

# Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors\*

Maged M. Michael<sup>†</sup>, Ashwini K. Nanda<sup>‡</sup>, Beng-Hong Lim<sup>‡</sup>, and Michael L. Scott<sup>†</sup>

<sup>†</sup>University of Rochester  
Department of Computer Science  
Rochester, NY 14627  
{michael,scott}@cs.rochester.edu

<sup>‡</sup>IBM Research  
Thomas J. Watson Research Center  
Yorktown Heights, NY 10598  
{ashwini,bhlim}@watson.ibm.com

## Abstract

Scalable distributed shared-memory architectures rely on coherence controllers on each processing node to synthesize cache-coherent shared memory across the entire machine. The coherence controllers execute coherence protocol handlers that may be hardwired in custom hardware or programmed in a protocol processor within each coherence controller. Although custom hardware runs faster, a protocol processor allows the coherence protocol to be tailored to specific application needs and may shorten hardware development time. Previous research show that the increase in application execution time due to protocol processors over custom hardware is minimal.

With the advent of SMP nodes and faster processors and networks, the tradeoff between custom hardware and protocol processors needs to be reexamined. This paper studies the performance of custom-hardware and protocol-processor-based coherence controllers in SMP-node-based CC-NUMA systems on applications from the SPLASH-2 suite. Using realistic parameters and detailed models of existing state-of-the-art system components, it shows that the occupancy of coherence controllers can limit the performance of applications with high communication requirements, where the execution time using protocol processors can be twice as long as using custom hardware.

To gain a deeper understanding of the tradeoff, we investigate the effect of varying several architectural parameters that influence the communication characteristics of the applications and the underlying system on coherence controller performance. We identify measures of applications' communication requirements and their impact on the performance penalty of protocol processors, which can help system designers predict performance penalties for other applications. We also study the potential of improving the performance of hardware-based and protocol-processor-based coherence controllers by separating or duplicating critical components.

\*This work was supported and performed at IBM Thomas J. Watson Research Center. Michael Scott was supported in part by NSF grants (CDA-9401142 and CCR-9319445).

To appear in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), Denver, CO, June 1997.

## 1 Introduction

Previous research has shown convincingly that scalable shared-memory performance can be achieved on directory-based cache-coherent multiprocessors such as the Stanford DASH [6] and MIT Alewife [1] machines. A key component of these machines is the coherence controller on each node that provides cache coherent access to memory that is distributed among the nodes of the multiprocessor. In DASH and Alewife, the cache coherence protocol is hardwired in custom hardware finite state machines (FSMs) within the coherence controllers. Instead of hardwiring protocol handlers, the Sun Microsystems S3.mp [10] multiprocessor uses hardware sequencers for modularity in implementing protocol handlers.

Subsequent designs for scalable shared-memory multiprocessors, such as the Stanford FLASH [5] and the Wisconsin Typhoon machines [12], have touted the use of programmable protocol processors instead of custom hardware FSMs to implement the coherence protocols. Although a custom hardware design generally yields better performance than a protocol processor for a particular coherence protocol, the programmable nature of a protocol processor allows one to tailor the cache coherence protocol to the application [2, 8], and may lead to shorter design times since protocol errors may be fixed in software. The study of the performance advantage of custom protocols is beyond the scope of this paper.

Performance simulations of the Stanford FLASH and Wisconsin Typhoon systems find that the performance penalty of protocol processors is small. Simulations of the Stanford FLASH, which uses a customized protocol processor optimized for handling coherence actions, show that the performance penalty of its protocol processor in comparison to custom hardware controllers is within 12% for most of their benchmarks [3]. Simulations of the Wisconsin Typhoon Simple-COMA system, which uses a protocol processor integrated with the other components of the coherence controller, also show competitive performance that is within 30% of custom-hardware CC-NUMA controllers [12] and within 20% of custom-hardware Simple-COMA controllers [13].

Even so, the choice between custom hardware and protocol processors for implementing coherence protocols remains a key design issue for scalable shared-memory multiprocessors. The goal of this research is to examine in detail the performance tradeoffs between these two alternatives in designing a CC-NUMA multiprocessor coherence controller. We consider symmetric multiprocessor (SMP) nodes as well as uniprocessor nodes as the building block for a multiprocessor. The availability of cost-effective SMPs, such as those based on the Intel Pentium Pro [11] makes SMP nodes an attractive choice for CC-NUMA designers [7]. However, the added load presented to the coherence controller by multiple SMP processors may affect the choice between custom hardware FSMs and protocol processors.

We base our experimental evaluation of the alternative coherence controller architectures on realistic hardware parameters for state-of-the-art system components. What distinguishes our work from previous research is that we consider commodity protocol processors on SMP-based CC-NUMA and a wider range of architectural parameters. We simulate eight applications from the SPLASH-2 benchmark suite [14] to compare the application performance of the architectures. The results show that for a 64-processor system based on four-processor SMP nodes, protocol processors result in a performance penalty (increase in execution time relative to that of custom hardware controllers) of 4% – 93%.

The unexpectedly high penalty of protocol processors occurs for applications that have high-bandwidth communication requirements, such as Ocean, Radix, and FFT. The use of SMP nodes exacerbates the penalty. Previous research did not encounter such high penalties because they were either comparing customized protocol processors in uniprocessor nodes, or they did not consider such high-bandwidth applications. We find that under high-bandwidth requirements, the high occupancy of the protocol processor significantly degrades performance relative to custom hardware.

We also study the performance of coherence controllers with two protocol engines. Our results show that for applications with high communication requirements on a 4×16 CC-NUMA system, a two-engine hardware controller improves performance by up to 18% over a one-engine hardware controller, and a controller with two protocol processors improves performance by up to 30% over a controller with a single protocol processor

This paper makes the following contributions:

- It provides an in-depth comparison of the performance trade-offs between using custom hardware and protocol processors, and demonstrates situations where protocol processors suffer a significant penalty.
- It characterizes the communication requirements for eight applications from SPLASH-2 and shows their impact on the performance penalty of protocol processors over custom hardware. This provides an understanding of application requirements and limitations of protocol processors.
- It evaluates the performance gains of using two protocol engines for custom hardware and protocol-processor-based coherence controllers.
- It introduces a methodology for predicting the impact of protocol engine implementation on the performance of important large applications through the detailed simulation of simpler applications.

The rest of this paper is organized as follows. Section 2 presents the multiprocessor system and details the controller design alternatives and parameters. Section 3 describes our experimental methodology and presents the experimental results. It demonstrates the performance tradeoffs and provides analysis of the causes of the performance differences between the architectures. Section 4 discusses related work. Finally, Section 5 presents the conclusions drawn from this research and gives recommendations for custom hardware and protocol processor designs in future multiprocessors.

## 2 System Description

To put our results in the context of the architectures we studied, this section details these architectures and their key parameters. First we describe the organization and the key parameters of the common system components for the architectures. Then, we describe the details of the alternative coherence controller architectures. Finally, we present key protocol and coherence controller latencies and occupancies.

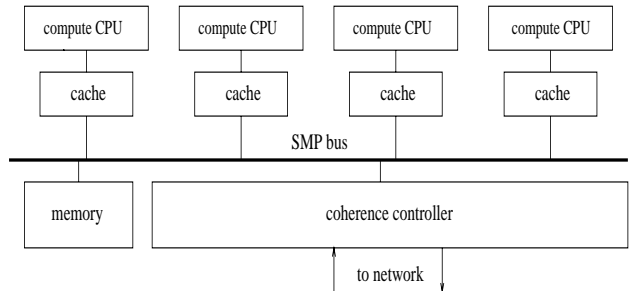


Figure 1: A node in a SMP-based CC-NUMA system.

Event	Latency
L1 to processor	1
L1 to L2	8
L2 to L1	4
L2 miss to address strobe on bus	4
Bus address strobe to bus response	14
Bus address strobe to start of cache-to-cache data response	18
Bus address strobe to next address strobe	4
Bus address strobe to start of data transfer from memory	20
Network point-to-point	14

Table 1: Base system no-contention latencies in compute processor cycles (5 ns.).

### 2.1 General System Organization and Parameters

The base system configuration is a CC-NUMA multiprocessor composed of 16 SMP nodes connected by a 32 byte-wide fast state-of-the-art IBM switch. Each SMP node includes four 200 MHz PowerPC compute processors with 16 Kbyte L1 and 1 Mbyte L2 4-way-associative LRU caches, with 128 byte cache lines. The SMP bus is a 100 MHz 16 byte-wide fully-pipelined split-transaction separate-address-and-data bus. The memory is interleaved and the memory controller is a separate bus agent from the coherence controller. Figure 1 shows a block diagram of an SMP node. Table 1 shows the no-contention latencies of key system components. These latencies correspond to those of existing state-of-the-art components. Note that memory and cache-to-cache data transfers drive the critical quad-word first on the bus to minimize latency.

### 2.2 Coherence Controller Architectures

We consider two main coherence controller designs: a custom hardware coherence controller similar to that in the DASH [6] and Alewife [1] systems, and a coherence controller based on commodity protocol processors similar to those in the Typhoon [12] system and its prototypes [13].

The two designs share some common components and features (see Figures 2 and 3). Both designs use duplicate directories to allow fast response to common requests on the pipelined SMP bus (one directory lookup per 2 bus cycles). The bus-side copy is abbreviated (2-bit state per cache line) and uses fast SRAM memory. The controller-side copy is full-bit-map and uses DRAM memory. Both designs use write-through directory caches for reducing directory read latency. Each directory cache holds up to 8K full-bit-map directory entries (e.g. approximately 16 Kbytes for a 16 node CC-NUMA system). The hardware-based design uses a custom on-chip cache, while the protocol-processor-based design uses the

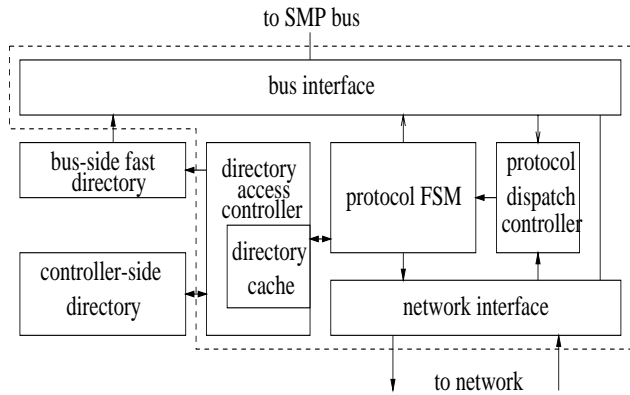


Figure 2: A custom-hardware-based coherence controller design (HWC).

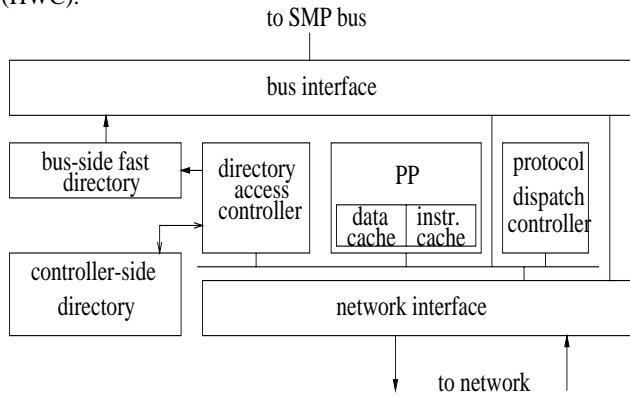


Figure 3: A commodity PP-based coherence controller design (PPC).

commodity processor’s on-chip data caches.<sup>1</sup> We assume perfect instruction caches in the protocol processors, as the total size of all protocol handlers in our protocol is less than 16 Kbytes.

Both designs include a custom directory access controller for keeping the bus-side copy of the directory consistent with the controller-side copy, and a custom protocol dispatch controller for arbitration between the request queues from the local bus and the network. There are 3 input queues for protocol requests: bus-side requests, network-side requests, and network-side responses. The arbitration strategy between these queues is to let the network transaction nearest to completion be handled first. Thus, the arbitration policy is that network-side responses have the highest priority, then network-side requests, and finally bus-side requests. In order to avoid live-lock, the only exception to this policy is to allow bus-side requests which have been waiting for a long time (e.g. four subsequent network-side requests) to proceed before handling any more network-side requests.

Figure 2 shows a block diagram of a custom hardware coherence controller design (HWC). The controller runs at 100 MHz, the same frequency as the SMP bus. All the coherence controller components are on the same chip except the directories. Figure 3 shows a block diagram of a protocol-processor-based coherence controller (PPC). The protocol processor (PP) is a PowerPC running at 200 MHz. The other controller components run at 100 MHz. The protocol processor communicates with the other components of the controller through loads and stores on the local (coherence controller) bus to memory-mapped off-chip registers in the other components. The protocol processor access to the protocol dispatch controller

<sup>1</sup>Although most processors use write-back caches, current processors (e.g. Pentium Pro [11]) allow users to designate regions of memory to be cached write-through.

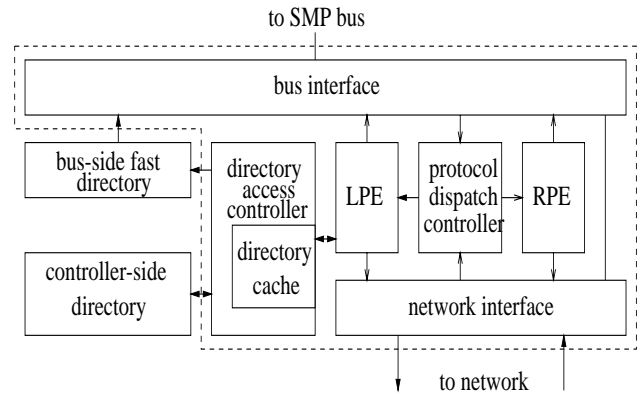


Figure 4: A custom hardware coherence controller design with local and remote protocol FSMs (2HWC).

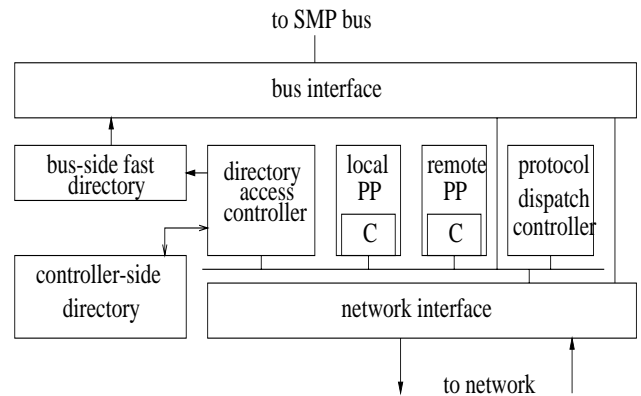


Figure 5: A commodity PP-based coherence controller design with local and remote protocol processors (2PPC).

register is read-only. Its access to the network interface registers is write-only (for sending network messages and starting direct data transfer from the bus interface), since reading the headers of incoming network messages is performed only by the protocol dispatch controller.

Both HWC and PPC have a direct data path between the bus interface and the network interface. The direct data path is used to forward write-backs of dirty remote data from the SMP bus directly to the network interface to be sent to the home node without waiting for protocol handler dispatch. Also, in the case of PPC, the PP only needs to perform a single write to a special register on either the bus interface or the network interface to invoke direct data transfer, without the need for the PP to read and write the data to perform the transfer.

In order to increase the bandwidth of the coherence controller we also consider the use of two protocol processors in the PPC implementation and two protocol FSMs in the HWC implementation. We use the term “protocol engine” to refer to both the protocol processor in the PPC design and the protocol FSM in the HWC design. For distributing the protocol requests between the two engines, we use a policy similar to that used in the S3.mp system [10], where protocol requests for memory addresses on the local node are handled by one protocol engine (LPE) and protocol requests for memory addresses on remote nodes are handled by the other protocol engine (RPE). Only the LPE needs to access the directory. Figures 4 and 5 show the two-engine HWC design (2HWC), and the two PP controller design (2PPC), respectively.

Sub-operation	HWC	PPC
Issue request to bus	2	8
Detect response from bus	2	8
Issue network message	2	9
Read special bus interface associative registers	4	10
Write special bus interface registers	2	4
Directory read (cache hit)	2	2
Directory read (cache miss)	22	22
Directory write	2	2
Handler dispatch	2	12
Condition	2	2
Loop (per iteration)	2	5
Clear bit field	-	3
Extract bit field	-	2
Other bit operations	-	1

Table 2: Protocol engine sub-operation occupancies for HWC and PPC in compute processor cycles (5 ns).

Step	HWC	PPC
detect L2 miss	8	8
issue bus read request	4	4
bus response	14	14
dispatch handler	2	12
extract home id	-	2
send message to home node	2	9
network latency	14	14
dispatch handler	2	12
read directory entry (cache hit)	2	2
conditions	2	6
issue bus read request	6	12
memory latency	20	20
detect bus response	2	8
extract requester's id	-	2
send message to the requester	2	9
network latency	14	14
dispatch handler	2	12
issue response to bus	6	12
L2 reissues read request	18	18
bus response	14	14
bus interface issues data	4	4
L1 fill	4	4
total	142	212

Table 3: Breakdown of the no-contention latency of a read miss to a remote line clean at home in compute processor cycles (5 ns).

### 2.3 Controller Latencies and Occupancies

We modeled HWC and PPC accurately with realistic parameters. Table 2 lists protocol engine sub-operations and their occupancies<sup>2</sup> for each of the HWC and PPC coherence controller designs, assuming a 100 MHz HWC and a 100 MHz PPC with a 200 MHz off-the-shelf protocol processor. The occupancies in the table assume no contention on the SMP bus, memory, and network, and all directory reads hit in the protocol engine data cache. The other assumptions used in deriving these numbers are:

- Accesses to on-chip registers for HWC take one system cycle

<sup>2</sup>Occupancy of sub-operations is the time a protocol engine is occupied by the sub-operation and cannot service other requests.

(2 CPU cycles).

- Bit operations on HWC are combined with other actions, such as conditions and accesses to special registers.
- PP reads to off-chip registers on the local PPC bus take 4 system cycles (8 CPU cycles). Searching a set of associative registers takes an extra system cycle (2 CPU cycles).
- PP writes to off-chip registers on the local PPC bus take 2 system cycles (4 CPU cycles) before the PP can proceed.
- PP compute cycles are based on the PowerPC instruction cycle counts produced by the IBM XLC C compiler.
- HWC can decide multiple conditions in one-cycle.

To gain insight into the effect of these occupancies and delays on the latency of a typical remote memory transaction, Table 3 presents the no-contention latency breakdown of a read miss from a remote node to a clean shared line at the home node. The relative increase in latency from HWC to PPC is only 49%, which is consistent with the 33% increase reported for Typhoon [13], taking into account that we consider a more decoupled coherence controller design and we use a faster network than that used in the Typhoon study. It is worth noting that in Table 3 there is no entry for updating the directory state at the home node. The reason is that updating the directory state can be performed after sending the response from the home node, thus minimizing the read miss latency. In our protocol handlers, we postpone any protocol operations that are not essential for responding to requests until after issuing responses.

Finally, in order to gain insight into the relative occupancy times of the HWC and PPC coherence controller designs, Table 4 presents the no-contention protocol handler occupancies for the most frequently used handlers. Handler occupancy times include: handler dispatch time, directory reference time, access time to special registers, SMP bus and local memory access times, and bit field manipulation for PPC. Note that memory is sequentially consistent, and that we use the same full-map directory, invalidation-based, write-back protocol, for both HWC and PPC. In our protocol, we allow remote owners to respond directly to remote requesters with data, but invalidation acknowledgments are collected only at the home node.

## 3 Experimental Results

In this section, we present simulation results of the relative performance of the different coherence controller architectures with several variations of communication-related architectural parameters. Then, we present analysis of the key statistics and communication measures collected from these simulations, and we conclude this section by presenting statistics and analysis of the utilization and workload distribution on two-protocol-engine coherence controllers. We start with the experimental methodology.

### 3.1 Experimental Methodology

We use execution-driven simulation (based on a version of the Augmint simulation toolkit [9] that runs on the PowerPC architecture) to evaluate the performance of the four coherence controller designs, HWC, PPC, 2HWC, and 2PPC. Our simulator includes detailed contention models for SMP buses, memory controllers, interleaved memory banks, protocol engines, directory DRAM, and external point contention for the interconnection network. Protocol handlers are simulated at the granularity of the sub-operations in Table 2, in addition to accurate timing of the interaction between the coherence controller and the SMP bus, memory, directory, and

Handler	HWC	PPC
bus read remote	4	23
bus read exclusive remote	4	23
bus read local (dirty remote)	10	33
bus read excl. local (cached remote)	10 + 4/inv.	32 + 16/inv.
remote read to home (clean)	38	73
remote read to home (dirty remote)	10	29
remote read excl. to home (uncached remote)	38	73
remote read excl. to home (shared remote)	10 + 4/inv.	32 + 16/inv.
remote read excl. to home (dirty remote)	10	30
read from remote owner (request from home)	32	81
read from remote owner (remote requester)	34	90
read excl. from remote owner (request from home)	32	81
read excl. from remote owner (remote requester)	34	90
data response from owner to a read request from home	8	21
write back from owner to home in response to a read req. from remote node	8	24
data response from owner to a read excl. request from home	6	16
ack. from owner to home in response to a read excl. request from remote node	4	17
invalidation request from home to sharer	26	49
inv. acknowledgment (more expected)	8	23
inv. ack. (last ack, local request)	10	33
inv. ack. (last ack, remote request)	36	75
data in response to a remote read request	4	16
data in response to a remote read excl. request	6	20

Table 4: Protocol engine occupancies in compute processor cycles (5 ns.).

network interface. All coherence controller implementations use the same cache coherence protocol.

We use eight benchmarks from the SPLASH-2 suite [14], (Table 5) to evaluate the performance of the four coherence controller implementations. All the benchmarks are written in C and compiled using IBM XLC C compiler with optimization level -O2. All experimental results reported in this paper are for the parallel phase only of these applications. We use a round-robin page placement policy except for FFT where we use an optimized version with programmer hints for optimal page placement. We observed slightly inferior performance for most applications when we used a first-touch-after-initialization page placement policy, due to load imbalance, and memory and coherence controller contention as a result of uneven memory distribution. LU and Cholesky are run on 32-

Application	Type	Problem size
LU	Blocked dense linear algebra	512×512 matrix, 16x16 blocks
Water-Spatial	Study of forces and potentials of water molecules in a 3-D grid	512 molecules
Barnes	Hierarchical N-body	8K particles
Cholesky	Blocked sparse linear algebra	tk15.O
Water-Nsquared	$O(n^2)$ study of forces and potentials in water molecules	512 molecules
Radix	Radix sort	1M integer keys, radix 1K
FFT	FFT computation	64K complex doubles
Ocean	Study of ocean movements	258×258 ocean grid

Table 5: Benchmark types and data sets.

processor systems (8 nodes × 4 processors each) as they suffer from load imbalance on 64 processors with the data sets used [14]. We ran all the applications with data sizes and systems sizes for which they achieve acceptable speedups.

### 3.2 Performance Results

In order to capture the main factors influencing PP performance penalty (the increase in execution time on PPC relative to the execution time on HWC), we ran experiments on the base system configuration with the four coherence controller architectures. We then varied some key system parameters to investigate their effect on the PP performance penalty.

#### Base Case

Figure 6 shows the execution times for the four coherence controller architectures on the base system configuration normalized by the execution time of HWC. We notice that the PP penalty can be as high as 93% for Ocean and 52% for Radix, and as low as 4% for LU. The significant PP penalties for Ocean, Radix and FFT indicate that commodity PP-based coherence controllers can be the bottleneck when running communication-intensive applications. This result is in contrast to the results of previous research, which showed the cases where custom PP-based coherence controllers suffer small performance penalties relative to custom hardware.

Also, we observe that for applications with high bandwidth requirements, using two protocol engines improves performance significantly relative to the corresponding single engine implementation, up to 18% on HWC and 30% on PPC, for Ocean.

We varied other system and application parameters that are expected to have a big impact on the communication requirements of the applications. We start with the cache line size.

#### Smaller cache line size

With 32 byte cache lines, we expect the PP penalty to increase from that experienced with 128 byte cache lines, especially for applications with high spatial locality, due to the increase in the rate of requests to the coherence controller. Figure 7 shows the execution

times normalized to the execution time of HWC on the base configuration. We notice a significant increase in execution time (regardless of the coherence controller architecture) relative to the corresponding execution times on the base system, for FFT, Cholesky, Radix, and LU, which have high spatial locality [14], and a minor increase in execution time for the other benchmarks.

Also, we notice a significant increase in the PP penalty (compared to the PP penalty on the base system) for applications with high spatial locality, due to the increase in the number of requests to the coherence controllers, which increases the demand on PP occupancy. For example, the PP penalty for FFT increases from 45% to 68%.

### Slower network

To determine the impact of network speed on the PP performance penalty, we simulated the four applications with the largest PP penalties on a system with a slow network (1  $\mu$ s. latency). Figure 8 shows the execution times normalized to the execution time of HWC on the base configuration. We notice a significant decrease in the PP penalty from that for the base system. The PP penalty for Ocean drops from 93% to 28%. Accordingly, systems designs with slow networks can afford to use commodity protocol processors instead of custom hardware, without significant impact on performance, when cache line size is large.

Also, we notice a significant increase in execution time (regardless of the coherence controller architecture) relative to the corresponding execution times on the base system, for Ocean and Radix, due to their high communication rates.

### Larger data size

To determine the effect of data size on the PP penalty, we simulated Ocean and FFT on the base system with larger data sizes, 256K complex doubles for FFT, and a 514x514 grid for Ocean. Figure 9 shows the execution times normalized by the execution time of HWC for each data size. We notice a decrease in the PP penalty in comparison to the penalty with the base data sizes, since the communication-to-computation ratios for Ocean and FFT decrease with the increase of the data size<sup>3</sup>. The PP penalty for FFT drops from 46% to 33%, and for Ocean from 93% to 67%.

However, since communication rates for applications like Ocean increase with the number of processors at the same rate that they decrease with larger data sizes, we can think of high PP performance penalties as limiting the scalability of such applications on systems with commodity PP-based coherence controllers.

### Number of processors per SMP node

Varying the number of processors per SMP node (i.e. per coherence controller), proportionally varies the demand on the coherence controller occupancy, and thus is expected to impact the PP performance penalty. Figure 10 shows the execution times on 64-processor systems (32 for LU and Cholesky) with 1, 2, 4, and 8 processors per SMP node, normalized to the execution time of HWC on the base configuration (4 processors/node).

We notice that for applications with low communication rates, the increase in the number of processors per node has only a minor effect on the PP performance penalty. For applications with high communication rates, the increase in the number of processors increases the PP performance penalty (e.g. the PP penalty increases from 93% for Ocean on 4 processors per node to 106% on 8 processors per node). However, the PP penalty can be as high as 79% (for Ocean) even on systems with one processor per node.

<sup>3</sup>Applications like Radix maintain a constant communication rate with different data sizes [14].

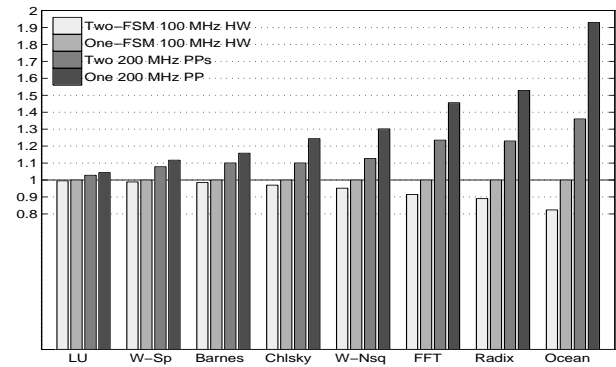


Figure 6: Normalized execution time on the base system configuration.

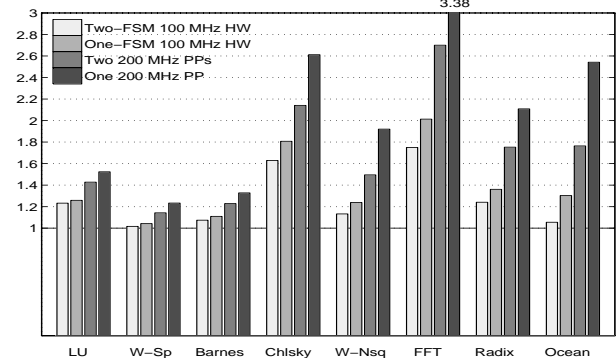


Figure 7: Normalized execution time for system with small (32 byte) cache lines.

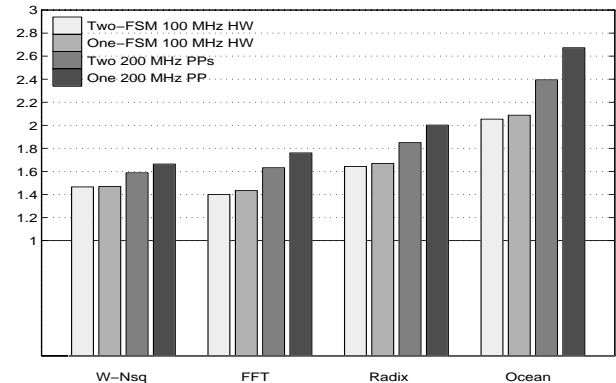


Figure 8: Normalized execution time for system with high (1  $\mu$ s.) network latency.

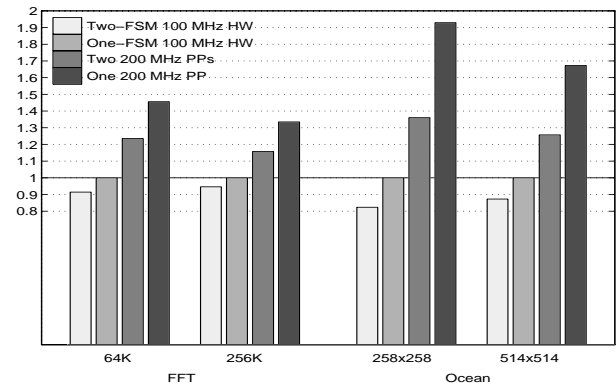


Figure 9: Normalized execution time for base system with base and large data sizes.

For each of the coherence controller architectures, performance of applications with high communication rates degrades with more processors per node, due to the increase in occupancy per coherence controller, which are already critical resources on systems with fewer processors per node.

Also, we observe that for applications with high communication rates (except FFT), the use of two protocol engines in the coherence controllers achieves similar or better performance than controllers with one protocol engine with half the number of processors per nodes. In other words, using two protocol engines in the coherence controllers, allows integrating twice as many processors per SMP node, thus saving the cost of half the SMP buses, memory controllers, coherence controllers, and I/O controllers.

### 3.3 Communication Statistics and Measures

In order to gain more insight into quantifying the application characteristics that affect PP performance penalty, we present some of the statistics generated by our simulations. Table 6 shows communication statistics collected from simulations of HWC and PPC on the base system configuration (except that Cholesky and LU are run on 32 processors). The statistics are:

- PP penalty: The increase in the execution time of PPC relative to the execution time of HWC.
- RCCPI (Requests to Coherence Controller Per Instruction) is the total number of requests to the coherence controllers divided by the total number of instructions.
- The total of the occupancies of all coherence controllers for PPC divided by that for HWC.
- Average HWC (PPC) utilization is the average HWC (PPC) occupancy divided by execution time.
- Average HWC (PPC) queuing delay is the average time a request to the coherence controller waits in a queue while the controller is occupied by other requests.
- Arrival rate of requests to HWC (PPC) per  $\mu s$ . (200 CPU cycles) is derived from the reciprocal of the mean inter-arrival time of requests to each of the coherence controllers.

In Table 6 we notice that as RCCPI increases, the PP performance penalty increases proportionally except for Cholesky. In the case of Cholesky, the high load imbalance inflates the execution time with both HWC and PPC. Therefore, the PP penalty which is measured relative to the execution time with HWC is less than the PP penalty of other applications with similar RCCPI but with better load balance.

Also, as RCCPI increases, the arrival rate of requests to the coherence controller per cycle for PPC diverges from that of HWC, indicating that the PPC has saturated, and that the coherence controller is the bottleneck for the base system configuration. This is also supported by the observation of the high utilization rates of HWC with Ocean, and of PPC with Ocean, Radix, and FFT, indicating that the coherence controller has saturated these cases, and it is the main bottleneck.

However, we notice that the queuing delays do not increase proportionally with the increase in RCCPI, since the queuing effect of the coherence controller behaves like a negative feedback system where the increase in RCCPI (the input) increases the queuing delay in proportion to the difference between the queuing delay and a saturation value, thus limiting the increase in queuing delay. Note that the high queuing delay for FFT is attributed to its bursty communication pattern [14].

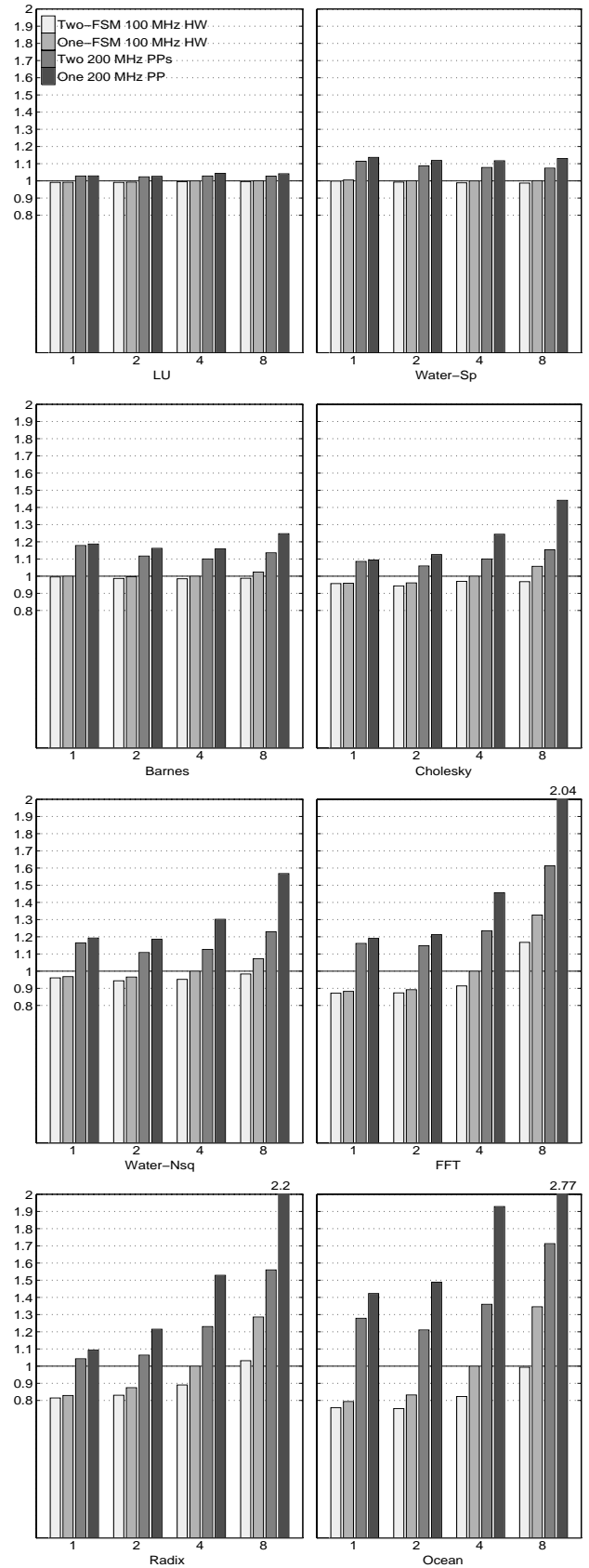


Figure 10: Normalized execution time with 1, 2, 4, and 8 processors per SMP node.

Application	PP Penalty	1000 × RCCPI	PPC/HWC occupancy	HWC utilization	PPC utilization	HWC queuing delay (ns.)	PPC queuing delay (ns.)	Average requests to HWC per $\mu$ s.	Average requests to PPC per $\mu$ s.
LU	4.37%	1.3	2.37	4.21%	9.58%	101	305	0.41	0.40
Water-Sp	11.69%	1.8	2.65	10.95%	25.99%	100	375	1.19	1.06
Barnes	15.81%	2.3	2.52	13.26%	28.88%	67	266	1.26	1.09
Cholesky	24.38%	4.1	2.23	26.38%	47.37%	113	365	2.34	1.86
Water-Nsq	30.15%	3.3	2.69	17.86%	36.87%	157	626	1.85	1.43
FFT-256K	33.44%	3.7	2.38	22.13%	39.54%	289	837	1.83	1.38
FFT-64K	45.59%	6.3	2.31	29.61%	46.96%	340	864	2.58	1.77
Radix	52.83%	9.8	2.36	36.82%	56.75%	229	640	3.66	2.33
Ocean-514	67.26%	14.0	2.29	47.54%	65.07%	226	648	3.87	2.31
Ocean-258	92.88%	23.2	2.47	52.89%	67.72%	232	720	4.69	2.41

Table 6: Communication statistics on the base system configuration.

Also, we observe that the ratio between the occupancy of PPC and the occupancy of HWC is more or less constant for the different applications, approximately 2.5.

Figure 11 plots the arrival rate of requests to each of the coherence controller architectures against RCCPI for all the applications on the base system configuration (except Cholesky and LU as they were run on 32 processors) including Ocean and FFT with large data sizes. The dotted lines show the trend for each architecture. The figure shows clearly the saturation levels of the different coherence controller architectures. The divergence in the arrival rates demonstrates that the coherence architecture is the performance bottleneck of the base system.

Figure 12 shows the effect of RCCPI on the PP penalty for the same experiments as those in Figure 11. We notice a clear proportional effect of RCCPI on the PP penalty. The gradual slope of the curve can be explained by the fact that the queuing model of the coherence controller resembles a negative feedback system. Without the negative feedback, the PP penalty would increase exponentially with the increase in RCCPI. The lower PP penalty for applications with low RCCPI such as Barnes and Water-Spatial is due to the fact that in those cases the coherence controller is under-utilized.

The previous analysis can help system designers predict the relative performance of alternative coherence controller designs. They can obtain the RCCPI measure for important large applications using simple simulators (e.g. PRAM) and relate that RCCPI to a graph similar to that in Figure 12 that can be obtained through the detailed simulation of simpler applications covering a range of communication rates similar to that of the large application. Although RCCPI is not necessarily independent of the implementation of the coherence controller, for practical purposes the effect of the architecture on RCCPI can be ignored. In our experiments the difference in RCCPI between the four implementations (HWC, PPC, 2HWC, and 2PPC) is less than 1% for all applications.

### 3.4 Utilization of Two-Engine Controllers

For coherence controller architectures with two protocol engines, there is more than one way to split the workload between the two protocol engines. In this study, we use a policy where protocol requests for memory addresses on the local node are handled by one protocol engine (LPE) and protocol requests for memory addresses on remote nodes are handled by the other protocol engine (RPE). In order to quantify the effectiveness of this policy, Table 7 shows the communication statistics collected from simulations of 2HWC and 2PPC on the base system configuration (except Cholesky and LU are run on 32 processors).

We observe that although most requests are handled by RPE

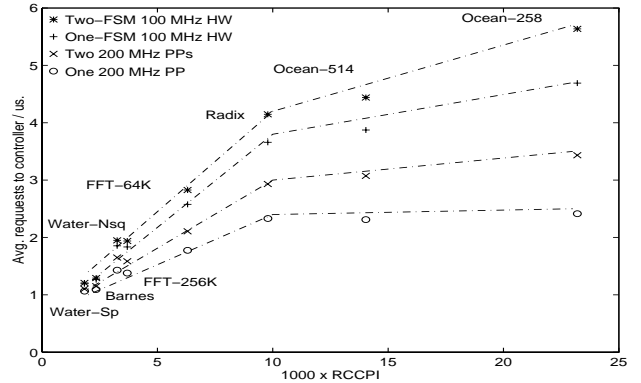


Figure 11: Coherence controller bandwidth limitations.

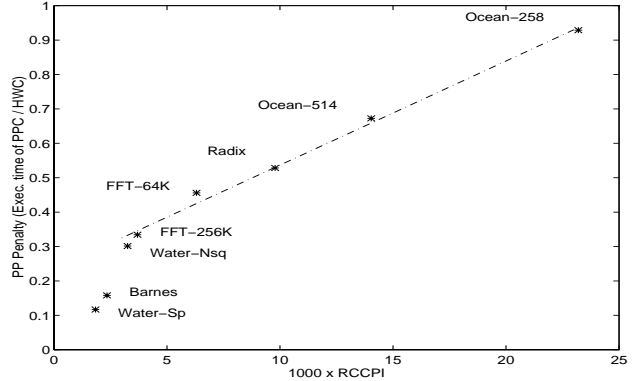


Figure 12: Effect of communication rate on PP penalty.

(53-63%), the occupancy of LPE is up to 3 times that of RPE for 2HWC, and up to 2 times for 2PPC (derived from the utilization numbers). This is because the average occupancy of protocol handlers performed on LPE are more than those on RPE, since the former are more likely to access the directory and main memory. Also, we observe that the sum of the average utilization numbers for LPE and RPE is more than the average utilization for the corresponding one-engine coherence controller (Table 6). This is due to the fact that the sum of the occupancies of LPE and RPE are almost the same as that for the one-engine controller, but the execution time decreases with the use of two protocol engines.

Due to the imbalance between the utilization figures of LPE and RPE, the queuing delays for RPE are lower than those for the corresponding one-engine controllers, while those for LPE are higher

Application	Architecture	Utilization		Request distribution		Queuing delay (ns.)	
		LPE	RPE	LPE	RPE	LPE	RPE
LU	2HWC	3.20%	1.09%	35.67%	64.33%	182	2
	2PPC	5.66%	3.92%	35.74%	64.26%	501	14
Water-Sp	2HWC	6.82%	4.29%	38.09%	61.91%	60	40
	2PPC	14.66%	12.38%	38.08%	61.92%	263	78
Barnes	2HWC	8.43%	5.22%	39.38%	60.62%	67	11
	2PPC	16.64%	13.85%	39.41%	60.59%	237	53
Cholesky	2HWC	20.26%	7.48%	38.27%	61.73%	128	8
	2PPC	30.34%	19.99%	38.27%	61.73%	348	36
Water-Nsq	2HWC	11.30%	7.89%	39.26%	60.74%	82	49
	2PPC	22.87%	19.81%	39.22%	60.78%	384	167
FFT-256K	2HWC	17.93%	5.92%	46.33%	53.67%	378	10
	2PPC	30.64%	15.05%	46.33%	53.67%	934	38
FFT-64K	2HWC	25.63%	7.45%	41.40%	58.60%	478	8
	2PPC	36.35%	19.17%	41.40%	58.60%	1137	39
Radix	2HWC	21.63%	21.32%	39.95%	60.05%	138	91
	2PPC	30.70%	40.86%	39.94%	60.06%	243	366
Ocean-514	2HWC	38.10%	18.33%	41.03%	58.97%	210	35
	2PPC	50.42%	36.59%	41.02%	58.98%	480	138
Ocean-258	2HWC	40.02%	25.97%	40.45%	59.55%	173	48
	2PPC	52.60%	44.19%	40.39%	59.61%	476	185

Table 7: Communication statistics for controllers with two protocol engines on the base system configuration.

for most applications despite the decrease in demand, due to the exclusion of the requests to RPE which typically have low occupancy requirements.

The large imbalance in the distribution of occupancy between LPE and RPE (derived from the utilization statistics) for most applications indicate that there is potential for further improvement in performance by using a more even policy for distributing the workload on the two (or possibly more) protocol engines. However, it is worth noting that in the design used in this paper, only one protocol engine, LPE, needs to access the directory. Furthermore, in the case of custom hardware, none of the handlers in the LPE FSM needs to be duplicated in the RPE FSM, and vice versa, thus minimizing the hardware overhead of two-engine HWC over one-engine HWC. Alternative distribution policies, such as splitting the workload dynamically or based on whether the request is from the local bus or another node, might lead to a more balanced distribution of protocol workloads on the protocol engines, but would also require allowing multiple protocol engines to access the directory, which increases the cost and complexity of coherence controllers.

## 4 Related Work

The proponents of protocol processors argue that the performance penalty of protocol processors is minimal, and that the additional flexibility is worth the performance penalty. The Stanford FLASH designers find that the performance penalty of using a protocol processor is less than 12% for the applications that they simulated, including Ocean and Radix [3]. Their measured penalties are significantly lower than ours for the following reasons: 1) FLASH uses a protocol processor that is highly customized for executing protocol handlers, 2) they consider only uniprocessor nodes in their experiments, and 3) they assume a slower network latency of 220 ns., as opposed to 70 ns. in our base parameters.

In [12], Reinhardt *et al.* introduce the Wisconsin Typhoon architecture that relies on a SPARC processor core integrated with the other components of the coherence controller to execute coherence handlers that implement a Simple COMA protocol. Their simula-

tions show that Simple COMA on Typhoon is less than 30% slower than a custom hardware CC-NUMA system. It is hard to compare our results to theirs because of the difficulty in determining what fraction of the performance difference is due to Simple COMA vs. CC-NUMA, and what fraction is due to custom hardware vs. protocol processors.

In [13], Reinhardt *et al.* compare the Wisconsin Typhoon and its first-generation prototypes with an idealized Simple COMA system. Here, their results show that the performance penalty of using integrated protocol processors is less than 20%. In contrast, we find larger performance penalties of up to 106%. There are two main reasons for this difference: 1) we are considering a more decoupled design than Typhoon, and 2) the application set used in the studies. Our results largely agree with theirs for Barnes, the only application in common between the two studies. However, we also consider applications with higher bandwidth requirements, such as Ocean, Radix, and FFT. Other differences between the two studies are: a) they compare Simple COMA systems, while we compare CC-NUMA systems, b) they assume a slower network with a latency of 500 ns., which mitigates the penalty of protocol processors, and c) they consider only uniprocessor nodes.

Holt *et al.* [4] perform a study similar to ours. They also find that the occupancy of coherence controllers is critical to the performance of high-bandwidth applications. However, their work uses abstract parameters to model coherence controller performance, whereas our work considers practical, state-of-the-art controller designs. Also, our work provides strong insight into coherence controller bottlenecks, and we study the effect of having multiple processors per node and two protocol engines per coherence controller.

## 5 Conclusions

The major focus of our research is on characterizing the performance tradeoffs between using custom hardware versus protocol processors to implement cache coherence protocols. By comparing designs that differ only in features specific to either approach and keeping the rest of the architectural parameters identical, we are

able to perform a systematic comparison of both approaches. We find that for applications with high bandwidth requirements, like Ocean, Radix, and FFT, the occupancy of off-the-shelf protocol processors significantly degrades performance by up to 106% for the applications we studied. On the other hand, the programmable nature of protocol processors allows one to tailor the cache coherence protocol to the application, and may lead to shorter design times since protocol errors may be fixed in software.

We also find that using a slow network or large data sizes results in tolerable protocol processor performance, and that for communication-intensive applications, performance degrades with the increase in the number of processors per node, as a result of the decrease in the number of coherence controllers in the system.

Our results also demonstrate the benefit of using two protocol engines in improving performance or maintaining the same performance of systems with larger number of coherence controllers. We are investigating other optimizations such as using more protocol engines for different regions of memory, and using custom hardware to implement accelerated data paths and handler paths for simple protocol handlers, which usually incur the highest penalties on protocol processors relative to custom hardware.

Our analysis of the application characteristics captures the communication requirements of the applications and their impact on performance penalty. Our characterization-RCCPI-can help system designers predict the performance of coherence controllers with other applications.

The results of our research imply that it is crucial to reduce protocol processor occupancy in order to support high-bandwidth applications. One approach is to custom design a protocol processor that is optimized for executing protocol handlers, as in the Stanford FLASH multiprocessor. Another approach is to customize coherence protocols to the communication requirements of particular applications. We are currently investigating an alternative approach: to add incremental custom hardware to a protocol-processor-based design to accelerate common protocol handler actions.

## Acknowledgments

We would like to thank several colleagues at IBM Research for their help in this research: Moriyoshi Ohara for his valuable insights and his generosity with his time and effort, Mark Giampapa for porting Augmint to the PowerPC, Kattamuri Ekanadham for useful discussions on coherence controller architectures, and Michael Rosenfield for his support and interest in this work. We also thank Steven Reinhardt at the University of Wisconsin for providing us with information about Typhoon performance, and Mark Heinrich and Jeffrey Kuskin at Stanford University for providing us with information about custom protocol processors.

## References

- [1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiawicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.
- [2] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, and D. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, November 1994.
- [3] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. P. Singh, R. Simoni, K. Gharachorloo, J. Baxter, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.
- [4] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford University, January 1995.
- [5] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.
- [6] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
- [7] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.
- [8] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 68–79, July 1995.
- [9] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *Proceedings of the 1996 IEEE International Conference on Computer Design*, pages 486–490, October 1996.
- [10] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995.
- [11] *Pentium Pro Family Developer's Manual*. Intel Corporation, 1996.
- [12] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.
- [13] S. Reinhardt, R. Pfile, and D. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.
- [14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.