Coherence Controller Architectures for Scalable Shared-Memory Multiprocessors

Maged M. Michael, Ashwini K. Nanda, and Beng-Hong Lim

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 shows minimal increase in application execution time due to protocol processors over custom hardware. With the advent of SMP nodes and faster processors and networks, the trade-off 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 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 commodity protocol processors can be twice as long as using custom hardware. We also 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 performance. We also study the potential of improving the performance of coherence controllers by separating or duplicating critical components.

Index Terms—Coherence controller, shared memory, multiprocessor, protocol processor.

✦

1 Introduction

Previous research has shown 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 [12] 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 [14], 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], [10] 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 percent 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 percent of custom hardware CC-NUMA controllers [14] and within 20 percent of custom hardware Simple-COMA controllers [15].

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 trade-offs between these 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 [13], 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 system components. What distinguishes our work from previous research is that we consider SMP-based

- *M.M Michael and A.K. Nanda are with the IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598. E-mail: michael@watson.ibm.com.*
- *B.-H. Lim is with VMWare, Inc., 44 Encina Ave., Palo Alto, CA 94301.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108230.

0018-9340/99/$10.00 © 1999 IEEE

TABLE 1
Base System No Contention Latencies in Computer Processor Cycles (5 ns.)

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 addr. 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

CC-NUMA and a wider range of architectural parameters. We simulate eight applications from the SPLASH-2 benchmark suite [16] to compare the application performance of the architectures. The results show that, for a 64-processor system based on four-processor SMP nodes, commodity protocol processors result in a performance penalty (increase in execution time relative to that of custom hardware controllers) of 4 percent-93 percent, and that custom protocol processors result in a performance penalty of 2 percent-49 percent.

The high penalty of protocol processors occurs for applications with high 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 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 commodity protocol processors significantly degrades performance relative to custom hardware and that custom protocol processors achieve intermediate performance between the two designs.

We also study the performance of coherence controllers with two protocol engines. Our results show that, for applications with high communication requirements, a two-engine hardware controller improves performance by up to 18 percent over a one-engine hardware controller, and a controller with two custom protocol processors improves performance by up to 24 percent over a controller with a single custom protocol processor, while a controller with two commodity protocol processors improves performance by up to 30 percent over a controller with a single commodity protocol processor. We also demonstrate the significant effect of coherence controller occupancy on overall system performance.

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 trade-offs and provides analysis of the causes of the performance differences between the architectures. Section 4 discusses related work. In Section 5, we conclude with our recommendations for future designs.

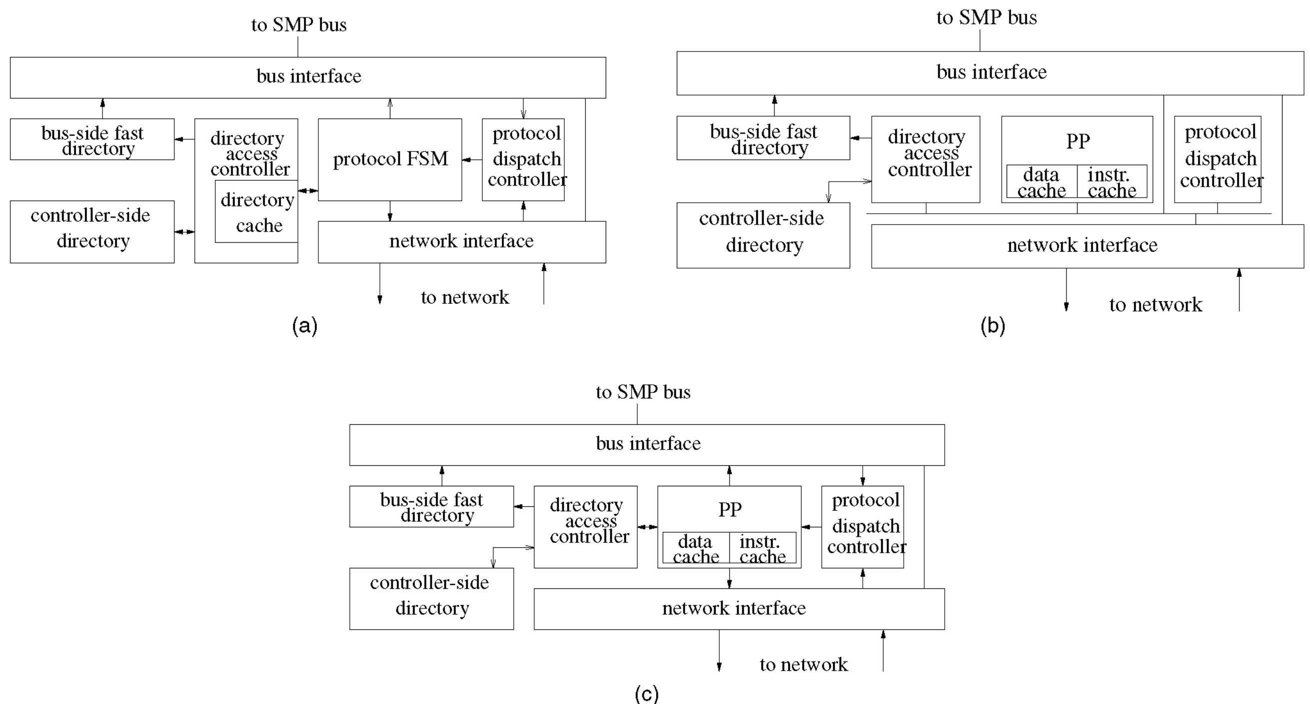

Fig. 1. (a) Custom hardware coherence controller design (HWC), (b) commodity PP-based coherence controller design (PPC), and (c) custom PP-based coherence controller design.

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.

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 network. Each SMP node includes four 200 MHz PowerPC compute processors with 16 KB L1 and 1 MB 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 bus. The memory is interleaved and the memory controller is a separate bus agent from the coherence controller. Table 1 shows the no contention latencies of key system components.

2.2 Coherence Controller Architectures

We consider three main coherence controller designs: a custom hardware coherence controller, a coherence controller based on custom protocol processors, and a coherence controller based on commodity protocol processors.

The three designs share some common components and features (Fig. 1). All three designs use duplicate directories to allow fast response to common requests on the pipelined SMP bus (one directory lookup per two 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. All three designs use directory caches [9] for reducing directory read latency. Each directory cache holds up to 8K full-bit-map directory entries. The hardware-based design uses a custom on-chip cache, while the design with a protocol-processor (PP) use the PP's on-chip data caches. We assume perfect instruction caches in the PPs, as the total size of all protocol handlers in our protocol is less than 16 KB.

All three 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.

Fig. 1a 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. Fig. 1b shows a block diagram of a commodity-PP-based coherence controller (PPC). The PP is a PowerPC running at 200 MHz. The other controller components run at 100 MHz. The PP 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.

Fig. 1c shows a block diagram of a custom-PP-based coherence controller (CPPC). The controller uses a processor core optimized for protocol handling integrated with all the other coherence controller components, except the directory, on the same chip. As in the design based on a commodity PP, the PP runs at 200 MHz, while the other controller components run at 100 MHz.

In order to increase the bandwidth of the coherence controller, we also consider the use of two PPs in the PPC and CPPC implementations and two protocol FSMs in the HWC implementation. We use the term "protocol engine" to refer to the PP in the PPC and CPPC designs 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 [12], 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. Fig. 2 shows the two-engine HWC design (2HWC), the two commodity PP controller design (2PPC), and the two custom PP controller design (2CPPC).

2.3 Controller Latencies and Occupancies

We modeled all designs accurately with realistic parameters. Table 2 lists protocol engine suboperations and their occupancies[1] for each of the HWC, PPC, and CPPC coherence controller designs, assuming a 100 MHz HWC and 100 MHz PPC and CPPC with 200 MHz PPs. 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: 1) accesses to on-chip registers for HWC and CPPC take one system cycle (two CPU cycles), 2) bit operations on HWC are combined with other actions, such as conditions and accesses to special registers, 3) PP reads to off-chip registers on the local PPC bus take four system cycles (eight CPU cycles) and searching a set of associative registers takes an extra system cycle (two CPU cycles), 4) PP writes to off-chip registers on the local PPC bus take two system cycles (four CPU cycles) before the PP can proceed, 5) PPC's PP compute cycles are based on the PowerPC instruction cycle counts produced by the IBM XLC C compiler, while CPPC's PP compute cycles assume a single cycle PP core design optimized for protocol handling, and 6) HWC can decide multiple conditions in one cycle.

3 Experimental Results

In this section, we present simulation results of the relative performance of the coherence controller architectures with several variations of communication-related architectural parameters. Then, we present analysis of the key 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.

1. Occupancy of suboperations is the time a protocol engine is occupied by the suboperation and cannot service other requests.

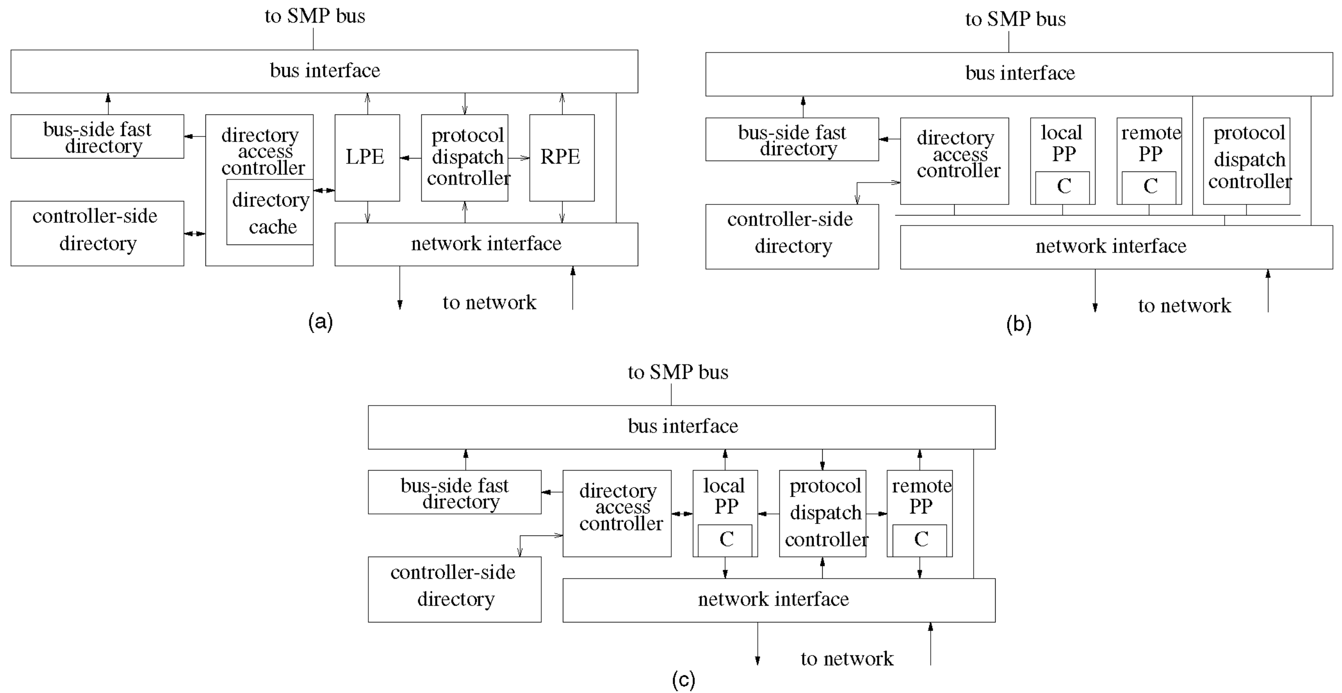

Fig. 2. (a) Custom hardware coherence controller design with local and remote protocol FSMs (2HWC), (b) commodity PP-based coherence controller design with local and remote PPs (2PPC), and (c) custom PP-based coherence controller design with local and remote PPs (2CPPC).

3.1 Experimental Methodology

We use execution-driven simulation (based on a PowerPC version of the Augmint simulation toolkit [11]) to evaluate the performance of the six coherence controller designs, HWC, PPC, CPPC, 2HWC, 2PPC, and 2CPPC. 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 suboperations in Table 2, in addition to accurate timing of the interaction between the coherence controller and the SMP bus, memory, directory, and network interface. All coherence controller implementations use the same cache coherence protocol.

TABLE 2
Protocol Engine Suboperation Occupancies for HWC, PPC, and CPPC in Compute Processor Cycles (5 ns)

Sub-operation	HWC	PPC	CPPC
Issue request to bus	2	8	2
Detect response from bus	2	8	2
Issue network message	2	9	4
Read bus interface associative registers	4	10	4
Write special bus interface registers	2	4	2
Directory read (cache hit)	2	2	2
Directory read (cache miss)	22	22	22
Directory write	2	2	2
Handler dispatch	2	12	6
Condition	2	2	2
Loop (per iteration)	2	5	2
Clear bit field		3	1
Extract bit field		2	1
Other bit operations		1	1

We use eight benchmarks from the SPLASH-2 suite [16], (Table 3) to evaluate the performance of the six 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. LU and Cholesky are run on 32-processor systems (eight nodes × four processors each), as they suffer from load imbalance on 64 processors with the data sets used [16]. We ran all the applications with data sizes and system sizes for which they achieve acceptable speedups.

3.2 Performance Results

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

3.2.1 Base Case

Fig. 3 shows the execution times for the six coherence controller architectures on the base system configuration

TABLE 3
Benchmark types and data sets

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

normalized by the execution time of HWC. The PPC PP penalty can be as high as 93 percent for Ocean and 52 percent for Radix, and the CPPC PP penalty can be as high as 49 percent for Ocean. The significant PP penalties for Ocean, Radix, and FFT indicate that 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 PP-based coherence controllers suffer small performance penalties relative to HWC.

The performance of 2PPC and 2CPPC is comparable to that of CPPC and HWC, respectively. This shows that the performance gains as a result of the extra bandwidth provided by using two protocol engines counter the performance losses due to using lower bandwidth (higher occupancy) protocol engine designs. Also, for applications with high bandwidth requirements, using two protocol engines improves performance significantly relative to the corresponding single engine implementation by up to 18 percent on HWC, 24 percent on CPPC, and 30 percent 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.

3.2.2 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. Fig. 4 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 [16] and a minor increase in execution time for the other benchmarks.

Also, we notice a significant increase in the PP penalties (compared to the PP penalties 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 PPC PP penalty for FFT increases from 45 percent to 68 percent.

3.2.3 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 μs latency). Fig. 5 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. For example, the PPC PP penalty for Ocean drops from 93 percent to 28 percent. Accordingly, systems designs with slow networks can afford to use 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.

3.2.4 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 514 $\times$ 514 grid for Ocean. Fig. 6 shows the execution times normalized by the execution time of HWC for each data size. We notice a decrease in the PP penalties in comparison to the penalties with the base data sizes, since the communication-to-computation ratios for Ocean and FFT decrease with the increase of the data size. The PPC PP penalty for FFT drops from 46 percent to 33 percent, and for Ocean from 93 percent to 67 percent, while the CPPC PP penalty for Ocean drops from 49 percent to 33 percent.

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.

3.2.5 Number of Processors per Node

Varying the number of processors per 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. Fig. 7 shows the execution times on 64-processor systems (32 for LU and Cholesky) with one, two, four, and eight processors per node, normalized to the execution time of HWC on the base configuration (four processors/node).

For applications with low communication rates, the increase in processors per node has only a minor effect on

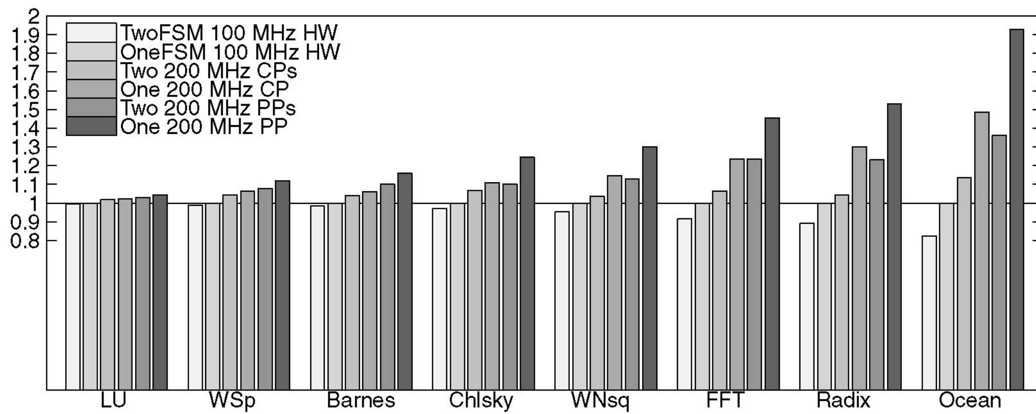

Fig. 3. Normalized execution time on the base system configuration.

the PP performance penalty. For applications with high communication rates, the increase in processors increases the PP performance penalty (e.g., the PPC PP penalty increases from 93 percent for Ocean on four processors per node to 106 percent on eight processors per node). However, the PPC PP penalty can be as high as 79 percent (for Ocean), even on systems with one processor per node.

For each of the 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. With the exception of FFT, with higher numbers of processors per node, the use of two protocol engines achieves similar or better performance than controllers with one protocol engine with half the number of processors per node. In other words, using two protocol engines allows integrating twice as many processors per node, thus saving the cost of half the coherence controllers in the system.

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 4 shows communication statistics collected from simulations of HWC, CPPC, and PPC on the base system configuration (except that Cholesky and LU are run on 32 processors).

The statistics are: 1) PP penalty, the increase in the execution time of PPC (CPPC) relative to the execution time of HWC, 2) RCCPI (Requests to Coherence Controller Per Instruction), the total number of requests to the coherence controllers divided by the total number of instructions, 3) the total of the occupancies of all coherence controllers for PPC (CPPC) divided by that for HWC, 4) average HWC (PPC, CPPC) utilization (i.e., the average HWC (PPC, CPPC) occupancy divided by execution time), 5) average HWC (PPC, CPPC) queuing delay, the average time a request to the coherence controller waits in a queue while the controller is occupied by other requests, and 6) arrival rate of requests to HWC (PPC, CPPC) per μs (200 CPU cycles).

In Table 4, we notice that, as RCCPI increases, the PP performance penalty increases proportionally (excluding LU and Cholesky, which are run on 32 processors). Also, as RCCPI increases, the arrival rate of requests to the coherence controller per cycle for PPC and CPPC diverges from that of HWC, indicating that the PPC and CPPC controllers have been saturated and that the coherence controller is the bottleneck for the base system configuration. This is also supported by the observation of the high

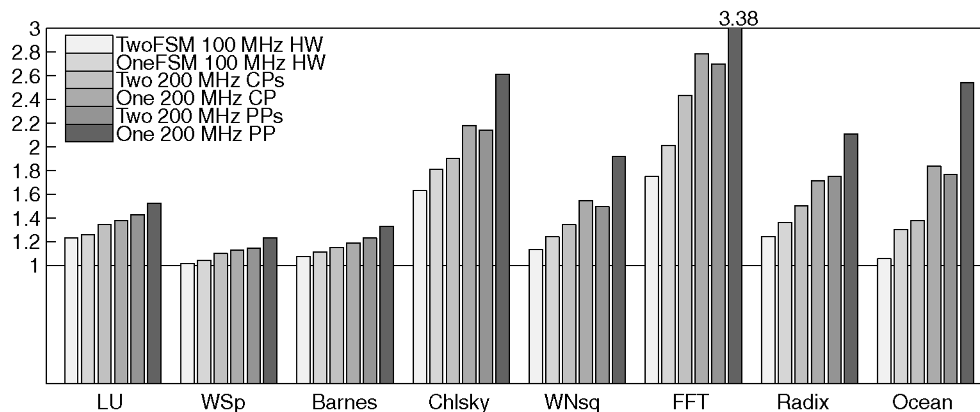

Fig. 4. Normalized execution time for system with small (32 byte) cache lines.

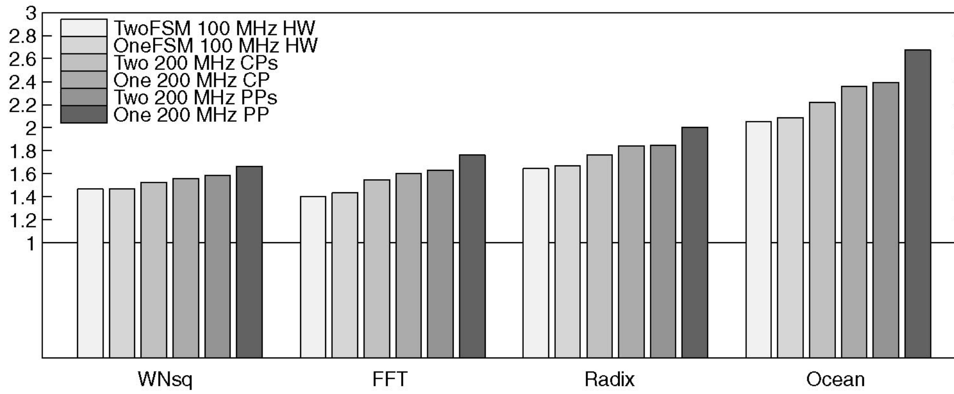

Fig. 5. Normalized execution time for system with high (1 μs) network latency.

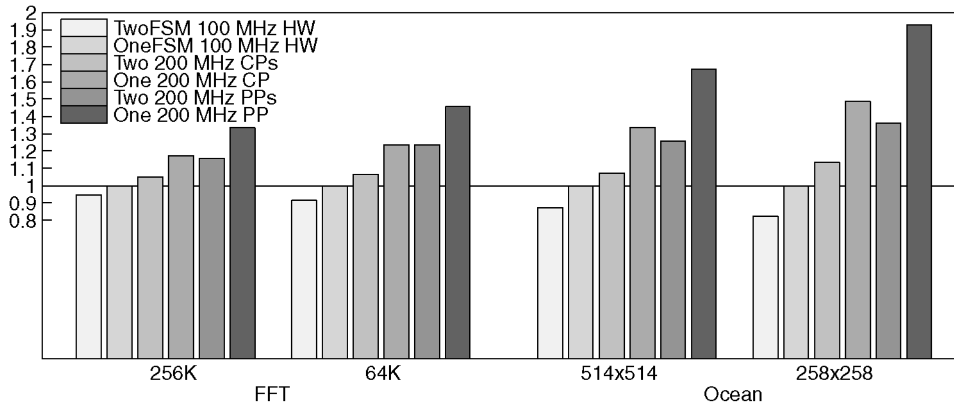

Fig. 6. Normalized execution time for base system with base and large data sizes.

utilization rates of HWC with Ocean and CPPC and PPC with Ocean, Radix, and FFT, indicating that the coherence controller has been saturated in these cases and that it is the main bottleneck.

However, the queuing delays do not increase proportionally with the increase in RCCPI. The queuing effect of the coherence controller behaves like a negative feedback system. In such a system, the output (queuing delay) is proportional to the input (RCCPI) and the difference between the output (queuing delay) and a saturation value. That is, the increase of queuing delay with the increase of RCCPI is dampened as the coherence controller nears saturation.

The ratio between the occupancy of PPC and the occupancy of HWC is more or less constant for the different applications, approximately 2.5, and the ratio between the average occupancies of CPPC and HWC is approximately 1.8. The strong correlation between the relative occupancies of PPC and CPPC to HWC, and the PP penalty for PPC and CPPC, respectively, shows the dominant effect of coherence controller occupancy on overall system performance.

Fig. 8 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 controller architecture is the performance bottleneck of the base system.

Fig. 9 shows the effect of RCCPI on the PPC PP penalty for the results from Table 4 (excluding LU and Cholesky, which are run on a different system configuration than the other applications). 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 underutilized.

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 5 shows the communication statistics collected from simulations of 2HWC, 2CPPC, and 2PPC on the base system

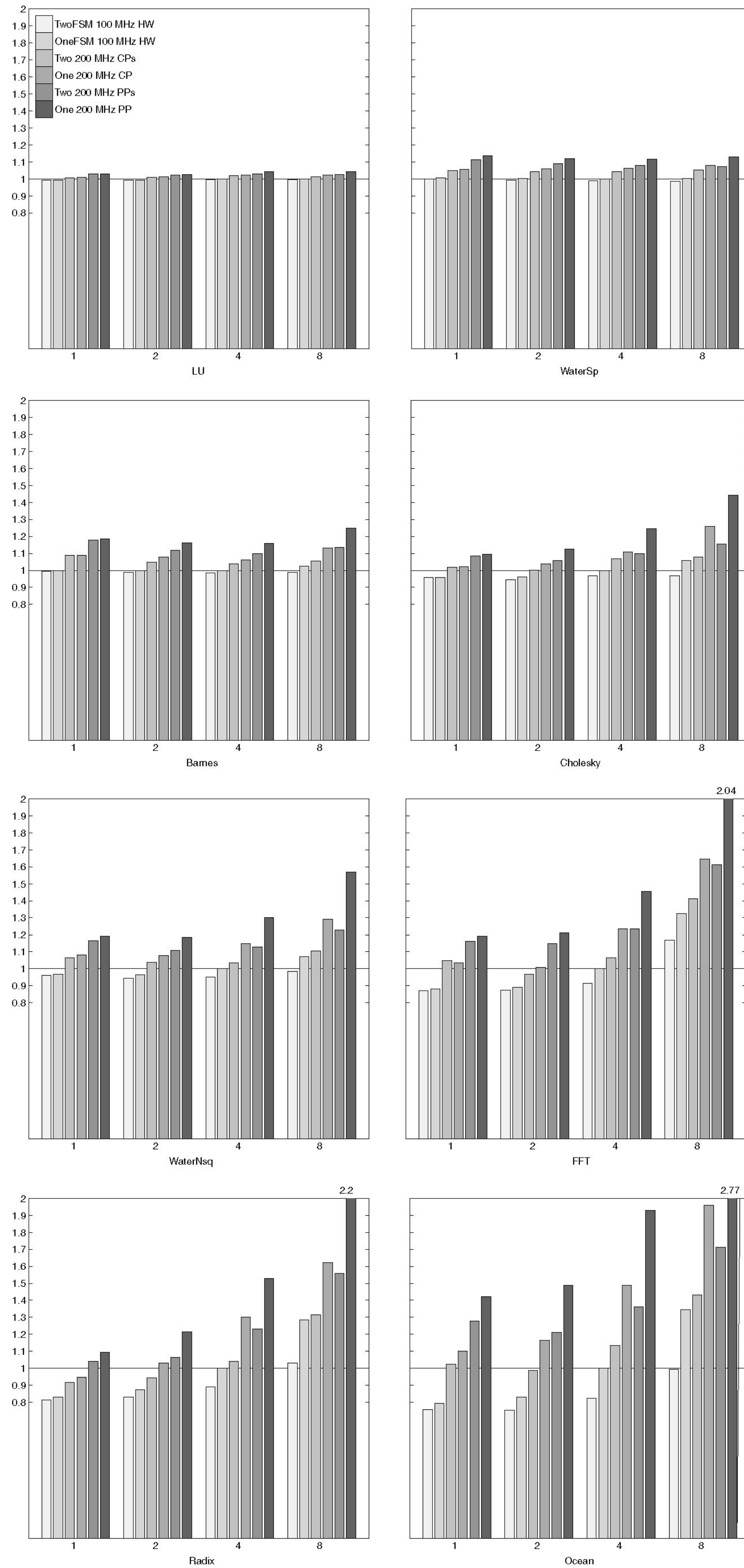

Fig. 7. Normalized execution time with one, two, four, and eight processors per node.

TABLE 4
Communication Statistics on the Base System Configuration

Application	PPC PP penalty	CPPC PP penalty	PPC/HWC occupancy	CPPC/HWC occupancy	HWC utilization	CPPC utilization	PPC utilization
LU	4.37%	2.20%	2.37	1.72	4.21%	7.09%	9.58%
Water-Sp	11.69%	6.47%	2.65	1.87	10.95%	19.23%	25.99%
Barnes	15.81%	7.28%	2.52	1.85	13.26%	22.87%	28.88%
Cholesky	24.38%	10.86%	2.23	1.69	26.38%	40.21%	47.37%
Water-Nsq	30.15%	13.80%	2.69	1.90	17.86%	29.82%	36.87%
FFT-256K	33.44%	16.79%	2.38	1.73	22.13%	32.78%	39.54%
FFT-64K	45.59%	22.82%	2.31	1.69	29.61%	40.74%	46.96%
Radix	52.83%	25.66%	2.36	1.69	36.82%	49.52%	56.75%
Ocean-514	67.26%	33.46%	2.29	1.67	47.54%	59.48%	65.07%
Ocean-258	92.88%	48.99%	2.47	1.72	52.89%	61.06%	67.72%

Application	1000 × RCCPI	HWC queuing delay (ns.)	CPPC queuing delay (ns.)	PPC queuing delay (ns.)	Average requests to HWC per μs.	Average requests to CPPC per μs.	Average requests to PPC per μs.
LU	1.3	101	196	305	0.41	0.40	0.40
Water-Sp	1.8	100	203	375	1.19	1.09	1.06
Barnes	2.3	67	143	266	1.26	1.13	1.09
Cholesky	4.1	113	239	365	2.34	1.98	1.86
Water-Nsq	3.3	157	324	626	1.85	1.53	1.43
FFT-256K	3.7	289	458	837	1.83	1.49	1.38
FFT-64K	6.3	340	562	864	2.58	2.02	1.77
Radix	9.8	229	452	640	3.66	2.79	2.33
Ocean-514	14.0	226	460	648	3.87	2.87	2.31
Ocean-258	23.2	232	504	720	4.69	3.30	2.41

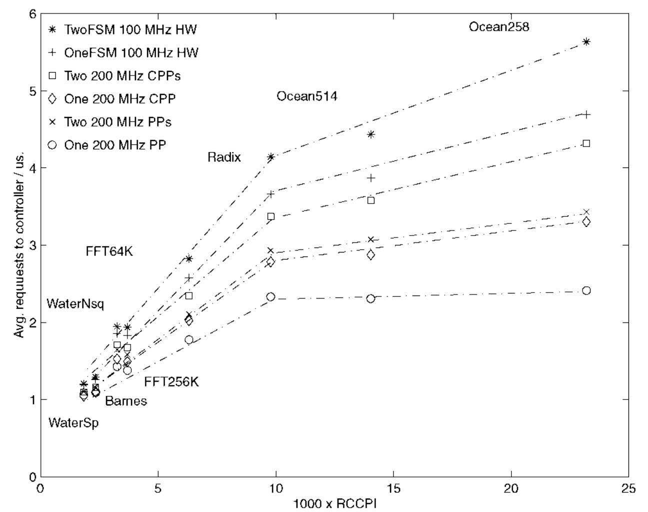

Fig. 8. Coherence controller bandwidth limitations.

configuration (except Cholesky and LU are run on 32 processors).

Although most requests are handled by RPE (53-63 percent), the occupancy of LPE is up to three times that of RPE for 2HWC, up to 2.5 times for 2CPPC, and up to two 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.

The large imbalance in the distribution of occupancy between LPE and RPE (derived from the utilization statistics) for most applications indicates 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.

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 percent 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.

In [14], 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 simulations show that Simple COMA on Typhoon is less than 30 percent 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 [15], Reinhardt et al. compare the Wisconsin Typhoon and its prototypes with an idealized Simple COMA system. Here, their results show that the performance penalty of using integrated protocol processors is less than 20 percent. In contrast, we find larger performance penalties of up to 106 percent. This is due to these differences: 1) They mostly consider applications with low communication requirements, 2) they compare Simple COMA systems, while we compare CC-NUMA systems, 3) they assume a slower network, which mitigates the penalty of protocol processors, and 4) 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

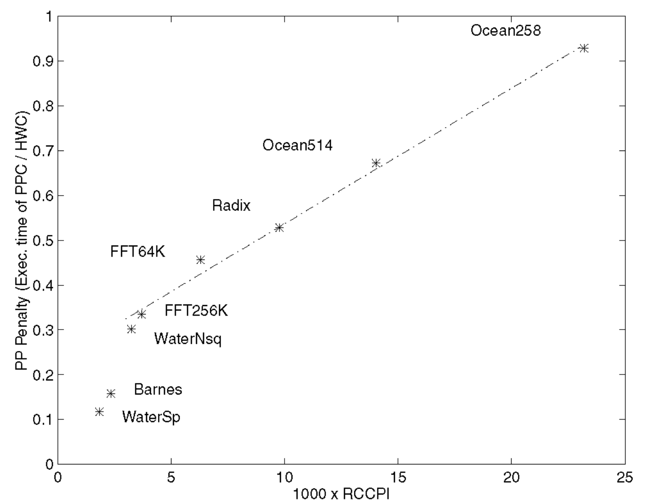

Fig. 9. Effect of communicaion rate on PP penalty.

TABLE 5
Communication Statistics for Controllers with Two Protocol Engines on the Base System Configuration

Appl.	Arch.	Utilization (%)		Distribution (%)		Queuing delay (ns.)	
		LPE	RPE	LPE	RPE	LPE	RPE
LU	2HWC	3.20	1.09	35.67	64.33	182	2
	2CPPC	4.48	2.62	35.70	64.30	311	7
	2PPC	5.66	3.92	35.74	64.26	501	14
Water-Sp	2HWC	6.82	4.29	38.09	61.91	60	40
	2CPPC	11.03	8.43	38.08	61.92	126	59
	2PPC	14.66	12.38	38.08	61.92	263	78
Barnes	2HWC	8.43	5.22	39.38	60.62	67	11
	2CPPC	13.04	9.87	39.40	60.60	125	24
	2PPC	16.64	13.85	39.41	60.59	237	53
Cholesky	2HWC	20.26	7.48	38.27	61.73	128	8
	2CPPC	25.96	15.08	38.27	61.73	239	16
	2PPC	30.34	19.99	38.27	61.73	348	36
Water-Nsq	2HWC	11.30	7.89	39.26	60.74	82	49
	2CPPC	17.07	14.24	39.24	60.76	173	99
	2PPC	22.87	19.81	39.22	60.78	384	167
FFT-256K	2HWC	17.93	5.92	46.33	53.67	378	10
	2CPPC	24.42	10.87	46.33	53.67	602	19
	2PPC	30.64	15.05	46.33	53.67	934	38
FFT-64K	2HWC	25.63	7.45	41.40	58.60	478	8
	2CPPC	31.76	13.96	41.40	58.60	893	20
	2PPC	36.35	19.17	41.40	58.60	1137	39
Radix	2HWC	21.63	21.32	39.95	60.05	138	91
	2CPPC	26.74	31.55	39.94	60.06	187	154
	2PPC	30.70	40.86	39.94	60.06	243	366
Ocean-514	2HWC	38.10	18.33	41.03	58.97	210	35
	2CPPC	44.78	28.38	41.03	58.97	323	76
	2PPC	50.42	36.59	41.02	58.98	480	138
Ocean-258	2HWC	40.02	25.97	40.45	59.55	173	48
	2CPPC	47.26	36.93	40.43	59.57	298	98
	2PPC	52.60	44.19	40.39	59.61	476	185

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 trade-offs 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 percent 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.

Finally, we recommend using custom hardware coherence controllers in future multiprocessor designs, with multiple protocol engines in order to increase the bandwidth of the controller. We also conclude that, for protocol processors to be practical, it is crucial to reduce their occupancy. Possible approaches for achieving that goal are by using extra customization in their design and/or by adding incremental custom hardware to accelerate common protocol handler actions.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 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," *Proc. Supercomputing '94,* Nov. 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. Rosemblum, and J. Hennessy, "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 274-285, Oct. 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 Univ., Jan. 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," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.

[6] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. Lam, "The Stanford DASH Multiprocessor," *Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.

[7] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 308-317, May 1996.

[8] M.M. Michael, A.K. Nanda, B.-H. Lim, and M.L. Scott, "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 219-228, June 1997.

[9] M.M. Michael and A.K. Nanda, "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors," *Proc. Fifth Int'l Symp. High Performance Computer Architecture,* pp. 142-151, Jan. 1999.

[10] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers and J. Saltz, "Efficient Support for Irregular Applications on Distributed-Memory Machines," *Proc. Fifth ACM Symp. Principles and Practice of Parallel Programming,* pp. 68-79, July 1995.

[11] A.-T. Nguyen, M.M. Michael, A.D. Sharma, and J. Torrellas, "The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures," *Proc. 1996 IEEE Int'l Conf. Computer Design,* Oct. 1996.

[12] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke and S. Vishin, "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. 1995 Int'l Conf. Parallel Processing,* Aug. 1995.

[13] *Pentium Pro Family Developer's Manual.* Intel Corp., 1996.

[14] S. Reinhardt, J. Larus, and D. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 325-336, Apr. 1994.

[15] S. Reinhardt, R. Pfile, and D. Wood, "Decoupled Hardware Support for Distributed Shared Memory," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 34-43, May 1996.

[16] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

Maged M. Michael received the PhD degree in computer science from the University of Rochester in 1997. He is currently a research staff member in the Scalable Server Architecture Department at the IBM T.J. Watson Research Center. His research interests include multiprocessor architecture, execution-driven simulation, shared memory multiprocessor synchronization, and cache coherence on large-scale multiprocessors.

Ashwini K. Nanda received the BSC (Eng) degree from Sambalpur University, India, the MTech degree in electrical engineering from the Indian Institute of Technology, Madras, and the PhD degree in computer science from Texas A&M university, College Station, Texas. He currently leads a research group on scaleable shared-memory systems and also manages the scaleable server architecture group at the IBM T.J. Watson Research Center in Yorktown Heights, New York. He has worked in the past for Wipro Information Technology Ltd. , Bangalore, Center for Advanced Computing, Pune, India, and Texas Instruments, Dallas, Texas. At Texas Instruments, he worked on the architecture and design of an experimental superscalar processor.

Beng-Hong Lim received his PhD in electrical engineering and computer science from MIT in 1994, where he worked on the architecture and software of the Alewife machine. Dr. Lim is currently working at VMWare, Inc., a startup company in Palo Alto, California. Prior to that, he worked on scalable, reliable multiprocessor servers at IBM in Yorktown Heights, New York.