RC 20675 (91675) 1/2/97
Computer Science/Mathematics 11 pages

Research Report

Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors

Maged M. Michael
University of Rochester
Department of Computer Science
Rochester, NY 14627

Ashwini K. Nanda, Beng-Hong Lim
IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Michael L. Scott
University of Rochester
Department of Computer Science
Rochester, NY 14627

IBM **Research Division**
Almaden · T.J. Watson · Tokyo · Zurich · Austin

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

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

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

‡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 are responsible for executing 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 development time. Previous research show that the increase in application execution time due to protocol processors over custom hardware is minimal ($< 20\%$).

However, with the advent of SMP processing nodes and the availability of 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 bandwidth 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 on performance penalty of varying several architectural parameters that influence the communication characteristics of the applications and the underlying system. We characterize each application's communication requirements and its impact on the performance penalty of protocol processors. This characterization 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 controllers by separating or duplicating critical components.

*This work was performed at IBM Thomas J. Watson Research Center.

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 [5] 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 [8] multiprocessor uses hardware sequencers for modularity in implementing protocol handlers.

Subsequent designs for scalable shared-memory multiprocessors, such as the Stanford FLASH [4] and the Wisconsin Typhoon machines [11], 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, and may lead to shorter design times since protocol errors may be fixed in software.

Simulations of 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 [2]. 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 perfor-

mance that is within 30% of custom-hardware CC-NUMA controllers [11] and within 20% of custom-hardware Simple-COMA controllers [10].

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 based on the Intel Pentium-Pro [9] makes SMP nodes an attractive choice for CC-NUMA designers [6]. 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 [12] 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 4x16 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 tradeoffs 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 demonstrates 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 CC-NUMA architectures under study. 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 fast state-of-the-art network. Each SMP node includes four 200 MHz PowerPC 604 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 a SMP node. Table 1 shows the no-contention latencies of key system components. These parameters correspond to those of existing state-of-the-art components. Note that the memory controller starts memory reads speculatively, thus the fast

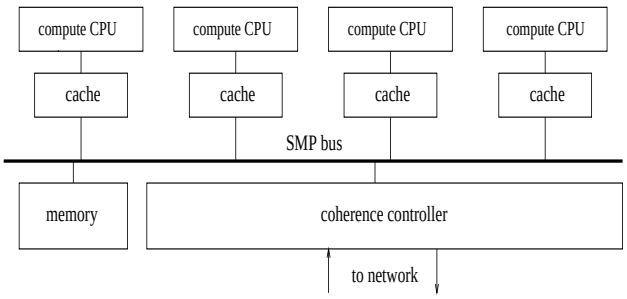

Figure 1: A node in a SMP-based CC-NUMA system.

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 latencies in compute processor cycles (5 ns.).

response time, and 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 [5] and Alewife [1] systems, and a coherence controller based on commodity protocol processors similar to those in the Typhoon [11] system and its prototypes [10].

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 commodity processor's on-chip data caches. [1]

[1] Although most processors use write-back caches, current processors (e.g. Pentium Pro [9]) allow users to designate regions of memory to be cached write-through.

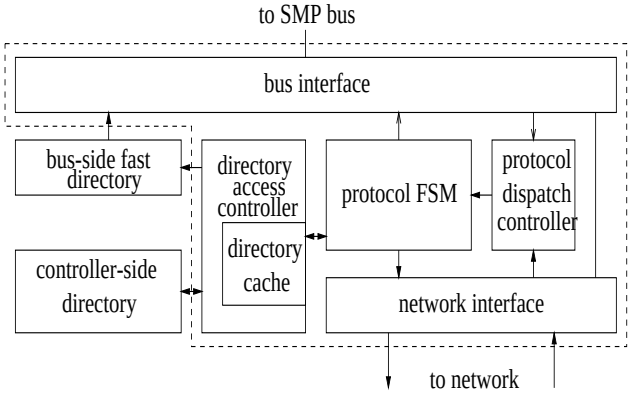

Figure 2: A hardware-based coherence controller design (HWC).

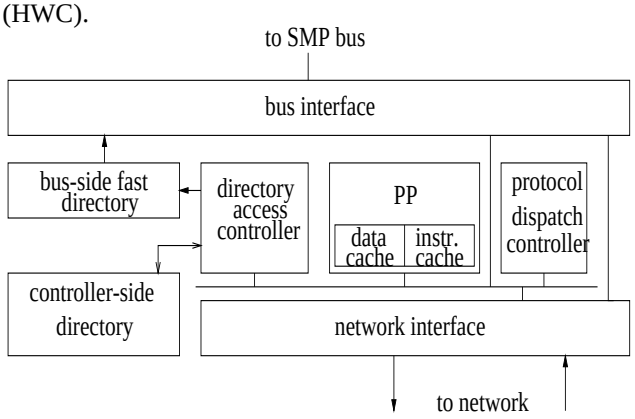

Figure 3: A commodity PP-based coherence controller design (PPC).

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 to 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 ex-

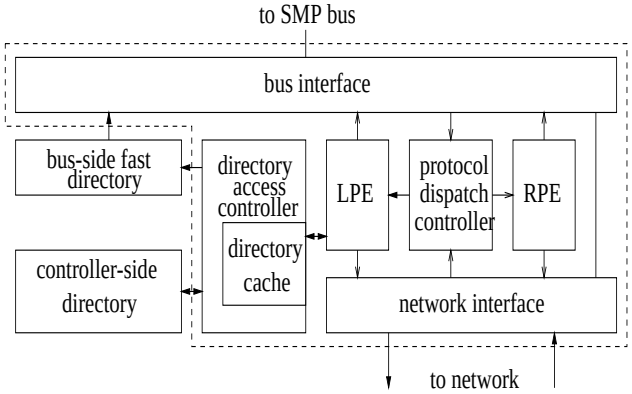

Figure 4: A custom hardware coherence controller design with local and remote protocol FSMs (2HWC).

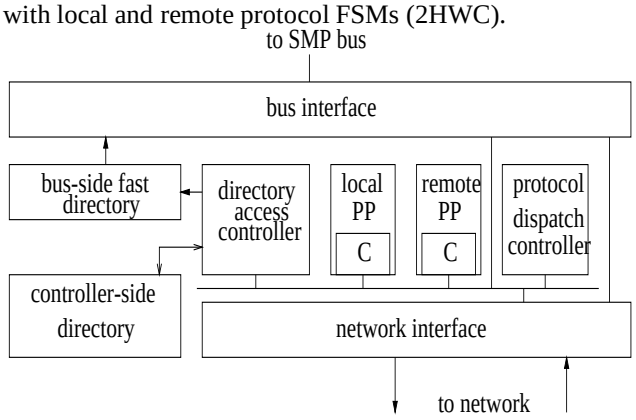

Figure 5: A commodity PP-based coherence controller design with local and remote protocol processors (2PPC).

cept the directories. Figure 3 shows a block diagram of a protocol-processor-based coherence controller (PPC). The protocol processor (PP) is a PowerPC 604 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 bus to memory-mapped off-chip registers in the other components. The protocol processor access to the protocol dispatch controller register is read-only. Its access to the network interface registers is write-only (for sending network message 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 [8], 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.

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 [2] 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 (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

[2] Occupancy refers to the time a resource is occupied and cannot service other potentially waiting requests.

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.).

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 [10], 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 we use the same full-map sequentially consistent directory-based write-back protocol for 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 Performance Results

3.1 Experimental Methodology

We use execution-driven simulation (based on a version of the Augmint simulation toolkit [7] 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 network interface. All coherence controller implementations use the same cache coherence protocol.

We use eight benchmarks from the SPLASH-2 suite [12], (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

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.).

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.

contention as a result of uneven memory distribution. LU and Cholesky are run on 32-processor systems (8 nodes × 4 processors each) as they suffer from load imbalance on 64 processors with the data sets used [12]. We ran all the applications with data sizes and systems sizes for which they achieve acceptable speedups.

3.2 Experimental 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 run experiments on the base system configuration with the four coherence controller architectures, and then we vary some key system parameters, in order 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 bottle-neck 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 using two protocol engines improves performance significantly relative to the correspond-

ing single engine implementation, 18% on HWC and 30% on PPC, for Ocean, and the gap between the one and two-engine designs widens for applications with high bandwidth requirements.

We varied other system and application parameters that are expected to have big impact on the communication requirements of the applications. We start by 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, Water-Nsquared, Cholesky, and LU, which have high spatial locality [12], while there is minor increase in execution time for the other benchmarks.

Also, we notice a significant increase in the PP penalty (compared to PP penalty on 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.

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). 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 514×514 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

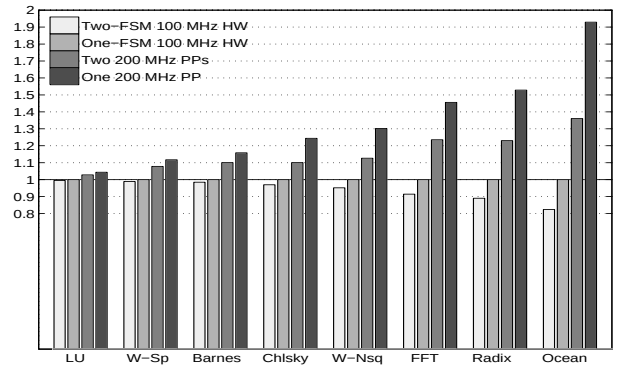

Figure 6: Normalized execution time on the base system configuration.

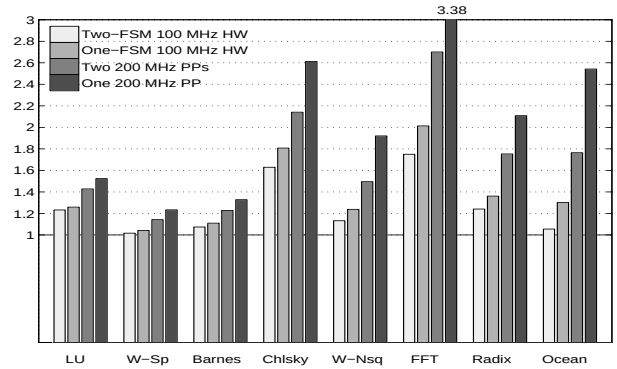

Figure 7: Normalized execution time for system with 32 byte cache lines.

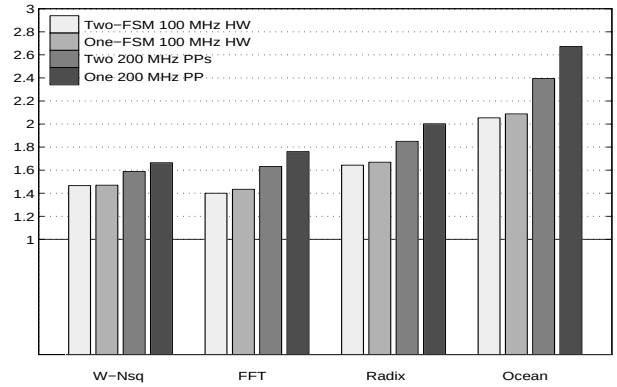

Figure 8: Normalized execution time for system with 1 μs network latency.

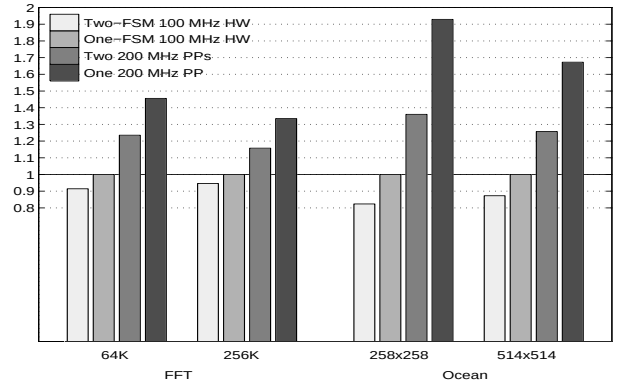

Figure 9: Normalized execution time for base system with base and large data sizes.

the base data sizes, since the communication-to-computation ratios for Ocean and FFT decrease with the increase of the data size [3]. 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 of its 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 have impact on 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 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 processor/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.

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 more processors per SMP node, thus saving the costs of half the SMP buses, memory controllers, coherence controllers, and I/O controllers.

3.3 Analysis

In order to gain more insight into quantifying the application characteristics that affect the 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

[3] Applications like Radix maintain a constant communication rate with different data sizes [12].

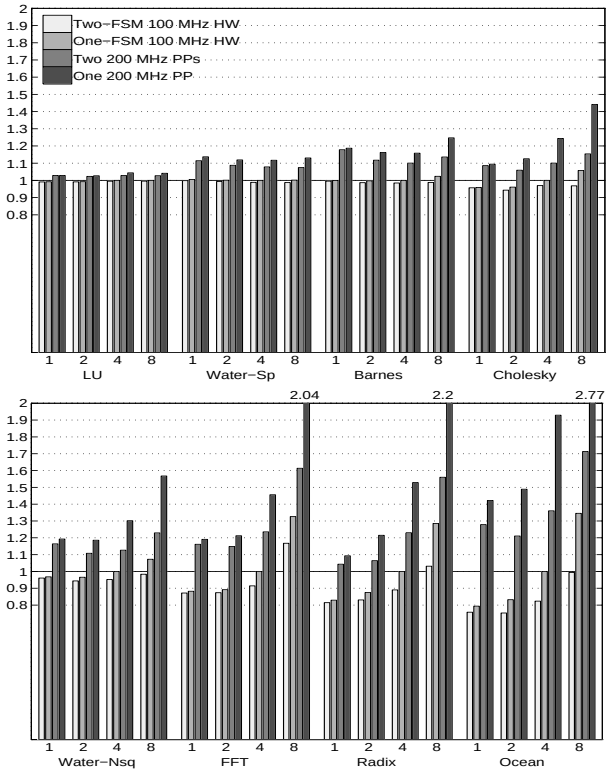

Figure 10: Normalized execution time with 1,2,4, and 8 processors per SMP node.

(except Cholesky and LU are run on 32 processors). The statistics are:

- PP penalty: The increase in 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. This measure is independent of the coherence controller implementation (but depends on the cache coherence protocol) and can be obtained from a simple PRAM simulation model.
- The total 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 μs. (200 CPU cycles) is derived from the reciprocal of the mean inter-arrival time of requests to each of the coherence controllers.

Application	PP Penalty	1000 RCCPI	PPC/HWC occupancy	HWC util.	PPC util.	HWC queuing delay	PPC queuing delay	Avg. requests to HWC. per μs.	Avg. requests to PPC. per μs.
LU	4.37%	1.3	2.37	4.21%	9.58%	20.2	61.0	0.41	0.40
Water-Sp	11.69%	1.8	2.65	10.95%	25.99%	20.0	75.0	1.19	1.06
Barnes	15.81%	2.3	2.52	13.26%	28.88%	13.4	53.2	1.26	1.09
Cholesky	24.38%	4.1	2.23	26.38%	47.37%	22.6	73.0	2.34	1.86
Water-Nsq	30.15%	3.3	2.69	17.86%	36.87%	31.4	125.2	1.85	1.43
FFT	45.59%	6.3	2.31	29.61%	46.96%	68.0	172.6	2.58	1.77
Radix	52.83%	9.8	2.36	36.82%	56.75%	45.8	128.0	3.66	2.33
Ocean	92.88%	23.2	2.47	52.89%	67.72%	46.4	144.0	4.69	2.41

Table 6: Communication statistics on the base system configuration.

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 the saturation of PPC, and that the coherence controller is the bottle-neck 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 the saturation of the coherence controllers in these cases, and that it is the main bottle-neck.

However we notice that the queuing delays do not increase proportionally with the increase in RCCPI, since the queuing model 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 [12].

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 architecture 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, and the divergence in the arrival rates for the different coherence controller architectures demonstrates that the coherence architecture is the performance bottle-neck of the base system.

Figures 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 low 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.

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 [2]. Their measured penalties are significantly lower than ours for the following reasons: i) FLASH uses a protocol processor that is highly customized for executing protocol handlers, ii) they consider only uniprocessor nodes in their experiments, and iii) they assume a slower network latency of 220 ns, as opposed to 70 ns in our base parameters.

In [11], 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

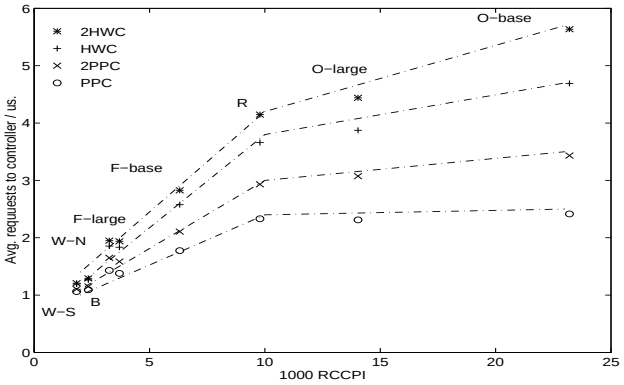

Figure 11: Coherence controller bandwidth limitations.

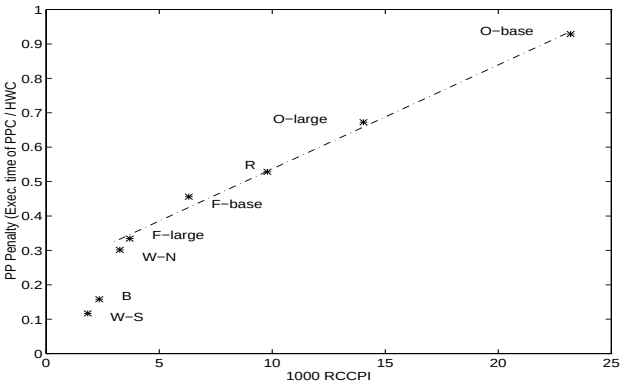

Figure 12: Effect of communication rate on PP penalty.

execute coherence handlers that implement a Simple COMA protocol. Their simulations 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 custom hardware vs. protocol processors.

In [10], 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 close to 100%. There are two main reasons for this difference: i) we are considering a more decoupled design than Typhoon, and ii) 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: i) they compare Simple COMA systems, while we compare CC-NUMA systems, ii) they assume a slower network with a latency of 500 ns, which mitigates the penalty of protocol processors, and iii) they considered only uniprocessor nodes.

Holt *et al.* [3] perform a study similar to ours on comparing various coherence controller architectures and the effect of latency, occupancy and bandwidth on application performance. They also find that the occupancy of coherence controllers is critical to the performance of high-bandwidth applications. However, their study is more theoretical in nature, as they use simple exponential multiplicative factors in modeling the ratios between the occupancies of the various architectures. For instance, the assume that off-the-shelf protocol processor occupancy is 8 times that of custom hardware. In contrast, by modeling the alternative designs in detail, we find that protocol processor occupancy is only 2.5 times that of custom hardware.

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 bandwidth-limited applications, like Ocean, Radix, and FFT, the occupancy of off-the-shelf protocol processors significantly degrades performance.

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. that 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 its impact on performance penalty. Our characterization 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 can either custom design a protocol processor that is optimized for executing protocol handlers, or add custom hardware to accelerate common protocol handler actions. The Stanford FLASH multiprocessor takes the former approach. We are currently investigating the latter approach.

Acknowledgements

We like to thank several colleagues at IBM Research for their help in this research: Moriyoshi Ohara for his valuable insights and his generousity with his time and effort, Mark Giampapa for porting Augmint to the PowerPC, 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. Kubiatowicz, 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 (ISCA'95)*, pages 2–13, June 1995.

[2] 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. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[3] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The Effects of Latency, Occupance, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical report, Stanford University, January 1995.

[4] 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 (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[5] 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.

[6] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–317, May 1996.

[7] 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 (ICCD)*, October 1996.

[8] 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 1995 International Conference on Parallel Processing*, August 1995.

[9] *Pentium Pro Family Developer's Manual*. Intel Corporation, 1996.

[10] S. Reinhardt, R. Pfile, and D. Wood. Decoupled Hardware Support for Duistributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34–43, May 1996.

[11] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[12] 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 International Symposium on Computer Architecture*, pages 24–36, June 1995.

Copies may be requested from:

IBM Thomas J. Watson Research Center
Publications Office, 16-220
Post Office Box 218
Yorktown Heights, NY 10598