedicated protocol processor, but potential gain from improving computation time. Under Floating, both processors perform computation, resulting in significantly better performance at higher compute-utilizations. This is not surprising since our microbenchmark is perfectly parallelizable.

Figure 3 (right) compares the performance of the policies for our fetch-block protocol. The figure corroborates our intuition that a dedicated protocol processor is more beneficial for light-weight protocols than for heavy-weight protocols. The result follows from the observation that Fixed does best when the interrupt overhead is much greater than the round-trip latency ($O_{int} >> L_{min}$). This result suggests that dedicated protocol processors may become more attractive as interrupt latencies go up (due to faster processors) and protocol weights go down (due to faster network interfaces).

This graph illustrates the surprising result that for a communication-bound program and low interrupt overhead, Single outperforms Fixed. This occurs because our synthetic protocol always reads message data into the protocol processor's cache. Under Fixed, the compute processor always misses on message data, resulting in a cache-to-cache transfer. Conversely, under Single, there is only one cache, so the transfer is eliminated. Network interfaces equipped with caches (e.g., CNI [19]) allow protocols to directly transfer data into the compute processor's cache.

Unlike the null-handler protocol, the Floating policy maintains its advantage over Fixed even at low compute-utilizations. Overheads in the fetch-block protocol account for an insignificant fraction of communication time. Moreover, at low compute utilizations the extra compute processor in Floating parallelizes communication by doubling the number of outstanding requests per node, improving performance over Fixed.

Multiple Compute Processors Per Node. More processors per node helps Floating by increasing the likelihood that an idle processor is acting as protocol processor. The added benefit of an extra compute processor, however, diminishes with a larger number of processors. Multiple compute processors also increase the contention for the single protocol processor. Under low compute-utilization, Floating approximates Fixed, since an acting protocol processor eliminates the (direct) interrupt overhead. Fixed, however, provides better throughput by also eliminating the (direct and indirect) overheads associated with protocol thread migration (Section 4).

Figure 4 (left) plots the normalized execution time for the null-handler protocol under the Floating policy, relative to Fixed, while varying the number of processors per node. Floating generally outperforms Fixed on two processors, but as the number of processors increases, greater demand for protocol processing makes communication the bottleneck. Because, the dedicated protocol processor minimizes protocol processor occupancy, Fixed provides greater throughput and can support a larger number of compute processors. Eventually, the proto-

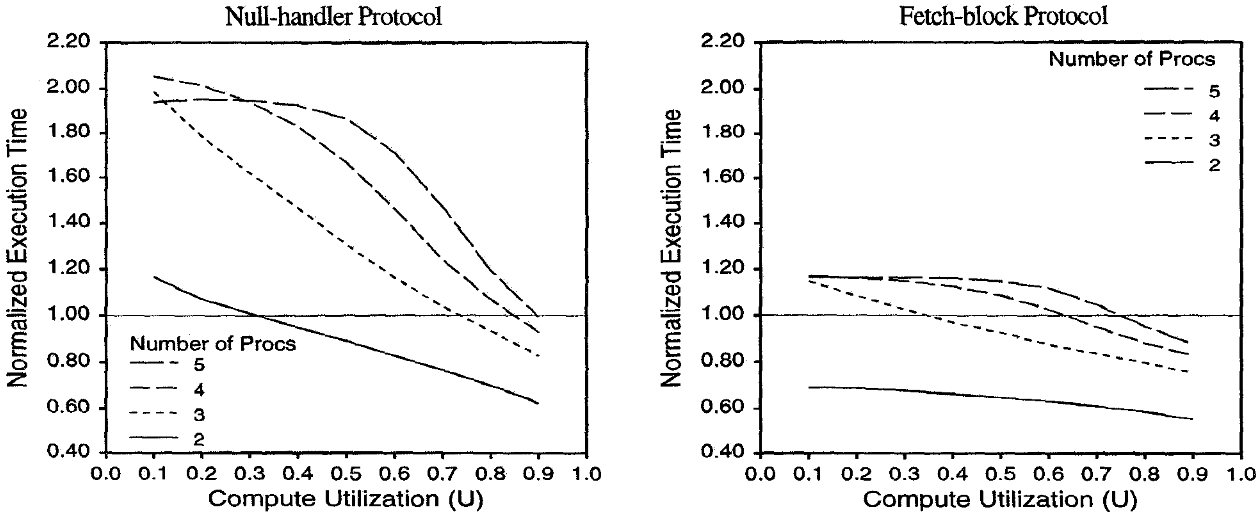

FIGURE 4. Relative performance of Fixed and Floating with varying number of processors

The figures compare execution times of Fixed and Floating versus thread compute-utilization (*U*). The figures plot execution time of Floating normalized to Fixed while varying the number of processors per node (O_{int} = 200 cycles). Values over the horizontal lines indicate better performance under the Fixed policy.

col processor *saturates* regardless of policy and the relative performance levels off.

At low compute-utilizations, Fixed saturates more quickly than Floating with an increase in the number of processors; request rates in Floating remain lower than those in Fixed because an acting protocol processor must return to computation before it can contribute to request traffic. As such, higher request rates in Fixed saturate the protocol processor with a fewer processors. At saturation, however, lower occupancy in Fixed nearly doubles the performance over Floating.

Compute-intensive programs take advantage of the extra compute processor in Floating to improve computation time. An increase in the number of processors, however, gradually diminishes Floating's advantage over Fixed because the added benefit of an extra compute processor becomes insignificant.

Figure 4 (right) plots the same graphs for the fetch-block protocol. Much like the null-handler protocol, Fixed outperforms Floating when protocol processor utilization is high, i.e., there are more than two processors per node and compute-utilization is low. At saturation, however, Fixed improves performance only by 20% because the overheads it saves are a small fraction of total (per-request) protocol processor occupancy.

Cost/Performance. Cost-performance also varies with an increase in the number of processors. Cost-performance only improves if the performance improvement from an extra processor is larger than the cost-increment. Adding processors to a node helps reduce the computation time, but also increases contention for the protocol processor. When communication becomes the bottleneck, cost-performance degrades with each extra (compute) processor. Adding a dedicated protocol processor, however, may improve cost-performance by decreasing protocol processor occupancy and increasing throughput.

Figure 5 (left) illustrates cost-performance for our null-handler protocol. The figure plots cost-performance ratio where 1 represents a uniprocessor node. We examine both policies at two compute-utilizations, against the number of processors per node. Values under the horizontal line (at 1) correspond to systems that are cost-effective—i.e., systems with better (lower) cost-performance than a uniprocessor node. Adding a dedicated protocol processor to a uniprocessor node is cost-effective for both low (*U=0.3*) and high (*U=0.7*) compute-utilizations. Thus, the overhead saved by a dedicated protocol processor justifies the additional cost. Using the second processor for computation takes advantage of the extra parallelism available in the benchmark and improves performance further, resulting in an even more cost-effective system.

Surprisingly, the Fixed policy always provides the most cost-effective system for our null-handler protocol. Fixed can always accommodate a larger number of (compute) processors than Floating, because of its lower protocol processor occupancy. A larger number of processors also reduce the relative cost-increment from an additional processor. The combined effect drives cost-performance lower under Fixed.

Figure 5 (right) illustrates the cost-performance graphs for the fetch-block protocol. Unlike the null-handler protocol, adding a dedicated protocol processor to a uniprocessor node is not cost-effective, whereas using a second processor for computation is. Similarly, Fixed is no longer most cost-effective. Communication rapidly becomes a bottleneck—with more than two processors per node—but the reduction in protocol processor occupancy from Fixed is not high enough to overcome the cost. Even at higher compute-utilizations (*U=0.7*), the relative cost-

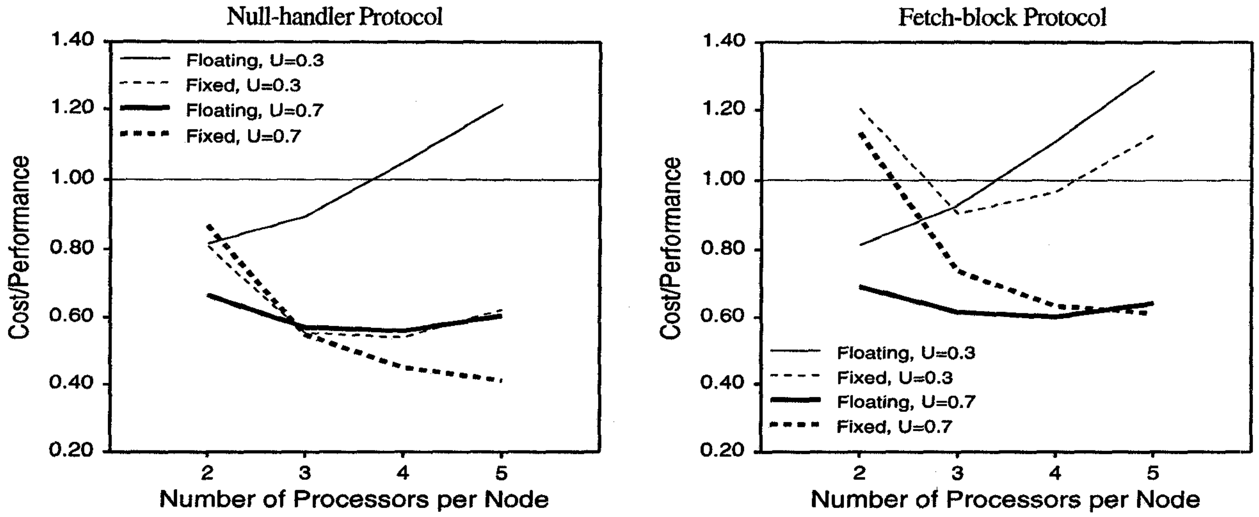

FIGURE 5. Cost-performance of Fixed and Floating with varying number of processors

The figures plot cost-performance of Fixed and Floating (O_{int} = 200 cycles) varying the number of processors per node for two values of compute-utilization (*U=0.3* and *U=0.7*). Costups and speedups are calculated with respect to a uniprocessor node (Single). The graphs assume that the cost of a processor is 30% of the cost of a uniprocessor node. Values over the horizontal lines indicate design points that are not cost-effective.

increment is high enough to prevent Fixed from improving cost-effectiveness over Floating.

6 Macrobenchmark Analysis

Although microbenchmark analysis helps develop intuition about relative performance, it makes many simplifying assumptions. For example, our experiments ignored synchronization, burstiness of communication, cache effects due to large data sets, and bandwidth limitations of the memory bus. In this section, we re-examine our policies in the context of a network of eight multiprocessor workstations, each with five processors.

Name	Input Data Set
appbt	24 x 24 x 24 cubes, 12 iters
barnes	8192 bodies
em3d	38400 nodes, 15% remote, 40 iterations
gauss	960 x 960 matrix
tomcatv	960 x 960 matrices, 10 iterations

TABLE 1. Application input parameters

Table 1 lists the applications and corresponding input data sets we use in this study. *Appbt* is a three-dimensional fluid dynamics application [7]. *Barnes* is an N-body simulation from the SPLASH-2 suite [29]. *Em3d* models the propagation of electromagnetic waves in three dimensions [5]. *Gauss* solves a linear system of equations using Gaussian elimination [3]. *Tomcatv* is a parallel version of the SPEC benchmark.

Our transparent distributed shared-memory system uses a 128-byte Stache [23] protocol to keep data coherent between nodes; intra-node communication occurs through the MOESI coherence protocol on the bus. We measure a minimum running time for the request handler to be 125 cycles, and reply and response handlers to be 140 cycles for a total of 900 cycles (4.5μs) of round-trip latency.

6.1 Baseline System

Figure 6 compares the performance of Fixed and Floating with varying number of processors per node. Except for *em3d*, adding a dedicated protocol processor to a uniprocessor node improves performance by at most 25%. *Em3d* is our most communication-intensive application with a compute-utilization of less than 50%. The application iterates over a bipartite graph, computing new values for each graph node. Fetching remote node values dominates the running time of an iteration. Eliminating interrupt overhead allows Fixed to improve performance by 63%.

Using the second processor for computation—under the Floating policy—improves performance by 54%-98% in all the applications. *Appbt*, *barnes*, *gauss* and *tomcatv* all have moderate to high compute-utilizations and can take advantage of the second compute processor. In *em3d*, the second processor both contributes to computation and alternates with the other processor to act as protocol processor.

As we increase the number of processors per node, we increase both computational resources and demand for protocol processing. *Tomcatv* is our most compute-bound application and primarily benefits from addition of compute

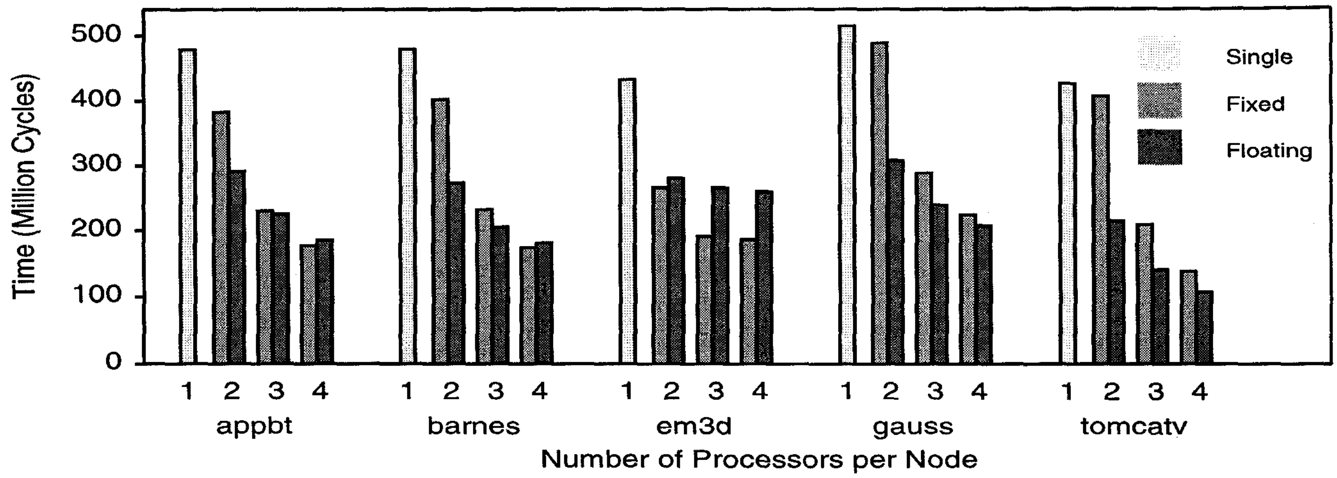

FIGURE 6. Performance of Fixed and Floating with varying number of processors per node

processors. Compute-utilizations in *appbt* and *barnes* are at moderate levels (≈ 70%). Protocol processing in these applications begins to dominate running time with three or more processors per node. The dedicated protocol processor in Fixed reduces occupancy and improves performance over Floating at four processors per node.

Gauss also exhibits a moderate level compute-utilization. Because communication in *gauss* is synchronous, an idle processor remains the acting protocol processor during the communication phase. As a result, Floating mimics the behavior of Fixed and stays competitive at four processors per node.

Em3d, with its low compute-utilization manages to saturate the protocol processor with only two (compute) processors. Although two or more processors virtually eliminate all the interrupts, at saturation point the indirect overhead of a floating protocol processor limits the performance under Floating to 70% of that under Fixed.

6.2 Interrupt Overhead

The performance of Floating (Single) is sensitive to how quickly the system can interrupt a processor and dispatch a protocol handler. Today's commercial operating systems do not provide fast delivery of user-level interrupts. Exception handling on these systems can take up to 200 μs [26], one to two orders of magnitude longer than that on some carefully tuned parallel computers [22]. In this experiment we study the sensitivity of the policy trade-off to interrupt overheads.

Table 2 presents execution times of Single and two-processor Floating, normalized to two-processor Fixed for three values of interrupt overhead. As predicted by our microbenchmark analysis, very high interrupt overheads severely impact the performance of Single. Increasing interrupt overhead by two orders of magnitude can increase the running time of Single by over 800%. This result corroborates the observation that with stock operating systems, networks of workstations (NOWs) [1] may have to rely on program instrumentation [27,14] to perform periodic polling.

Application	Interrupt Overhead		
	1 μs	10 μs	100 μs
Single/Fixed			
appbt	1.25	1.78	5.14
barnes	1.19	1.51	3.63
em3d	1.63	2.32	8.76
gauss	1.05	1.18	2.57
tomcatv	1.05	1.18	1.88
Floating/Fixed			
appbt	0.76	0.94	**1.95**
barnes	0.68	0.76	**1.04**
em3d	**1.06**	**1.07**	**1.22**
gauss	0.63	0.66	0.83
tomcatv	0.53	0.54	0.65

TABLE 2. Sensitivity to interrupt overhead
The Table presents execution times of Single and two-processor Floating normalized to two-processor Fixed for various interrupt overheads. Numbers appearing in boldface indicate points where Fixed outperforms Floating.

Another observation, consistent with our microbenchmark results, is that very high interrupt overheads have a much smaller impact on the performance of Floating than Single. In all the applications, a two orders of magnitude increase in interrupt overhead slows the program down by at most 160%. This is because an idle processor acting as protocol processor eliminates many of the interrupts. High interrupt overhead has the largest impact on *appbt,* because this application uses spin locks to synchronize threads in a gaussian elimination phase. As such, an idle processor spinning on a lock takes an interrupt upon arrival of every message.

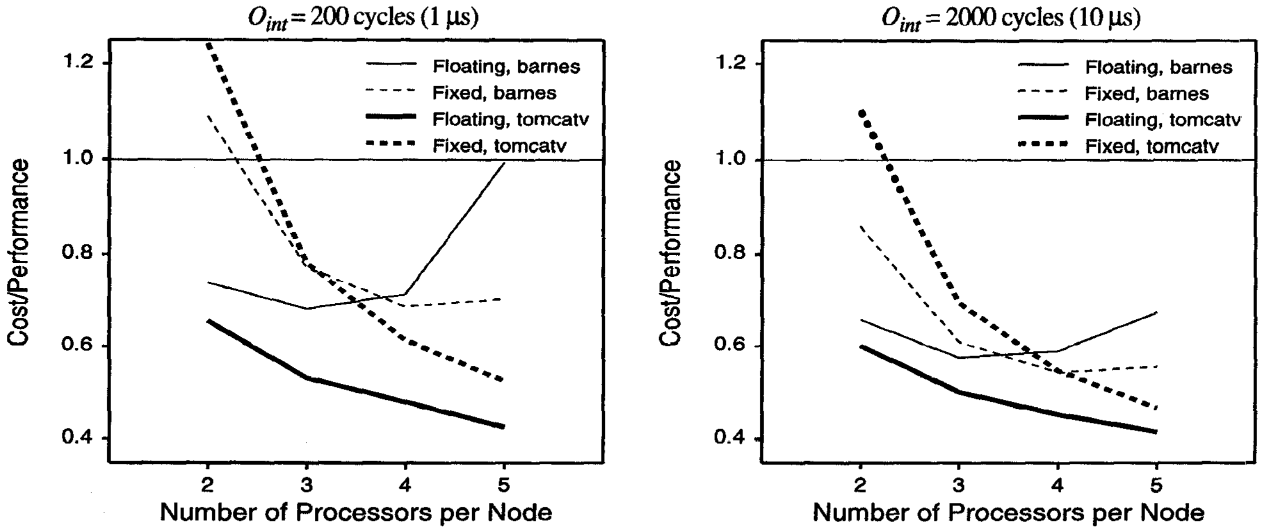

FIGURE 7. Cost-performance of Fixed and Floating with varying number of processors
The figures plot cost-performance of Fixed and Floating varying the number of processors per node for *barnes* and *tomcatv.* Costups and speedups are calculated with respect to a uniprocessor node (Single). The graphs assume that the cost of a processor is 30% of the cost of a node. Values over the horizontal lines indicate design points that are not cost-effective.

6.3 Cost/Performance

Figure 7 plots cost-performance for two applications with moderate (*barnes*) to high (*tomcatv*) compute-utilizations versus the number of processors. The graphs indicate that adding a dedicated protocol processor to a uniprocessor node is never cost-effective for the lower interrupt overhead (left). This is not surprising since performance improves by at most 20% whereas the system cost goes up by 30%. When overhead is high (right), performance in *barnes* improves by 50% justifying the cost of the dedicated protocol processor. Computation in *tomcatv* remains the dominant factor in the running time. Even with higher interrupt overhead the program benefits little from a dedicated protocol processor. A second compute processor, however, improves performance in the two applications by at least 70% and is therefore cost-effective.

Much as our microbenchmarks predicted, when interrupt overhead is low—as compared to protocol weight—the system is most cost-effective under the Floating policy; for *barnes*, cost-performance under Fixed reaches a minimum close to, but not the same as, that under Floating. *Tomcatv* speeds up linearly and therefore always reaches a lower cost-performance under Floating. When the number of processors is large enough (> 6), speedup dominates cost-performance in *tomcatv* causing it to eventually level off. At this point, Floating results in a marginal improvement in cost-performance over Fixed.

High interrupt overhead, however, changes the balance. *Barnes* achieves a minimum cost-performance under the Fixed policy. The high overhead increases protocol processor occupancy, resulting in a higher protocol processing to running time ratio. The Fixed policy reduces protocol processor occupancy, allowing the protocol processor to accommodate a larger number of processor before protocol processing saturates. At this point, the performance improvement due to a dedicated protocol processor is large enough to offset its incremental cost. Floating remains most cost-effective for the more compute-intensive application, *tomcatv.* High interrupt overhead, however, slightly closes the gap in cost-performance between to the two policies for this application.

7 Summary and Conclusions

In this paper, we examined how protocol processing should be scheduled on an SMP node parallel machine. Previous systems such as the Intel Paragon have dedicated a processor specifically for protocol processing. Others have recently argued that all processors should be used for both computation and communication [12,6]. We examined when it does and does not make sense to dedicate a protocol processor.

We presented results from synthetic benchmarks for two general request/reply protocols to illustrate the trade-offs between the policies. The results showed that: i) a dedicated protocol processor benefits light-weight protocols much more than heavy-weight protocols; ii) Fixed improves performance over Floating when communication becomes the bottleneck, which is more likely when the application is very communication-intensive, protocol overheads are very high, or there are multiple (i.e., more than two) processors per node; iii) a system with optimal cost-effectiveness is likely to include a dedicated protocol processor when overheads are a significant component of protocol processor occupancy.

Finally, we evaluated these policies in the context of a fine-grain user-level distributed shared-memory system. We presented results from simulating a network of eight multiprocessor workstations—each with up to five processors—running five shared-memory applications using a software coherence protocol. Besides corroborating our findings from the first experiment, the results also show that bursty and synchronous communication patterns in some applications reduce overhead and therefore decrease the benefit of the Fixed policy.

Acknowledgements

We would like to thank Doug Burger, Jim Goodman, Mark Hill, Stefanos Kaxiras, and Anne Rogers for their helpful comments on earlier drafts of this paper.

References

[1] Tom Anderson, David Culler, and David Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[3] 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*, pages 61–73, San Jose, California, 1994.

[4] Derek Chiou, Boon S. Ang, Arvind, Michael J. Beckerle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StarT-NG: Seamless Parallel Computing. In *Proceedings of EURO-PAR'95*, 1995.

[5] 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.

[6] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[7] 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.

[8] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[9] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[10] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

[11] Vijay Karamcheti and Andrew A. Chien. Software Overhead in Message Layer: Where Does the Time Go? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, 1994.

[12] Magnus Karlsson and Per Stenstrom. Performance Evaluation of a Cluster-Based Multiprocessor Build from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

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

[14] 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)*, pages 291–300, June 1995.

[15] Beng-Hong Lim, Phillip Heidelberger, Pratap Pattanik, and Marc Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, February 1997.

[16] Tom Lovett and Russel Clapp. STiNG: A CC-NUMA Compute System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

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

[18] 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.

[19] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[20] Steven Reinhardt, Robert Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[21] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[22] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

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

[24] James D. Salehi, James F. Kurose, and Don Towsley. Scheduling for cache affinity in parallelized communication protocols. Technical Report UM-CS-1994-075, University of Masschusetts, October 1994.

[25] 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 (ASPLOS VI)*, pages 297–307, October 1994.

[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] 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.

[28] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. LU Factorization and the LINPACK benchmark on the Intel Paragon. Technical Report ftp://ftp.cs.sandia.gov/pub/papers/dewombl/paragon_linpack_benchmark.ps, Sandia National Laboratories, March 1994.

[29] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

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