

High-Throughput Coherence Controllers*

Ashwini K. Nanda[†], Anthony-Trung Nguyen[‡], Maged M. Michael[†], and Douglas J. Joseph[†]

[†]IBM Research
Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{ashwini,michael,djoseph}@watson.ibm.com

[‡]University of Illinois, Urbana-Champaign
Department of Computer Science
Urbana, IL 68101
anguyen@cs.uiuc.edu

Abstract

Recent research shows that the occupancy of the coherence controllers is a major performance bottleneck for distributed cache coherent shared memory multiprocessors. In this paper we study three approaches to alleviating this problem in hardwired coherence controllers, namely, multiple protocol engines, pipelined protocol engines, and split request-response streams.

Split request-response streams is an innovative contribution of this paper. The performance of pipelining in the context of coherence controllers has not been presented in the literature. Multiple protocol engines has not been studied in the context of hardwired controllers except for a study of ours and only to a limited extent.

Using both commercial and scientific benchmarks on detailed simulation models, we present experimental results that show that each mechanism is highly effective at reducing controller occupancy by as much as 66% and improving execution time by as much as 51%, for applications with high communication bandwidth requirement. A combination of mechanisms further reduces controller occupancy and execution time by as much as 78% and 61%, respectively.

Our results show that applying any of the parallel mechanisms in the coherence controllers allows integrating four times as many processors per coherence controller, thus reducing system cost, while maintaining or even exceeding the performance of systems with larger number of coherence controllers.

1 Introduction

Previous research has shown that scalable shared-memory performance can be achieved on directory-based cache-coherent multiprocessors such as the Stanford

FLASH [3] and the MIT Alewife [1] machines. A key component of this type of machines is the coherence controller on each node that provides cache coherent access to memory that is distributed among the nodes of the multiprocessor.

Recent research results [4, 11] show that the occupancy of the coherence controller (CC) can be the performance bottleneck for applications with high communication requirements.

Motivated by these results, we study three approaches to alleviating this problem: multiple protocol engines (PEs); split request-response streams; and pipelined protocol engines. A coherence controller with multiple PEs can handle multiple coherence transactions in parallel. Each PE can have two protocol handling units, one for handling the request stream and the other for handling the response stream, in parallel. Finally, the protocol handling unit can be pipelined to allow overlapping queue arbitrations, directory accesses, protocol processing, and state maintenance. The increased parallelism and overlapping of coherence operations offered by these approaches serves to decrease the overall occupancy of coherence controllers.

Multiple PEs were employed in the Sun S3.mp [15] architecture. The coherence controller dedicates one protocol engine for handling transactions to local addresses and one for remote addresses. The architects of the Sequent STiNG [8] system also considered a similar approach as one of the ways to reduce controller occupancy. However, the impact of this approach on the performance of these systems was not studied.

Michael et al [11, 10] evaluate systems with one local and one remote PEs in the context of comparing the performance of hardwired and programmable coherence controllers. Their results show significant improvements in the performance of such systems over systems with single protocol-engine controllers. They also find load imbalance of coherence traffic at the protocol engines indicating the potential for more improvements if this issue is resolved.

*This work was performed at the IBM Thomas J. Watson Research Center as part of the HighT project.

Falsafi and Wood [2] study the performance of parallel execution of fine-grained software coherence handlers on an SMP cluster. They find that applications with high communication requirements exhibit low speedups in the base system with a single dedicated protocol processor. On the proposed system that utilizes idle processors on SMP nodes for parallel protocol processing, these applications show acceptable performance improvements.

The Magic chip [3] uses a pipelined and two-way superscalar programmable controller to handle coherence transactions. The pipeline in Magic is very similar to a processor pipeline which fetches and executes instructions from memory to manipulate coherence transactions.

In this paper, we cover the issue of multiple PEs in more depth and its synergy with other occupancy reduction techniques: split request-response streams and pipelining. The split streams approach is an innovative contribution of this paper. In contrast to the Magic chip, the pipelining approach studied in this paper fetches protocol transactions from the network and SMP input ports and manipulates them directly in hardware and is custom architected for coherence protocol handling. The performance impact of pipelining in coherence controllers has not been discussed in the literature in the past. We focus on hardwired coherence controllers because they have been shown to yield superior performance over protocol processors [4, 11].

The results show that each approach is highly effective at reducing controller occupancy by as much as 28% and improving execution time by as much as 16%, for applications with high communication bandwidth requirement on a system with 1 SMP node (or 4 processors) per CC. A combination of mechanisms further reduces controller occupancy and execution time by as much as 32% and 17%, respectively.

As more SMP nodes are attached to each CC, the improvement is more significant. With 4 SMP nodes per CC, the occupancy improves by as much as 66% and execution time by 51%. The compound effect of the three mechanisms is a reduction in controller occupancy and execution time by as much as 78% and 61%, respectively.

This paper makes the following contributions:

- It quantifies the relative performance impact of the three approaches and their synergy.
- It introduces the split request-response streams approach to increase coherence controller parallelism.
- It applies pipelining to the design of hardwired coherence controller. It also applies fine-grained memory interleaving to multiple protocol engines.

The rest of this paper is organized as follows. Section 2 details the three mechanisms for reducing coherence controller occupancy. Section 3 describes our experimental

methodology and presents the experimental results and their implications. Finally, Section 4 presents our conclusions and recommendations for coherence controller designs in future architectures.

2 Coherence Controller Microarchitecture

In this section, we detail the microarchitecture of the coherence controller designs under study, focusing on multiple PEs, pipelining, and split request-response streams. First, we describe the overall system and coherence controller architecture that is common for all the designs presented in the paper.

2.1 General System Organization

In this study, we use a shared memory system as shown in Figure 1. The system includes a number of SMP nodes, one of which is shown in Figure 1(a). Each SMP node includes four processors with L1 and L2 caches, a remote cache, interleaved main memory, and a pipelined, split-transaction SMP bus. One to four SMP nodes are connected to each coherence controller (CC) as shown in Figure 1(b). Even with four SMP nodes connected to a single CC, the pin count of the CC chip can be kept well below 500.

The base CC architecture used in our study consists of a single protocol engine as shown in Figure 2. The PE contains a directory cache [9] which mitigates both the latency and bandwidth constraints of the directory. To address scalability concerns, we devised a new directory design called the Complete and Concise Remote (CCR) directory [6] which preserves the properties of a full map directory but is comparable in size with a sparse directory. The details of the CCR directory and the directory cache are beyond the scope of this paper and can be found in [13]. The PE uses a pending buffer (PB) to keep transient state for the protocol transactions in progress. The coherence protocol is implemented within the protocol engine in a hardwired finite state machine (FSM).

2.2 Multiple Protocol Engines

The base architecture can be extended to include multiple protocol engines as shown in Figure 3. One or more PEs handle requests to locations resident on remote memory (RPE) and others handle requests to locations in the local home memory (LPE).

The CC can have multiple LPEs and multiple RPEs based on memory interleaving. Each of the multiple RPEs or the multiple LPEs handles a non-overlapping region of physical memory. The multiple PEs can be assigned to interleaved memory regions by using address bits at any position in the physical address. However, low order memory

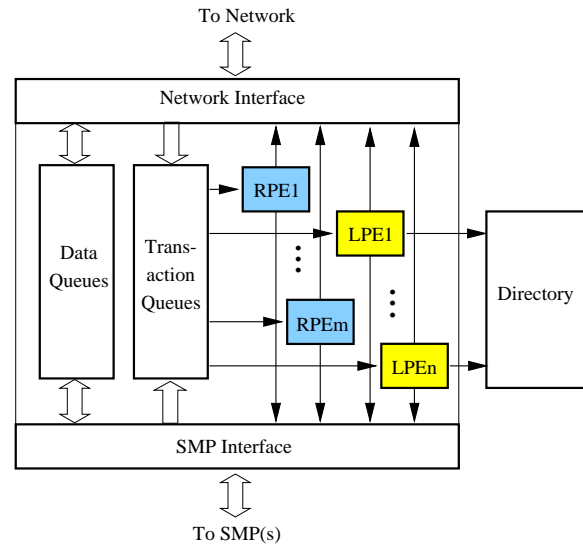
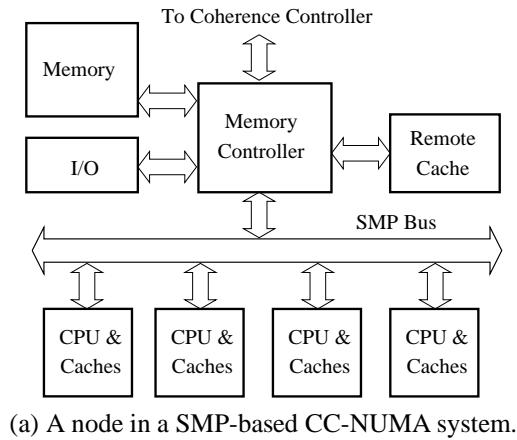


Figure 1. A shared memory system environment.

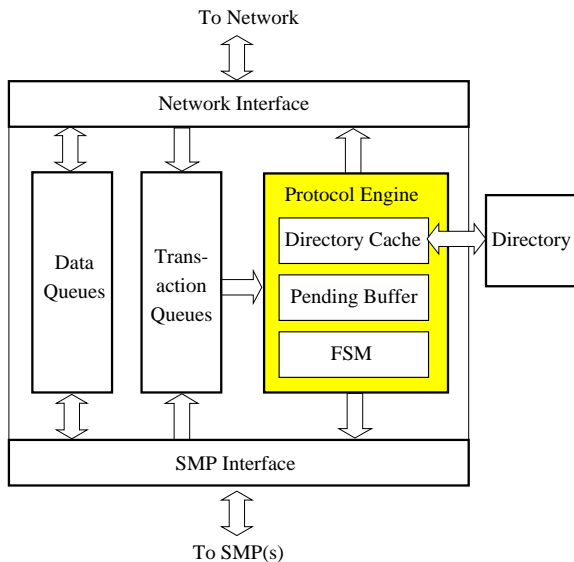


Figure 2. Base coherence controller architecture.

Figure 3. A coherence controller with multiple protocol engines.

interleaving assignment of the PEs intuitively results in less contention, as consecutive memory accesses would go to different PEs.

The RPEs do not access the directory. The LPEs may share the same directory, but each of them has a directory cache associated with it.

2.3 Pipelined Protocol Handling

The operations that process a coherence transaction, such as arbitration among incoming queues, pending buffer and directory lookup and directory updates, etc., would take several cycles if performed in a single pipe stage. Protocol processing operations can be broken into low latency pipe stages to increase coherence controller throughput. The pipeline in the protocol engine of our extended model breaks naturally into four stages as shown in Figure 4. In the first stage a transaction is fetched from an input queue. In the second stage the directory cache and pending buffer are read. If the transaction is a request, a lookup in both the directory cache and the pending buffer is performed. However, if the transaction is a response, only the pending buffer is read. In the third stage the transaction and the results of the directory cache and pending buffer lookup are presented to a protocol handling unit for execution. In the last stage the pending buffer and directory cache are updated and one or more new transactions are issued to output queues of the bus interface and/or network interface of the CC.

Each pipe stage occupies one coherence controller clock cycle. Assuming typical standard cell technology, timing analysis indicates that the four pipe stages are well bal-

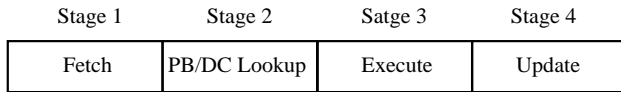


Figure 4. A protocol engine pipeline.

anced.

2.4 Split Request-Response Streams

Within each PE two independent protocol handling units can be used concurrently, one for protocol request transactions and one for response transactions [7]. The motivation for separating request and response handling arises from the observation that they access the pending buffer differently.

When a request is processed, an associative lookup of the pending buffer is needed for collision detection and a pending buffer entry is allocated if no collision is detected. The index of the allocated pending buffer entry is assigned to all downstream transactions generated as a result of processing the request. The transient directory state is also saved in the allocated pending buffer entry. Each response carries a pending buffer index that is used to retrieve the content of the corresponding pending buffer entry when it arrives back at the coherence controller from which the corresponding request was generated. No collision detection is necessary in the case of responses.

Noting this distinction, we separate the pending buffer into associative and indexed halves, as illustrated in Figure 5. The associative half (PBA, 'A' for address) contains the address of the transaction in progress, and a valid bit. The indexed half (PBC, 'C' for content) is where the state of the pending transaction is maintained. Supporting multiple ports on the non-associative PBC portion is relatively easy which allows simultaneous accesses by request and response handlers. The PBC is multi-ported with two read ports and two write ports: one read-write pair for the request stream and another pair for the response stream. The PBA is single-ported and supports associative lookup for requests.

The fetch unit retrieves a request or a response and dispatches it to the lookup unit. If it is a request, the lookup unit launches reads to the PBA and the directory cache. If there is a hit in the directory cache and no collision in the pending buffer, the transaction is forwarded to the request protocol handling unit for execution and an entry in the pending buffer is allocated. If there is a collision in the pending buffer, the content of the existing entry is forwarded to the request protocol handling unit to handle the collision.

If the directory cache misses (and there is no collision), the contents of the directory cache set that missed is passed along to the directory cache controller which determines

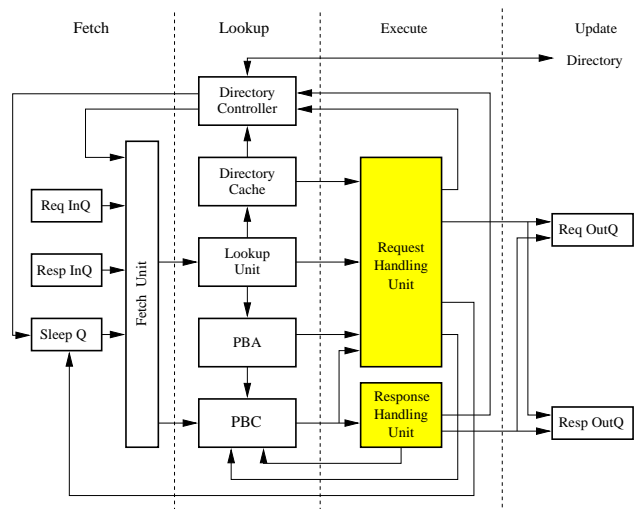


Figure 5. A protocol engine with split request and response streams.

if an eviction is required. If so, the directory controller launches a writeback to the directory. The request protocol handling unit is notified of the directory cache miss and responds by enqueueing the transaction in the sleep queue. When the response from the directory is received, the directory controller reactivates the transaction in the sleep queue. The fetch unit will eventually fetch the transaction from the sleep queue and reprocess it. The request protocol handling unit processes the transaction and updates the pending buffer. It also generates new request(s) and/or response(s) which it inserts into the output queues.

Response processing is somewhat less involved than request processing. As noted, in the lookup phase the PBC is read instead of the PBA. Since there is no directory cache lookup the transaction is passed deterministically to the response protocol handling unit for execution. The response protocol handling unit may generate new responses which it launches by inserting them into output queues. It also updates the PBC on a dedicated write port and schedules a directory cache update, if applicable, through the directory controller.

Remote protocol engines are very similar to local protocol engines, except that they do not contain a directory cache or a sleep queue. Consequently, they operate much more deterministically than local protocol engines. Also, the coherence protocol is very different for remote memory transactions than for local memory transactions. Therefore, separate local and remote protocol engines not only increase the degree of parallelism in the controller, but also simplify protocol engine design.

Device	Description
CPU	600MHz PowerPC
L1	16KB, 4-way, LRU, 64-byte lines
L2	1MB, 4-way, LRU, 64-byte lines
Remote Cache	16MB, 4-way, LRU
SMP Bus	150MHz 16 byte-wide, fully pipelined, split-transaction, separate address and data buses
CC	150MHz CC, each connecting to 1-4 SMP nodes

Table 1. Architectural parameters of an SMP node.

3 Experimental Results

In this section, we present simulation results of the effect of the three throughput-enhancing mechanisms on system performance. We investigate the synergy among these mechanisms and their effect on CC performance. First, we present the experimental methodology.

3.1 Experimental Methodology

We use execution-driven simulation based on a PowerPC version of the Augmint simulation toolkit [14]. Our simulator includes detailed contention models for SMP buses, memory controllers, interleaved memory banks, protocol engine pipelines, directory memory, and external point contention for the interconnection network. Table 1 details the architectural parameters of an SMP node. No-contention latencies of the system are shown in Table 2.

We use both commercial and scientific benchmarks in the performance experiments. The commercial workloads are traces of 1 billion instructions of TPCD and 400 million instruction of TPCC [16] runs on IBM DB2 [5] using memory-resident databases running on 32-processor systems. We collected traces of the commercial workloads using the COMPASS [12] environment. The scientific benchmarks are six applications from the SPLASH-2 suite [17] of parallel applications running on 64-processor systems. The data set sizes we use for the SPLASH-2 applications are as in Table 3. All benchmarks are written in C and compiled using the 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 ran all the applications with data sizes and systems sizes for which they achieve acceptable speedups (all except for LU achieve at least 50% efficiency).

The base architecture includes CCs without any of the mechanisms under study. Each CC is a single non-pipelined, combined request-response streams PE. We use

Event	Latency
L1 to CPU	1
L1 to L2	8
L2 to L1	4
L2 miss to address strobe on bus	8
Bus address strobe to bus response	28
Bus address strobe to start of cache-to-cache data response	36
Bus address strobe to next address strobe	8
Bus address strobe to start of data transfer from memory	40
Network point-to-point	52
CC issue request to bus	4
CC detect response from bus	4
CC issue network message	4
CC directory read (on-chip DC hit)	4
DRAM directory read	40
CC directory write	4
CC handler dispatch	4

Table 2. No-contention latencies in processor cycles.

Application	Type	Problem size
FFT	FFT computation	256K complex doubles
LU	Blocked dense linear algebra	512×512 matrix, 16x16 blocks
Ocean	Study of ocean movements	258×258 ocean grid
Radix	Radix sort	1M integer keys, radix 1K
Water-Spatial	Study of forces and potentials of water molecules in a 3-D grid	512 molecules
Water-Nsquared	$O(n^2)$ study of forces and potentials in water molecules	512 molecules
TPCD	Benchmark for measuring relational database performance of Decision Support applications.	1GB database
TPCC	Benchmark of On-Line Transaction Processing workload.	500MB database

Table 3. Benchmark types and data sets.

its performance as a comparison point for the performance of other configuration with more features. We vary the number of CCs in the system while keeping the total number of processors in the system constant, 64 processors for SPLASH-2 and 32 processors for TPC benchmarks, to vary the load on the CCs.

3.2 Multiple Protocol Engines

Table 4 shows the impact of multiple PEs on system performance with various system configurations (column 2). We use the expression L_xR_y to denote a CC with x LPEs and y RPEs. For a CC with x PEs that handle both local and remote transactions, we label it as L_xR_0 . The base configuration utilizes a single combined local-remote PE in each CC, as specified by the expression L_1R_0 . The most aggressive configuration in the table is denoted by L_2R_2 , i.e., each CC contains 2 LPEs and 2 RPEs.

The three rightmost columns show the improvement in application execution times on systems with different configurations of multiple PEs over the execution time of the base system with 1, 2, and 4 SMPs per CC, respectively. For applications with low communication rates such as Water-Spatial and LU, the CC architecture has little impact on overall system performance. However, for applications with high communication rates such as FFT and Ocean, using multiple PEs per CC can improve performance by as much as 51%. Of the commercial workloads, TPCC and TPCD improve by 26% and 17%, respectively.

In Table 5 we notice that multiple PEs are effective in reducing average CC occupancy (column 3) and average CC queuing delay (column 4). However, a significant reduction in CC occupancy and CC queue delay translates to a substantial improvement in execution time only for applications with high CC utilization (shown in column 5), as in the case of FFT, Ocean, Radix, TPCC, and TPCD. The remaining applications have low CC utilization as a small portion of memory accesses involve the CCs. Because they encounter little contention in the CCs, their execution times are less affected by the increase in CC parallelism.

The last column of Table 5 shows the average number of processed coherence operations per CC cycle (OPC). We see that for the applications with high CC utilization and long CC queuing delay, the OPC increases as PE parallelism increases. However, as more parallelism is added to the system, the OPC remains almost constant and the improvement in execution times diminishes.

Table 6 shows the load distribution between an LPE and an RPE. We observe that although there are more operations on the RPE on the average than on the LPE, the aggregate occupancy requirements (i.e bandwidth) of the LPE are more than those of the RPE as a result of directory access occupancy in the LPE, since RPEs do not access the direc-

Applications	CC Features	Improvement in Execution Time (%)		
		1 SMP	2 SMPs	4 SMPs
FFT	L_1R_1	12.58	27.18	31.46
	L_2R_0	11.48	26.70	34.05
	L_2R_1	15.08	34.54	43.68
	L_2R_2	15.66	37.90	51.13
LU	L_1R_1	0.23	1.00	1.42
	L_2R_0	0.15	1.54	1.47
	L_2R_1	0.26	2.00	2.02
	L_2R_2	0.28	2.03	2.13
Ocean	L_1R_1	6.10	24.17	26.38
	L_2R_0	5.65	22.25	27.82
	L_2R_1	7.21	24.72	26.92
	L_2R_2	7.99	31.23	41.84
Radix	L_1R_1	3.46	6.87	11.96
	L_2R_0	11.55	11.33	14.48
	L_2R_1	12.30	12.84	17.67
	L_2R_2	12.70	13.95	19.74
TPCC	L_1R_1	5.40	14.40	22.04
	L_2R_0	5.32	13.88	22.73
	L_2R_1	6.09	16.35	25.69
	L_2R_2	6.23	17.06	26.36
TPCD	L_1R_1	3.25	11.93	14.53
	L_2R_0	3.22	11.75	14.89
	L_2R_1	4.08	13.48	16.38
	L_2R_2	4.07	14.13	16.92
Water-Nsq	L_1R_1	1.01	5.20	8.11
	L_2R_0	0.97	4.66	7.32
	L_2R_1	1.21	5.38	8.81
	L_2R_2	1.39	5.73	10.36
Water-Sp	L_1R_1	0.59	1.02	1.06
	L_2R_0	0.56	0.94	0.95
	L_2R_1	0.66	1.03	1.03
	L_2R_2	0.68	1.38	1.42

Table 4. Effect of multiple protocol engines on execution time (normalized to L_1R_0). Column 2 uses L_xR_y to denote a coherence controller with x LPEs and y RPEs.

tory. From Table 4 we find that only the benchmarks with high LPE to RPE occupancy ratio, benefited from using 2 LPEs and 1 RPE over 1 LPE and 1 RPE. For instance, FFT and Radix have high LPE-RPE occupancy ratios and thus, are able to take advantage of the additional parallelism provided by the L_2R_1 configuration. On the other hand, Ocean has the same occupancy in the LPE and RPE and thus, unable to take advantage of the extra LPE in the L_2R_1 configuration.

It is interesting to note that the configuration with 1 LPE

Apps	CC features	Reduction in CC occu. (%)	CC queuing time (CC cycles)	CC util (%)	OPC $\times 100$
FFT	L ₁ R ₀	-	51.34	76	20.2
	L ₁ R ₁	40.55	25.90	66	29.4
	L ₂ R ₀	43.63	24.19	65	30.6
	L ₂ R ₁	55.96	16.62	60	35.8
	L ₂ R ₂	65.79	10.71	54	41.3
LU	L ₁ R ₀	-	15.94	8	2.2
	L ₁ R ₁	14.74	7.83	7	2.3
	L ₂ R ₀	14.95	7.18	7	2.3
	L ₂ R ₁	20.83	4.54	6	2.3
	L ₂ R ₂	23.22	3.73	6	2.3
Ocean	L ₁ R ₀	-	23.23	64	18.3
	L ₁ R ₁	27.36	11.11	63	25.0
	L ₂ R ₀	24.04	9.68	67	29.7
	L ₂ R ₁	25.37	8.96	65	29.2
	L ₂ R ₂	41.97	4.03	64	36.9
Radix	L ₁ R ₀	-	35.23	42	10.2
	L ₁ R ₁	23.32	18.36	37	11.6
	L ₂ R ₀	27.40	14.46	36	12.0
	L ₂ R ₁	34.38	9.87	33	12.5
	L ₂ R ₂	39.04	7.07	32	12.8
TPCC	L ₁ R ₀	-	15.32	85	23.2
	L ₁ R ₁	24.90	4.07	82	30.7
	L ₂ R ₀	25.99	3.78	81	31.1
	L ₂ R ₁	30.99	2.31	79	32.5
	L ₂ R ₂	33.68	1.85	77	32.9
TPCD	L ₁ R ₀	-	10.85	81	22.5
	L ₁ R ₁	23.01	2.79	73	27.0
	L ₂ R ₀	23.58	2.65	73	27.1
	L ₂ R ₁	27.89	1.86	70	27.7
	L ₂ R ₂	30.62	1.53	68	27.9
Water-Nsq	L ₁ R ₀	-	19.80	23	6.7
	L ₁ R ₁	32.83	8.39	17	7.1
	L ₂ R ₀	31.72	8.71	17	7.1
	L ₂ R ₁	36.75	5.91	16	7.4
	L ₂ R ₂	45.10	3.42	14	7.4
Water-Sp	L ₁ R ₀	-	20.34	7	2.0
	L ₁ R ₁	25.14	8.50	5	2.0
	L ₂ R ₀	26.81	8.07	5	2.0
	L ₂ R ₁	28.03	7.32	5	2.0
	L ₂ R ₂	39.44	3.13	4	2.1

Table 5. Effect of multiple protocol engines on CC performance on a system with 4 SMPs per CC.

and 1 RPE (L₁R₁) usually provides better performance than that with 2 PEs (L₂R₀), for systems with 1 or 2 SMPs per CC. The former configuration provides an exclusive RPE for memory accesses to remote addresses. The RPE encounters less stalls because it does not access the directory. Table 6 shows that the number of coherence transactions destined for remote nodes outnumbers that for the local node. With less stalls and more transactions destined for

Applications	Number of Operations (LPE:RPE)	Occupancy (LPE:RPE)
FFT	0.86	1.36
LU	0.92	1.26
Ocean	0.78	1.08
Radix	0.79	1.45
TPCC	1.00	1.44
TPCD	1.00	1.40
Water-Nsq	0.81	1.08
Water-Sp	0.80	1.06

Table 6. Load ratio between LPE and RPE on a system with 4 SMPs per CC.

remote nodes, the RPE provides higher throughput while maintaining load balance with the LPE. The L₂R₀ configuration, on the other hand, interleaves all transactions among two PEs that process transactions for both local and remote addresses. As such, there is a higher chance that a local-address transaction missing in the directory cache stalls a remote-address transaction.

On the system with 4 SMPs per CC, Table 4 shows that the L₂R₀ configuration performs better than the L₁R₁ one. As more SMPs are attached to each CC while keeping the total number of processors constant, there are fewer nodes in the system and thus there are more accesses to the local memory modules. L₂R₀ provides two LPEs to process traffic destined for local node and improves the throughput. Although the transactions destined for remote nodes are more likely to be delayed by directory cache miss stalls, the balance in traffic to the PEs improves the overall execution time. On the other hand, the L₁R₁ configuration suffers load imbalance because it has only a single LPE to handle the increased traffic to local addresses.

3.3 Pipelined Protocol Handling Units

Table 7 compares the performance of systems with coherence controller with pipelined PEs against the performance of the base system. The last three columns show the improvement in execution time of pipelined-PE systems over that of the base system, with 1, 2, and 4 SMPs per CC, respectively. We observe that using pipelining can improve execution time by as much as 39%, which is comparable to the results with 2 PEs.

Table 8 shows similar trends in average CC occupancy and average CC queuing time reduction as multiple-PE systems. Column 2 uses P_u to denote u stages in the pipeline of each PE. P_1 indicates that a PE has a single non-pipeline stage that can handle only a single coherence transaction until it exits the PE. P_4 indicates that the PE has 4 pipeline stages with a throughput of 1 coherence transaction per CC

Applications	Improvement in Execution Time (%)		
	1 SMP	2 SMPs	4 SMPs
FFT	8.81	28.28	39.25
LU	-0.18	3.66	1.83
Ocean	3.54	25.69	34.46
Radix	6.01	7.18	12.16
TPCC	2.99	13.72	24.07
TPCD	1.91	11.17	14.64
Water-Nsq	-0.28	4.56	8.84
Water-Sp	-0.04	0.57	0.97

Table 7. Effect of pipelined protocol engines on execution time.

cycle when the pipeline is full. In the the P_1 design, each transaction takes (assuming a directory cache hit) three CC cycles to go through the PE. That is, pipelining increases PE latency from 3 to 4 cycles, but improves throughput from one operation per 3 cycles to one operation per cycle.

Similar to multiple-PE systems, applications that have high CC occupancy (column 3) combined with high CC utilization (column 5), show significant improvement in execution time. For these applications, pipelining is effective at increasing throughput. Indeed, the last column shows that the number of processed coherence operations per cycle increased significantly for FFT, Ocean, and TPCC. The OPC numbers are similar to those of multiple-PE configurations, indicating that pipelined-PE systems are as effective as multiple-PE systems in enhancing coherence throughput. Note that the directory cache is essential to minimize bubbles in the pipeline by keeping the average directory access time small.

On a system with 1 SMP per CC, LU, Water-Spatial and Water-Nsq do not show improvement with pipelining. These applications have low communication and do not require more aggressive PEs. Their execution times increase slightly, because they incur one more cycle in the pipelined design than the non-pipelined design without benefiting from the throughput-enhancing effect of pipelining.

3.4 Split Request-Response Streams

Table 9 shows the improvement in execution time of a system with split-stream protocol engines over the execution time of the base system with 1-SMP, 2-SMP, and 4-SMP CCs. We observe that using split-stream PEs can improve execution time by as much as 35%, which is comparable to the 34% improvement in using 2 PEs and 39% with pipelined PEs.

As in the case of multiple and pipelined PEs, split-stream PEs also reduce average CC occupancy and average CC queuing time. Column 2 of Table 10 uses S_1 to denote that

Apps	CC features	Reduction in CC occu. (%)	CC queuing time (CC cycles)	CC util (%)	OPC $\times 100$
FFT	P_1	-	51.34	76	20.2
	P_4	50.75	18.10	62	33.2
LU	P_1	-	15.94	8	2.2
	P_4	14.32	3.78	7	2.3
Ocean	P_1	-	23.23	64	18.3
	P_4	30.13	5.36	68	34.0
Radix	P_1	-	35.23	42	10.2
	P_4	25.67	15.04	36	11.7
TPCC	P_1	-	15.32	85	23.2
	P_4	28.20	1.61	80	32.1
TPCD	P_1	-	10.85	81	22.5
	P_4	23.06	1.30	73	27.2
Water-Nsq	P_1	-	19.80	23	6.7
	P_4	35.57	3.73	16	7.5
Water-Sp	P_1	-	20.34	7	2.0
	P_4	27.37	4.24	5	2.0

Table 8. Effect of pipelined protocol engines on CC performance (averages) on a system with 4 SMPs per CC. P_1 and P_4 in column 2 represent non-pipelined and pipelined protocol engines, respectively.

Applications	Improvement in Execution Time (%)		
	1 SMP	2 SMPs	4 SMPs
FFT	11.09	26.45	34.64
LU	0.32	3.90	1.82
Ocean	5.11	23.16	28.91
Radix	8.58	8.22	13.74
TPCC	5.56	14.54	24.32
TPCD	4.03	12.47	15.55
Water-Nsq	1.03	4.91	8.28
Water-Sp	0.68	1.24	0.95

Table 9. Effect of split request-response streams on execution time.

each PE has 1 handling unit in each PE for both requests and responses, and S_2 to indicate that there are 2 handling units, one for requests and the other one for responses. Column 3 shows that the reduction in CC occupancy is more than 44% for all applications and up to 58% for FFT. Similarly, column 4 shows that the response time of applications for the split-stream-PE systems is much lower than that of the base system. However, the overall execution times improve significantly only for high-communication applications such as FFT, Ocean, TPCD, and TPCC that also have high CC utilization (column 5).

One issue with having two different handling units for requests and responses is that whether they are balanced in

Apps	CC features	Reduction in CC occu. (%)	CC queuing time (CC cycles)	CC util (%)	OPC $\times 100$
FFT	S ₁	-	51.34	76	20.2
	S ₂	58.24	24.18	49	30.9
LU	S ₁	-	15.94	8	2.2
	S ₂	41.61	5.96	5	2.3
Ocean	S ₁	-	23.23	64	18.3
	S ₂	46.25	9.57	48	30.1
Radix	S ₁	-	35.23	42	10.2
	S ₂	46.93	15.95	26	11.8
TPCC	S ₁	-	15.32	85	23.2
	S ₂	64.42	3.02	40	31.8
TPCD	S ₁	-	10.85	81	22.5
	S ₂	62.86	2.20	36	27.5
Water-Nsq	S ₁	-	19.80	23	6.7
	S ₂	50.90	7.94	12	7.2
Water-Sp	S ₁	-	20.34	7	2.0
	S ₂	44.32	10.09	4	2.0

Table 10. Effect of split request-response streams on CC performance (averages) on a system with 4 SMPs per CC. S_v represents a protocol engine with v handling units.

terms of number of coherence operations, occupancy, response time, and the interarrival time of coherence messages at these units. Table 11 compares the request unit (RQ) with the response unit (RS). Columns 2 and 3 show that the number of requests sent to both units are comparable. Columns 4 and 5 show that request and response units have similar occupancy. The occupancy at the request unit is slightly higher than that of the response unit for most applications because requests usually look up directory caches for directory entries. Similarly, the interarrival times of messages at both units (column 6 and 7, respectively) are also very close. In short, Table 11 indicates that the two units are balanced.

3.5 Synergy among Mechanisms

We showed that multiple PEs, pipelining, and split-streams are equally effective at enhancing CC throughput, when applied individually. Now, we discuss the synergy among the three mechanisms on system performance. Table 12 shows the improvement in execution time of various combinations of mechanisms relative to the execution time of the base CC, on a system with 4 SMPs per CC. We observe that combining two mechanisms improves performance beyond that achieved with a single mechanism. For example, the execution times of FFT improve by at least 31% by employing a single mechanism. With a combina-

Apps	Num. of operations (million)		Occupancy (million CC cycles)		Inter-arrival time (in CC cycles)	
	RQ	RS	RQ	RS	RQ	RS
FFT	1.03	0.98	4.15	3.54	6.64	6.33
LU	0.22	0.22	0.78	0.75	87.84	88.33
Ocean	3.58	3.29	12.12	12.10	6.93	6.36
Radix	0.55	0.55	2.62	1.90	17.12	16.91
TPCC	2.63	2.50	19.07	18.39	6.46	6.14
TPCD	2.83	2.83	21.05	19.80	7.28	7.29
Water-Nsq	0.32	0.30	1.10	1.05	29.02	26.94
Water-Sp	0.06	0.06	0.22	0.20	103.47	92.90

Table 11. Request and response handling units.

tion of two mechanism, its execution time improves by at least 50%. In particular, the improvements are 52% with pipelined and split-stream PEs ($L_1R_0P_4S_2$), 50% with multiple split-stream PEs ($L_1R_1P_1S_2$), and 52% with multiple pipelined PEs ($L_1R_1P_4S_1$).

However, applying more features yields diminishing returns. For TPCD and TPCC, applying pipelining on top of other features degrades performance as the throughput of the CC becomes adequate. In these cases pipelining only adds to the latency.

The simplest and least expensive path to achieving most of the performance benefit is the configuration with 1 LPE and 1 RPE, with split request-response streams ($L_1R_1P_1S_2$). Separating the FSMs of the LPE and RPE and the response and request streams simplifies the design by reducing the number of input/output queues and wire channels between the queues and the PEs. Separating the RPE from the LPE allows responses to proceed deterministically without stalls through the RPE. Similarly, separating the response stream from the request stream allows responses to proceed without stalls. Split streams provides efficient utilization of PE resources such as the pending buffer, queues, and the directory cache. Our results show that using the $L_1R_1P_1S_2$ configuration on the average achieves 89% of the performance improvement obtained by using the best performing configuration. Nearly all of the remaining 11% can be achieved most simply with the configuration with 2 LPEs and 2 RPEs, with split request-response streams ($L_2R_2P_1S_2$).

3.6 Number of SMPs per Coherence Controller

In this subsection we relate the effect of the three mechanisms on a system with one SMP per CC to that of systems with two and four SMPs per CC.

Table 13 shows the improvements in execution time for systems with various combinations of CC architectural features relative to the execution time of the base configura-

FFT		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	31.46	51.13
	S ₂	34.64	50.37	59.61
P ₄	S ₁	39.25	52.34	59.22
	S ₂	51.95	58.24	60.80
LU		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	1.42	2.13
	S ₂	1.82	2.33	2.42
P ₄	S ₁	1.83	2.15	2.24
	S ₂	2.21	2.26	2.26
Ocean		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	26.38	41.84
	S ₂	28.91	42.46	46.55
P ₄	S ₁	34.46	43.61	45.02
	S ₂	43.18	44.98	45.28
Radix		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	11.96	19.74
	S ₂	13.74	18.04	22.31
P ₄	S ₁	12.16	17.15	20.76
	S ₂	16.86	19.20	21.28
TPCC		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	22.04	26.36
	S ₂	24.32	26.81	27.34
P ₄	S ₁	24.07	25.35	25.51
	S ₂	25.18	25.63	25.72
TPCD		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	14.53	16.92
	S ₂	15.55	17.16	17.45
P ₄	S ₁	14.64	15.35	15.53
	S ₂	15.37	15.45	15.57
Water-Nsq		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	8.11	10.36
	S ₂	8.28	10.85	11.16
P ₄	S ₁	8.84	9.79	9.73
	S ₂	9.64	9.89	9.73
Water-Sp		L ₁ R ₀	L ₁ R ₁	L ₂ R ₂
P ₁	S ₁	-	1.06	1.42
	S ₂	0.95	1.42	1.51
P ₄	S ₁	0.97	1.00	1.12
	S ₂	1.21	1.12	1.19

Table 12. Improvement in execution time over base CC configuration.

tion on a system with one SMP per CC. The second column uses $L_x R_y P_n S_m$ to denote a system with x LPEs, y RPE, n pipeline stages, and m request-response streams. We concentrate on FFT and Ocean as representatives of communication-intensive applications that stress the CCs. For systems with 2 SMPs/CC and 4 SMP/CCs, we notice that the case of $L_1 R_0 P_1 S_1$ underperforms the same configuration on the base system, as there are fewer CCs in the system for nearly as many concurrent coherence operations. For these high-communication applications, the high contention at the CCs negates the benefits of the improved memory locality. However, applying any of the parallel fea-

Applications	CC Features	Improvement in Execution Time (%)		
		1 SMP	2 SMPs	4 SMPs
FFT	L ₁ R ₀ P ₁ S ₁	0.00	-14.61	-52.13
	L ₁ R ₁ P ₁ S ₁	12.58	16.54	-4.27
	L ₁ R ₀ P ₄ S ₁	8.81	17.80	7.58
	L ₁ R ₀ P ₁ S ₂	11.09	15.70	0.57
	L ₁ R ₁ P ₄ S ₁	13.43	27.32	27.50
	L ₁ R ₁ P ₁ S ₂	16.34	28.96	24.50
	L ₁ R ₀ P ₄ S ₂	12.48	25.99	26.90
	L ₂ R ₂ P ₁ S ₁	15.66	28.82	25.65
	L ₁ R ₁ P ₄ S ₂	14.81	30.20	36.47
	L ₂ R ₂ P ₄ S ₂	15.08	31.24	40.36
Ocean	L ₁ R ₀ P ₁ S ₁	0.00	-6.76	-13.45
	L ₁ R ₁ P ₁ S ₁	6.10	19.05	16.49
	L ₁ R ₀ P ₄ S ₁	3.54	20.67	25.65
	L ₁ R ₀ P ₁ S ₂	5.11	17.97	19.35
	L ₁ R ₁ P ₄ S ₁	5.84	25.16	36.03
	L ₁ R ₁ P ₁ S ₂	7.09	26.54	34.72
	L ₁ R ₀ P ₄ S ₂	5.61	24.84	35.53
	L ₂ R ₂ P ₁ S ₁	7.99	26.58	34.02
	L ₁ R ₁ P ₄ S ₂	6.00	26.19	37.58
	L ₂ R ₂ P ₄ S ₂	6.15	26.87	37.92

Table 13. Improvement in execution time of FFT and Ocean over the base configuration with 1 SMP per CC and unoptimized PEs.

tures allows the system to maintain or even exceed its performance but with fewer CCs, thus reducing system cost. As shown on the last column of the table, FFT improves by 7.6% on a system with 4 SMPs/CC with pipelined PEs over the base system with 1 SMP/CC. For Ocean on the 4 SMPs/CC system, pipelined PEs reduce the execution time significantly over the base system, improving it by 26%.

For low communication applications, even with fewer CCs, there are not enough concurrent coherence activities to increase contention at the CCs. The execution times of these applications actually improve over the base case because of the increased memory locality.

4 Conclusions

In this paper we study three approaches to alleviating the contention in hardwired coherence controllers, namely, multiple protocol engines, split request-response streams, and pipelined protocol engines.

We explore the design space using both technical and commercial workloads. The results show that each approach is highly effective at reducing controller occupancy by as much as 28% and improving execution time by as much as 16%, for applications with high communication bandwidth requirement on a system with one SMP node

per CC. A combination of mechanisms further reduces controller occupancy and execution time by as much as 32% and 17%, respectively. However, applying more aggressive parallelism yields diminishing returns as the throughput of the CC becomes adequate.

The effect becomes more significant as multiple SMPs are connected to each CC. With 4 SMP nodes per CC, the occupancy improves by as much as 66% and execution time by 51%. Using combinations of the three mechanisms continues to bring significant performance gain for high-communication applications. The compound effect of the three mechanisms is a reduction in controller occupancy and execution time by as much as 78% and 61%, respectively.

By connecting multiple SMPs to each CC and applying the throughput-enhancing mechanisms, we can reduce the number of CC nodes in the system and the size of the interconnection network. This reduces the cost of the system with a small increase in CC complexity as multiple PEs, pipeline, and split-streams are employed. Our results show that applying any of the parallel features to a system with two or four SMPs per CC allows maintaining or even exceeding the performance of a system with one SMP per CC but without parallel features. That is, we can reduce the cost of the CCs by up to 75% while improving performance by using the parallel features.

For future high-throughput coherence controller designs, we recommend that high throughput features be added in the following order: separate LPE and RPE controllers, split request/response streams, pipelining, multiple memory interleaved protocol engines. Separating the FSMs of the LPE and RPE and the response and request streams simplifies the design by reducing the number of input/output queues and wiring channels between the queues and protocol engines. Furthermore, this design eliminates stalls in the RPE and the response stream in the LPE. Split streams and pipelining provide efficient utilization of protocol engine resources such as the pending buffer, queues and directory cache.

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 and D. Wood. Parallel Dispatch Queue: A Queue-Based Programming Abstraction to Parallelize Fine-Grain Communication Protocols. In *Proceedings of High-Performance Computer Architecture*, January 1999.
- [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. Roseblum, 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, Occupance, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical report, Stanford University, January 1995.
- [5] IBM Corporation. DATABASE 2 Information and Concepts Guide for Common Servers Version 2, Document number S20H-4664-00, May 1995.
- [6] D. Joseph, M. Michael, and A. Nanda. Complete and Concise Remote (CCR) Directory. Patent pending, 1999.
- [7] D. Joseph, M. Michael, and A. Nanda. Split Multiported Pending Buffer. Patent pending, 1999.
- [8] 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.
- [9] M. M. Michael and A. K. Nanda. Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors. In *Proceedings of High-Performance Computer Architecture*, pages 142–151, January 1999.
- [10] M. M. Michael, A. K. Nanda, and B.-H. Lim. Coherence Controller Architectures for Scalable Shared-Memory Multiprocessors. *IEEE Trans. Computers*, 48(2):63–79, February 1999.
- [11] M. M. Michael, A. K. Nanda, B.-H. Lim, and M. L. Scott. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the 24rd International Symposium on Computer Architecture*, pages 219–228, June 1997.
- [12] A. Nanda, Y. Hu, M. Ohara, C. Benveniste, M. Giampapa, and M. Michael. The Design of COMPASS: An Execution Driven Simulator for Commercial Applications Running on Shared Memory Multiprocessors. In *Proceedings of the International Parallel Processing Symposium*, 1998.
- [13] A. Nanda, A.-T. Nguyen, M. Michael, and D. Joseph. HighT: a High Throughput Coherence Controller Microarchitecture for Scalable Shared Memory Multiprocessors. Technical report, IBM TJ Watson Research Center, 1999.
- [14] A.-T. Nguyen, M. M. Michael, A. D. 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.
- [15] A. Nowatzky, 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.
- [16] Transaction Processing Performance Council. TPC Benchmark D and TPC Benchmark C: Standard Specifications.
- [17] 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.