Design Trade-Offs in High-Throughput Coherence Controllers *

Anthony-Trung Nguyen
Microprocessor Research Labs
Intel Corporation
Santa Clara, CA 95052
anthony.d.nguyen@intel.com

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
torrellas@cs.uiuc.edu

Abstract

Recent research shows that the high occupancy of Coherence Controllers (CCs) is a major performance bottleneck in scalable shared-memory multiprocessors. In this paper, we propose to take microarchitectural enhancements used for microprocessors and apply them to improve the throughput of hardwired CCs. These enhancements are CC support for nonblocking execution, early fetches of directory and L3 information, and superpipelining. Nonblocking execution in the CC reduces stalls by processing subsequent coherence transactions in the presence of misses in the directory cache and tag cache. Early fetching in the CC hides misses in the directory and tag caches and, therefore, also removes stalls. Finally, superpipelining in the CC increases its processing bandwidth. These supports all serve to increase the overall throughput of CCs and improve overall system performance.

Using both SPLASH-2 and parallelized SPEC95 applications on detailed simulation models, we show that CCs that support nonblocking execution and superpipelining boost the performance of machines substantially. With these CCs, a 64-processor machine with four nodes of four SMPs per node runs on average 3.56 times faster than if it used conventional CCs. In addition, the machine runs about as fast as a more costly 64-processor machine with sixteen nodes of one SMP per node and the same advanced CCs. This is despite using much less network, chassis, and node hardware. Consequently, with our proposed advanced CCs, we can reduce the system cost significantly without affecting performance.

1 Introduction

Scalable shared-memory multiprocessors can significantly increase performance over uniprocessor systems by coordinating work among multiple processor nodes. A key feature of cache-coherent scalable shared-memory systems is a Coherence Controller (CC) at each node, which ensures that cached data are kept coherent.

Past research [6, 13] shows that the occupancy of CCs can be a performance bottleneck for applications with high communication requirements. A high CC occupancy hinders performance by inducing contention and reducing CC throughput. As microprocessors used in multiprocessors become more aggressive by generating and tolerating an increasing number of outstanding memory references, the throughput demands on CCs increase. Moreover, scalable systems that use SMP nodes as building blocks create even more demand on CC throughput because all the processors in a node share the same CC. Nowadays, server vendors such as IBM and Sun produce SMP systems as wide as 32-64 processors [7, 17], which can be used as nodes in a scalable shared-memory multiprocessor.

We propose to take microarchitectural enhancements used in microprocessors and apply them to hardwired CCs. We analyze the differences between microprocessors and CCs. Based on the differences, we select three microprocessor techniques and adapt them to enhance the throughput in hardwired CCs. The three techniques are: nonblocking execution in the CC, early fetches of directory and L3 information, and superpipelining in the CC. Nonblocking execution in the CC reduces stalls by processing subsequent coherence transactions when current transactions suffer misses in directory caches or L3 tag caches. Early fetching hides stall latency by fetching necessary information from directory caches or L3 tag caches early. Finally, superpipelining subdivides the pipeline of the CCs into more, finer stages to increase CC bandwidth. Overall, the increased parallelism and overlapping of coherence operations enabled by these enhancements serves to increase the overall throughput of CCs.

To further enhance CC throughput, these three optimization techniques can be combined with previously proposed CC optimizations such as multiple protocol engines, pipelining, and split request-response streams [1, 2, 8, 9, 11, 13, 14, 15]. Multiple protocol engines replicate the core processing engine of a CC to increase concurrency. Hardwired CCs are pipelined to increase their bandwidth. Split request-response streams allocate dedicated hardware resources in a CC to process a request and a response in parallel.

We evaluate combinations of these two sets of CC enhancements using both SPLASH-2 and parallelized SPEC95 applications. We use detailed simulation models of multiprocessors with state-of-the-art superscalar processors. Our results show that using CCs that combine nonblocking execution and superpipelining boosts the performance of machines significantly. With these CCs, a 64-processor machine with four nodes of four SMPs per node runs on average 3.56 times faster than if it used conventional CCs. Moreover, the machine is as fast as a more costly 64-processor machine with sixteen nodes of one SMP per node and the same advanced CCs. This is despite using much less network, chassis, and node hardware. Consequently, we reduce the system cost significantly without affecting performance.

The rest of the paper is organized as follows: Section 2 describes the baseline CC architecture and previously proposed CC optimiza-

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel.

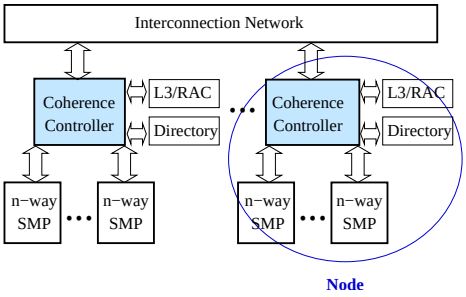
tions; Section 3 details our three new mechanisms for enhancing CC throughput; Section 4 presents our experimental methodology; Section 5 evaluates CC design trade-offs; Section 6 discusses related work; and finally, Section 7 concludes.

2 Baseline Coherence Controller Architecture

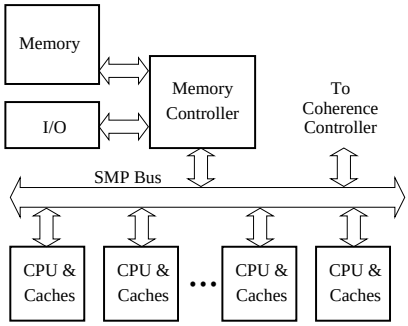
2.1 General System Organization

We use a cache-coherent, scalable shared-memory multiprocessor (Figure 1(a)) where each node contains a CC connected to several n-way SMPs. Each SMP includes n superscalar processors with L1 and L2 caches, interleaved main memory, and a pipelined, split-transaction SMP bus (Figure 1(b)). Cache coherence is maintained by a bus-based snoopy protocol within an SMP and enforced by CCs using a directory-based cache coherence protocol across the machine. Our directory-based protocol uses an invalidation-based approach.

Each CC also connects to either a Remote Access Cache (RAC) [10] or an L3 cache. A RAC keeps recently-accessed copies of remote memory lines. An L3 cache keeps both local and remote memory lines. If the RAC or L3 can satisfy a local request to a remote memory line, the request does not need to traverse the network in order to fetch that memory line from the home node.



(a) A scalable shared-memory multiprocessor where each node contains several n-way SMPs.



(b) N-way SMP. Several such SMPs may form a single node of the machine.

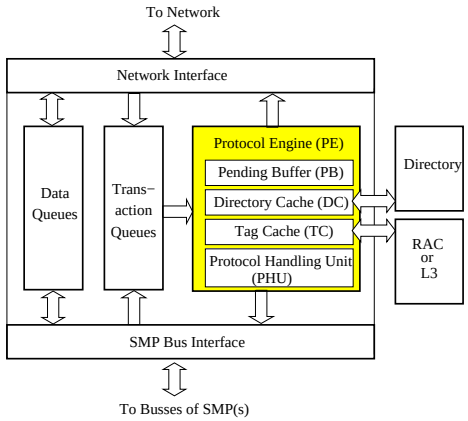
Figure 1. Architecture considered.

2.2 Coherence Controller Architecture

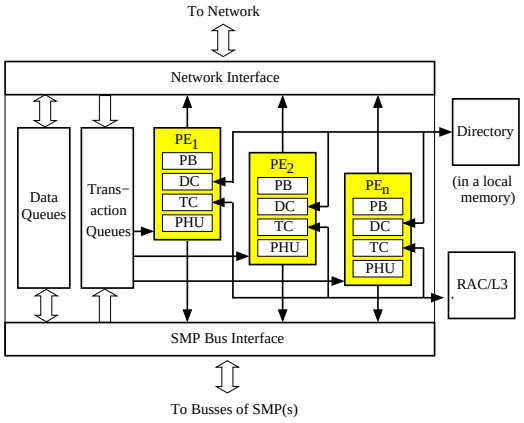
Our baseline CC architecture consists of a network interface, an SMP bus interface, queues for data and coherence transactions, and a *Protocol Engine* (PE) as shown in Figure 2(a). This PE is a self-contained protocol processing unit that fetches coherence transactions from transaction queues for processing. It consists of four major components: a directory cache, a tag cache, a pending buffer, and a protocol handling unit FSM (Figure 2(a)).

For coherence operations to local memory lines, the PE needs to access directory entries that correspond to these lines. Since the directory maintains state for all the local memory lines, the directory is large and its entries are stored in main memory. The PE uses a *Directory Cache* (DC) [3, 12] to mitigate both the latency and bandwidth constraints of the directory. Our baseline design uses a 4-way, 16K-entry directory cache per node. Each line in the directory cache contains state for 16 memory lines.

The *Tag Cache* (TC) in Figure 2(a) is a new structure that we propose to reduce the latency of some RAC or L3 accesses. The TC caches part of the tag state of the RAC or L3. If a request hits in the TC and finds the entry invalid, there is no need to access the RAC or L3.



(a) Base coherence controller.



(b) A CC with multiple protocol engines.

Figure 2. Coherence controller architecture.

The PE uses a *Pending Buffer* (PB) to keep transient states for protocol transactions in progress. Since a transaction may be composed of multiple request messages and multiple response messages, this buffer provides a hardware context for an ongoing transaction. A hardware context contains information such as the re-

quester ID, transaction type, memory line address, directory state, and the number of outstanding invalidation acknowledgments that will be needed by subsequent response messages.

Finally, a hardwired FSM within the PE called the *Protocol Handling Unit* (PHU) implements the directory-based cache coherence protocol. This unit consists of protocol handlers that are invoked by the PE to process specific coherence transactions.

2.3 Previously Proposed Optimizations

Prior studies proposed optimized CC designs that included multiple PEs per CC, pipelined PEs, and split request-response streams [1, 2, 8, 9, 11, 13, 14, 15]. In this section, we briefly describe these CC organizations.

2.3.1 Multiple Protocol Engines

The multiple-PE optimization enhances the baseline CC architecture of Figure 2(a) to include multiple PEs (Figure 2(b)). Each PE handles protocol transactions directed to a different set of memory line addresses. Consequently, PEs work fairly independently of each other and share minimal resources with each other.

One way to divide the memory line addresses between PEs is to use the low-order bits of the line address. Such a fine-grain interleaving assignment is simple and provides good load balance. Another, complementary way of dividing the addresses is between local and remote addresses. In this case, some PEs called *Local PEs* process transactions to local addresses, while other PEs called *Remote PEs* handle transactions to locations in remote memory. The rationale behind this approach is that the coherence protocol for remote memory transactions is very different from the one for local memory transactions. Consequently, this way to partition PEs not only increases the degree of parallelism in the CC, but also simplifies PE design.

2.3.2 Pipelined Protocol Engines

In a typical design, a PE takes several cycles to process a transaction. First, the PE arbitrates among incoming queues to select the transaction to process. Second, it uses the address of the memory line being accessed in the transaction to allocate a pending buffer entry and to look up the directory cache. Third, it invokes the corresponding protocol handler to process the transaction. Finally, it updates the directory state and pending buffer, and sends coherence messages to the SMP busses or network. To increase CC throughput, PE operations can be decomposed into multiple low-latency pipeline stages as in Figure 3. For example, the pipeline in the figure has four stages: fetch, lookup, execute, and writeback. Each pipeline stage takes one CC clock cycle.

2.3.3 Split Request-Response Streams

Unlike the multiple-PE optimization, which replicates entire PEs, this optimization allocates hardware resources within a single PE to provide two streams of concurrent execution. More specifically, two independent protocol handling units are provided: one for *Request* protocol messages and one for *Response* protocol messages. This optimization allows a PE to process a request message from one transaction and a response message from another transaction concurrently. A coherence transaction can be composed of

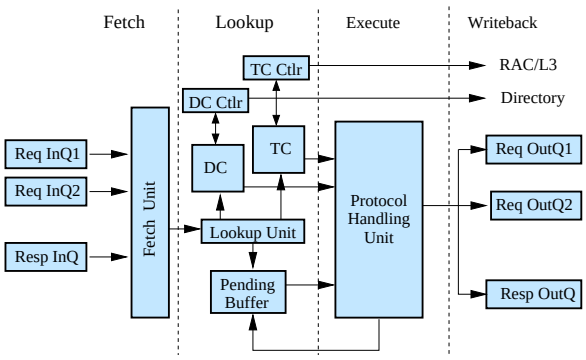


Figure 3. A protocol engine pipeline.

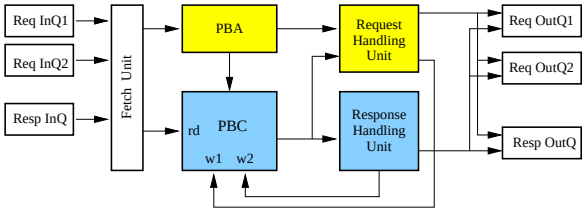


Figure 4. A protocol engine with split request and response streams.

multiple request messages and multiple response messages. One difference between a request and response is that a request message accesses either the directory cache or tag cache, while a response message does not. Hence, only request messages stall in the PE when they miss in either cache.

To facilitate concurrent accesses to the pending buffer for both streams, the pending buffer is separated into an associative half for requests and an indexed half for responses (Figure 4). An entry in the associative half (PBA, 'A' for address) contains the address of a transaction in progress and a valid bit. The indexed half (PBC, 'C' for content) maintains in each entry the state of a pending transaction.

When a request is processed, an associative lookup of the PBA is needed to detect a collision (i.e., multiple accesses to the same memory line). If no collision is detected, a pending buffer entry is allocated and its index is assigned to all downstream messages. This index is used to store transient information and directory states in a corresponding PBC entry. The index will also be used later by response messages to retrieve the content of the pending buffer entry. No collision detection is necessary in the case of responses.

3 New Coherence Controller Optimizations

This section discusses our three new optimizations for reducing CC occupancy and enhancing CC throughput: nonblocking PEs, directory and RAC/L3 early fetch, and superpipelined PEs.

3.1 Rationale: Coherence Controllers vs. Microprocessors

The architecture of CCs discussed in Section 2.2 is not unlike the architecture of conventional microprocessors. For example, both use scratchpad storage for quick access (pending buffer vs. register

file), on-chip caches (directory cache and tag cache vs. L1 and L2 caches), and execute units (protocol handling units vs. functional units). Because of their similar nature, some optimization techniques that work in one architecture may also work for the other. In fact, Section 2.3 described three CC optimizations that are similar to techniques used to increase the parallelism of microprocessors.

However, the characteristics of coherence transactions are sufficiently different from processor instructions that the design and impact of optimizations also differ. In this section, we analyze the differences and take advantage of them to increase the parallelism of CCs while minimizing negative performance and hardware complexity issues associated with the optimization techniques used in microprocessors.

Coherence transactions, unlike processor instructions, have no control dependences. The CC architecture, therefore, can be much simpler than the processor architecture. For instance, no support is needed for branch prediction, speculative execution based on branch predictions, and misprediction recovery. Furthermore, certain optimizations that are hindered by control dependence in microprocessors may provide better gain for CCs. For example, in microprocessors, deepening the pipeline facilitates the increase in clock speed, but becomes highly inefficient when branch misprediction is high. In contrast, this optimization enhances CC throughput without any misprediction penalty because there is no control dependence among coherence transactions.

On the other hand, coherence transactions suffer from data dependences as do processor instructions. Dependences occur when two or more concurrent transactions are directed to the same memory line address. However, data dependences among coherence transactions are not as common as among processor instructions, and can be handled cheaply without a significant impact on overall performance. When two transactions access the same memory line address, one is allowed to proceed while the other one is either bounced back to the source for replay or buffered until the former transaction completes.

For load instructions, microprocessors must resolve the associated address before the request can be issued to memory. This requirement sometimes delays the load. For CCs, memory line addresses are already available in transactions. We take advantage of this fact to fetch directory and tag information early on and, therefore, hide the long latency of directory cache and tag cache misses.

Finally, microprocessors utilize out-of-order execution to avoid stalls in the pipeline when instructions at the head of the queue incur long-latency actions. Hardware support for this feature is nontrivial as the processors must handle in-order retirement, misprediction recovery, and precise exceptions. In contrast, hardware support for nonblocking execution in CCs is not as significant, as it uses: a buffer to store away long-latency coherence transactions, nonblocking directory and tag caches, and extra wiring.

3.2 Nonblocking Protocol Engines

In the baseline design, when a PE processes a transaction and an access misses in the directory cache or in the tag cache, the transaction stalls until the directory or the RAC/L3 satisfies the miss (From this point on, we shall use L3 to refer to RAC/L3). This stall adds extra delay to the overall round-trip latency of the current transaction and may degrade performance. Since the baseline PE design processes transactions in order, the stall not only affects the current transaction. Indeed, all subsequent transactions stall as well, thereby reducing CC throughput.

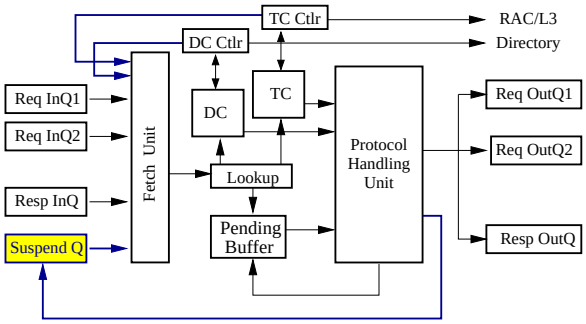


Figure 5. Pipeline of a nonblocking PE. It contains a Suspend queue for coherence transactions that have outstanding directory cache or tag cache misses.

To avoid stalling subsequent transactions, we propose a simple design of nonblocking execution for a PE pipeline (Figure 5). In this design, when a transaction causes a miss in the directory cache or in the tag cache, the protocol handling unit is notified. The latter responds by enqueuing the transaction in a *Suspend* queue. Subsequent transactions to different addresses are allowed to flow through the pipeline without stall. Later, when the controller for the directory cache or tag cache receives the response from the directory or L3, the suspended transaction in the Suspend queue is reactivated. That transaction is then allowed to arbitrate to reenter the PE. This transaction is given higher priority to avoid starvation. In the meantime, subsequent transactions to the address of the suspended transaction are bounced or buffered until that transaction completes and deallocates its pending buffer entry. Note that, although we describe the nonblocking optimization in the context of pipelined PEs, this optimization makes sense even with unpipelined PEs.

3.3 Directory & RAC/L3 Early Fetching

We propose early fetching as a second way to address PE stalls caused by misses in the directory cache or tag cache. Typically, an incoming coherence transaction must wait in one of the CC input queues to be fetched by a PE. If there are other transactions ahead in the queue, the wait time for this transaction can be long. The early fetching optimization exploits this wait time by fetching the associated directory entry into the directory cache. Specifically, as a coherence transaction enters an input queue, the corresponding memory line address is sent to the directory cache to check for the associated directory entry. If the entry is not in the directory cache, the directory controller launches a read request to the directory. Hopefully, by the time the transaction is finally processed by the PE, the corresponding entry will already be in the directory cache. Similarly, this optimization also fetches an L3 tag into the tag cache early.

As in nonblocking execution, early fetching is most beneficial when there is contention at CCs. While nonblocking execution reduces PE stalls by allowing subsequent transactions to proceed, early fetching uses the wait time in input queues to hide the latency of directory cache and tag cache misses. If the wait time is long, the majority of the directory and L3 tag access latency is hidden.

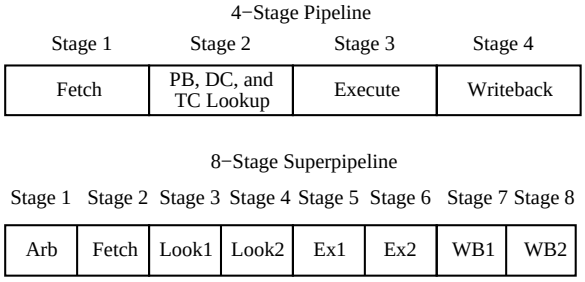


Figure 6. Baseline pipeline (top) and 8-stage superpipeline (bottom).

3.4 Superpipelined Protocol Engines

With superpipelining, the PE pipeline is made deeper by decomposing each pipeline stage into smaller ones. As a result, the PE can be clocked faster. In conventional microprocessors, this technique has the drawback of increasing the cycle penalty on branch misprediction. However, in CCs, there are no "branch transactions". Consequently, by superpipelining the PE, we can achieve higher throughput without being affected by control hazards.

In this paper, we decompose each pipeline stage from Section 2.3.2 into two shorter stages, as shown in Figure 6. Note that it now takes two cycles to look up the pending buffer, the directory cache, and the tag cache (*Look1* and *Look2*). The sizes of the directory cache and tag cache play a central role in the superpipeline technique. The deeper the pipeline and the smaller the directory cache and tag cache, the higher the CC clock rate. However, this will generally come at the expense of lower directory cache and tag cache hit rates. One way to balance the demands of high cache hit rates and short cycle times is to multi-bank the directory cache and the tag cache arrays, or design support for wave-pipelined accesses. Another way is to use nonblocking execution and early fetching to overcome the high miss rates of a small directory cache or tag cache.

3.5 Design Tradeoffs

The main advantage of the multiple-PE optimization (Section 2.3.1) is that only a single, simple PE module needs to be designed, tested, and verified. That module is replicated multiple times and glued together with wires, queues, and other logic. With such a design, we increase the CC throughput easily. One shortcoming of this design is the contention for shared resources. Although each PE has a directory cache and tag cache, all PEs share the directory and the L3. Contention occurs when more than one PE misses in its own directory cache or tag cache at the same time, causing multiple concurrent accesses to the directory or L3.

The pipelining optimization (Section 2.3.2) offers high throughput at the expense of higher overall latency and extra design complexity. A directory cache or tag cache must supply one directory entry or an L3 tag to a PE pipeline every single pipeline cycle to keep the pipeline full. Consequently, the directory cache and tag cache must be designed to be sufficiently small to keep their access to a single cycle, or be designed to handle pipelined lookups. The pipeline must also be designed to handle multiple transactions to the same memory line in the pipeline.

The split-streams optimization (Section 2.3.3) allows multiple concurrent transactions to share PE resources such as the fetch mechanism, the pending buffer, the directory cache, or tag cache. One advantage of this optimization is that request messages stalled due to misses in the directory cache or tag cache do not stall response messages. Hence, responses flow through the PE more deterministically. Another benefit comes from the decomposition of the pending buffer into two parts: a direct-mapped content buffer (PBC) and fully-associative address buffer (PBA). This not only facilitates concurrent accesses to the different buffers (requests access the PBA while responses access the PBC), but it also allows the direct-mapped PBC to be easily multiported for a lookup and an update at the same time. This scheme needs two protocol handling units, namely one for requests and one for responses.

The nonblocking execution optimization (Section 3.2) adds robustness to the baseline CC design or to the previously proposed throughput-enhancing optimizations by making them more tolerant to misses in the directory cache and tag cache. This optimization effectively minimizes bubbles in the PE pipeline in the presence of such misses. The additional complexity added includes a suspend queue, a wakeup mechanism, some additional wire routing, and nonblocking cache designs for the directory cache and tag cache.

In the early fetching optimization (Section 3.3), an early fetch is issued while the transaction waits at the CC input queues. The higher the contention, the longer the wait time, and the higher the opportunity for the early fetch to be effective. Note that this scheme requires an extra read port in the directory cache and tag cache from where the early fetches are issued. In general, when data is fetched early, three scenarios are possible: the early-fetched data arrives before it is needed but is displaced before being used, it arrives before it is needed and is found in the cache when it is needed, and it is still in flight when it is needed. The frequency of these three scenarios affects the impact of this optimization.

Finally, the superpipelining optimization (Section 3.4) is an extension of the pipelining optimization. The directory cache and tag cache play a central role in the decision space of pipelining and superpipelining. With superpipelining, the clock cycle time is shorter and, as a result, only smaller directory caches and tag caches can be accessed in one cycle. Unfortunately, small caches have higher miss rates, which cause bubbles in the pipeline and reduce the effective throughput of superpipelining. In our designs, we only consider single-cycle directory cache and tag cache designs for pipelining, and two-cycle directory cache and tag cache designs for superpipelining.

4 Experimental Methodology

4.1 NUMA Architecture Organization

To evaluate our CC designs, we model a 64-processor NUMA multiprocessor based on the organization shown in Figure 1. We consider three multiprocessors: one with 16 nodes of 1 SMP module each, one with 8 nodes of 2 SMP modules each, and one with 4 nodes of 4 SMP modules each. In all cases, an SMP module contains 4 processors. Table 1 details the parameters of the architecture modeled. Table 2 lists some of the latencies in the unloaded system. Note that we simulate small caches because the working sets of the applications that we run are small (Section 4.2). By using small caches as suggested in [18], we try to replicate the miss rate and network traffic conditions that we would see in a real system.

Parameter	Value
CPU	1GHz, 6-issue, out-of-order
Instruction window	128
Functional units	3 integer, 3 FP, 2 unifi ed LD/ST
Branch prediction	2K 2-bit saturating counters
Memory	MESI cache coherence protocol
Per-processor L1	16KB, 4-way, 64B lines
Per-processor L2	64KB, 4-way, 64B lines
Per-node L3	512KB/1MB/2MB, 4-way, 64B lines
SMP bus	250MHz, 16B wide, fully pipelined, split-transaction, separate address and data (like the 6XX IBM bus)
Network	250MHz, pipelined, 16B flit, node-to-node
CC	250MHz, each connected to 1-4 SMP modules
Directory cache	16K entries, 4-way, 16-entry lines
Tag cache	16K entries, 4-way, 16-entry lines

Table 1. Parameters of the architecture modeled.

Event	Latency
Processor to L1 (round trip)	2
Processor to L2 (round trip)	10
Processor to memory, supplied by another cache in local SMP (round trip)	54
Processor to memory, supplied by memory in local SMP (round trip)	118
CC detects response from bus	4
CC processes a message	12/16/16
CC issues request to bus	4
CC issues network message	4
Directory cache hit or tag cache hit	4
PHU executes handler	4
Directory, L3, or RAC access	24
Network node-to-node	52

Table 2. Latencies in processor cycles for the unloaded system. The latency for the CC to process a message is 12, 16, and 16 processor cycles for the baseline, pipelined, and superpipelined CC designs, respectively.

We use execution-driven simulations that accurately model a shared-memory multiprocessor system with wide-issue, out-of-order processors. Our simulator includes detailed contention models for processors, cache hierarchies, buses, memory controllers, interleaved memory organizations, PE pipelines, directory memory, and external point contention for the interconnection network.

4.2 Applications

For our evaluation, we use applications from both the SPLASH-2 [18] and SPECFP95 [16] suites and run them for 64 processors. From SPLASH-2, we select six applications that exhibit a range of communication behaviors. On the one hand, FFT, Ocean, and Radix have high communication-to-computation ratio and exhibit bursty traffic that may cause contention at CCs. On the other hand, LU, Water-Nsquared, and Water-Spatial have low communication-to-computation ratio and cause little contention at CCs. From SPECFP95, we use Hydro2D and Tomcatv. Hydro2D and Tomcatv have a medium to high communication-to-computation ratio. The data set sizes for the applications are listed in Table 3.

The SPLASH-2 applications are written in C and compiled using the SGI C compiler with optimization level -O2. The SPECFP95 applications are parallelized by the SUIF compiler [4]. For the SPLASH-2 applications, we discard serial sections because the only serial sections are initializations and terminations. For the SPECFP95 applications, serial and parallel sections alternate in a master-slave behavior; consequently, we include both types of sections in our results. We run all the applications with data and systems sizes that achieve acceptable speedups: the average parallel efficiency for 64 processors is 65%.

Appl	What It Does	Data Set Size
FFT	FFT computation	256K complex doubles
Radix	Integer radix sort	1M integer keys, radix 1K
Ocean	Study of ocean movements based on eddy and boundary currents	258×258 ocean grid
LU	LU factorization	512×512 matrix, 16x16 blocks
Water-Nsquared	$O(n^2)$ study of forces and potentials of water molecules	512 molecules
Water-Spatial	Study of forces and potentials of water molecules	512 molecules
Hydro2D	Computation of galactic jets using hydrodynamical Navier Stokes equations	Reference input
Tomcatv	A vectorized mesh generation program	Reference input

Table 3. Applications and data sets used.

5 Evaluation

This section evaluates the impact of the proposed throughput-enhancing CC optimizations on system performance. First, we show the impact of each individual optimization, and then the impact of a few combinations of them. In the following, *Baseline* corresponds to a system with the CC architecture described in Section 2.2. Such a CC does not even include the previously proposed optimizations of Section 2.3.

Note that we keep the total directory cache and tag cache size of a CC constant as the number of PEs in a CC varies. For example, each PE of a dual-PE CC has half of the directory cache and tag cache that a PE of a single-PE CC has. This allows us to evaluate the effect of PE parallelism on performance while keeping the effect of cache size constant.

5.1 Individual Optimizations

5.1.1 Speedup

Figure 7 shows the speedup of the different machine organizations and CC optimizations relative to a system with 16 1-SMP nodes and baseline CCs. For each application, the bars are grouped based on the machine organization: 16 1-SMP nodes (*1 SMP/CC*), 8 2-SMP nodes (*2 SMPs/CC*), and 4 4-SMP nodes (*4 SMPs/CC*). Within each group, each bar corresponds to no CC optimization (*Baseline*) or a single CC optimization: two PEs per CC (*2PEs*), 4-stage pipelined PE (*P4*), PE with split request and response streams (*S2*), nonblocking PE (*NoBlk*), PE with early fetch (*EarlyF*), and 8-stage superpipelined PE (*P8*). In our plots, we include both the new optimizations and the previously proposed ones because we want to evaluate them all under the same, modern system architecture.

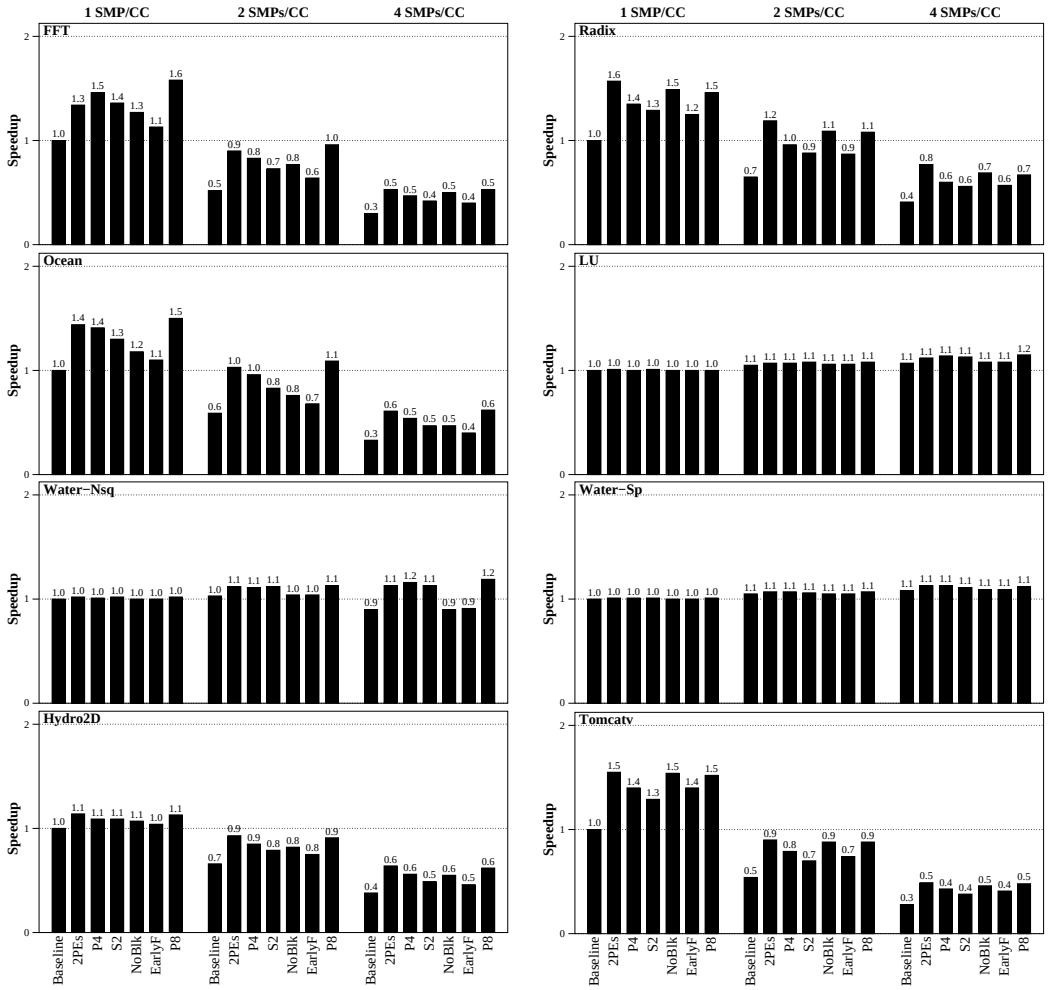


Figure 7. Speedup for different machine organizations and single CC optimizations. For each application, the bars are grouped based on machine organization: 1 SMP per CC (left), 2 SMPs per CC (middle), and 4 SMPs per CC (right). Within each group, each bar corresponds to a different CC design.

The figure shows that, as we go from *1 SMP/CC* to *2 SMPs/CC* and *4 SMPs/CC*, most applications run slower. For example, a *4 SMPs/CC* system with baseline CCs obtains a speedup of 0.3-0.4 relative to a *1 SMP/CC* system with baseline CCs for FFT, Radix, Ocean, Hydro2D, and Tomcatv. This effect is due to the higher CC and resource contention in the machine with fewer nodes. However, such a machine is *cheaper*, as it has fewer nodes, a smaller chassis, and a smaller interconnection fabric. On the other hand, compute-intensive applications such as LU, Water-Nsq and Water-Sp experience little or no slowdown as they benefit from better locality of accesses in systems with fewer, wider nodes.

We also see that all individual CC optimizations speed up the applications over the baseline CC for a machine with the *same number of nodes*. The speedups go all way up to 2 (for example, for superpipeling in *2 SMPs/CC* in FFT). The highest speedups are delivered by superpipelining and two PEs per CC. To a lesser extent, pipelining and nonblocking execution also deliver good speedups. CCs with two PEs outperforms pipelining in most cases and superpipelining in some cases. CCs with dual PEs are effective at tolerating misses in the directory cache or tag cache. Indeed, when one PE suffers a miss and stalls, the other PE can still process coherence transactions. However, for FFT, CCs with two PEs underperform CCs with pipelined PEs because the former have half the cache size per PE compared to the latter. This results in higher directory cache and tag cache misses. A unified large cache in the 2-PE CC may work better for certain types of access patterns.

As node size increases, coherence traffic to each CC increases and causes higher miss rates in directory caches and tag caches. Because frequent stalls can wipe out the benefit of pipelining and superpipelining, these optimizations are more sensitive to directory cache and tag cache misses than CCs with two PEs. For instance, FFT shows a speedup of 1.6 for superpipelining and 1.3 for dual-PE CCs in *1 SMP/CC*, and 0.5 for both with *4 SMPs/CC*. In Section 5.2, we will show that pipelining and superpipelining are much more robust when they are combined with the nonblocking technique to tolerate directory cache and tag cache misses.

We also note that, even with two PEs per CC or superpipelining, a system with 4 4-SMP nodes (*4 SMPs/CC*) is still significantly

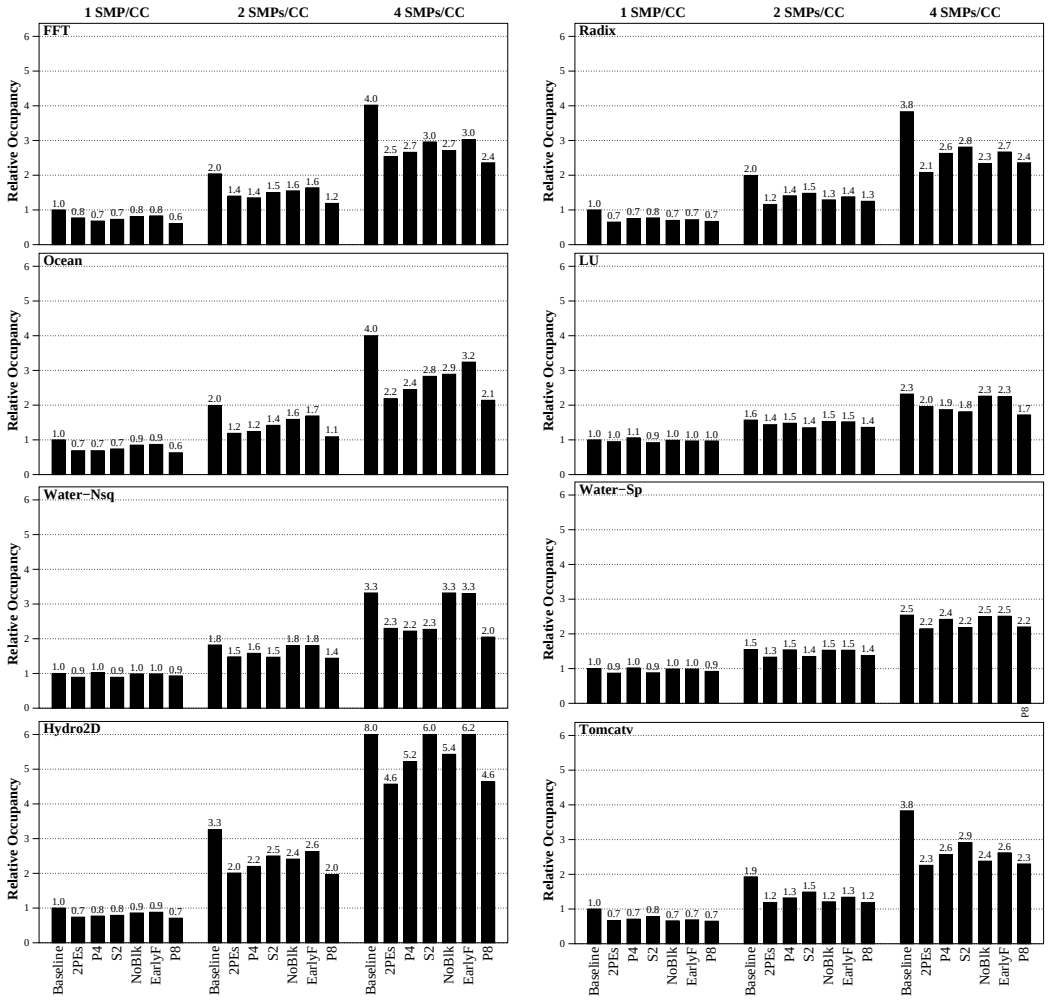


Figure 8. Relative CC occupancy for different machine organizations and single CC optimizations. Each bar in the figure corresponds to a different CC design.

slower than a system with 16 1-SMP nodes (*1 SMP/CC*) and baseline CCs for communication-intensive applications such as FFT, Radix, Ocean, Hydro2D, and Tomcatv. CC contention and directory cache and tag cache misses are sufficiently high that a single optimization is unable to overcome them.

The split request-response streams optimization is relatively less effective because PEs with split streams provide concurrency only when there are both request and response coherence messages in the same PE.

Finally, we consider the two techniques designed to hide or tolerate directory cache and tag cache miss latency. The nonblocking PE optimization allows CCs to continue processing subsequent coherence transactions in the presence of misses in these caches. This optimization is effective at improving the execution of applications that suffer from frequent misses in these caches, such as FFT, Radix, and Tomcatv. The early-fetch technique, however, is not as effective because it is able to hide only a small portion of the entire miss latency. CCs still stall when not all of the miss latency is complete hidden.

5.1.2 CC Occupancy

We define CC occupancy as the total number of cycles the CC is utilized. CCs with low occupancy are usually idle and can service incoming coherence transactions immediately. In contrast, CCs with high occupancy are most likely busy and may not be able to process new transactions quickly.

Figure 8 shows the CC occupancy in all system configurations relative to the occupancy in the baseline CC of the system with 16 1-SMP nodes. The figure illustrates several trends. First, CC occupancy changes significantly as we change the width of the node. Ideally, we would expect that, as we reduce the number of nodes by 2X or 4X, the occupancy increases by 2X or 4X, respectively. However, two factors change this scenario.

The first factor is that, with fewer nodes, the miss rates of the directory caches and tag caches increase. Recall that the size of these caches in a CC remains fixed regardless of node size. As a result of these additional misses, CC occupancy tends to go up. The second factor is that, with fewer nodes, more memory accesses are satisfied locally. It can be shown that, on average, local transactions

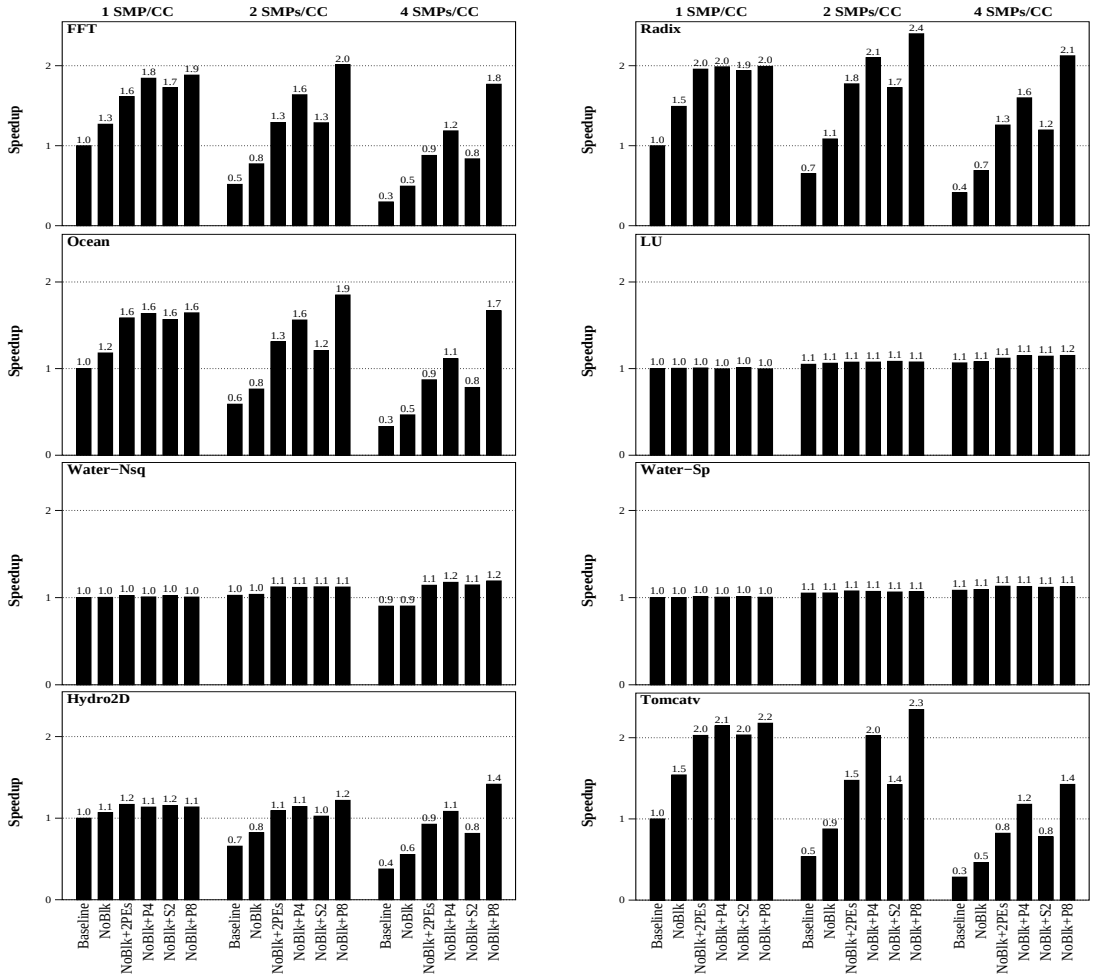


Figure 9. Speedup for different machine organizations and CCs that combine nonblocking execution with one other optimization.

use the CCs for fewer cycles than remote transactions. As a result, the CC occupancy tends to go down.

Figure 8 shows that both effects appear. For example, the occupancy of Hydro2D increases 8 times as the number of nodes is reduced from 16 to 4. In contrast, the CC occupancy in LU increases only 2.3 times as the number of nodes decreases 4 times.

We also see from the figure that, in communication-intensive applications such as FFT, Radix, Ocean, and Tomcatv, CC occupancy significantly decreases as we apply CC optimizations. For example, single CC optimizations reduce the CC occupancy of FFT with 4 SMPs/CC by an average of 32%. On the other hand, compute-intensive applications such as LU and Water-Sp exhibit a less drastic decrease in CC occupancy with CC optimizations. These applications have low CC occupancy and do not stress CCs much.

5.2 Combining Nonblocking Execution with Other Optimizations

In this section, we combine CC optimizations to try to exploit some of the synergies between them. In particular, nonblocking execution may have a multiplicative effect on the impact of other optimizations because it eliminates long stalls in misses. The exception is early fetch, which targets a similar problem. Consequently, in this section, we combine nonblocking execution with all other optimizations except early fetch.

5.2.1 Speedup

Figure 9 shows the speedup of systems where the CC combines nonblocking execution (*NoBlk*) with one other optimization. As usual, the speedup is relative to a system with 16 1-SMP nodes and baseline CCs. As a reference, we show the speedup of the CC with only the nonblocking execution optimization.

The figure shows that nonblocking execution plus pipelining (*NoBlk+P4*) or superpipelining (*NoBlk+P8*) boost the speedups substantially. Consider nonblocking plus superpipelining. On average for all the applications, the resulting speedups relative to a system with the *same number of nodes* and baseline CCs are 1.48 for *1 SMP/CC*, 2.50 for *2 SMPs/CC*, and 3.56 for *4 SMPs/CC*. In particular, FFT shows speedups of 4 for *2 SMPs/CC* and 6 for *4 SMPs/CC*. These speedups are much higher than those in Figure 7. They are higher for the systems with more SMPs per node because a harder

bottleneck is being removed. On the other hand, LU, Water-Nsq and Water-Sp experience little slowdown across *different machine organizations* and, therefore, benefit little from optimization.

The reason why nonblocking execution especially improves pipelining and superpipelining is that it allows these two techniques to maintain high bandwidth even in the presence of directory cache and tag cache misses. Without nonblocking execution, pipelined and superpipelined PEs stall during misses. Nonblocking execution allows these PEs to process subsequent transactions without stall. The benefit also increases as node size increases, because the miss rates of the directory cache and tag cache are higher.

Recall from Figure 7 that pipelining and superpipelining with 2 SMPs/CC and 4 SMPs/CC typically underperform the baseline CC with 1 SMP/CC. In contrast, Figure 9 shows that either one of the two pipelining techniques combined with nonblocking execution outperforms the baseline CC with 1 SMP/CC.

CCs with two PEs benefit from nonblocking execution, but not as much as pipelining or superpipelining. One reason is that CCs with two PEs stall only when both PEs suffer directory cache or tag cache misses concurrently; when one PE suffers a miss, the other PE can still continue executing. Moreover, with nonblocking execution, CCs with two PEs do not have as much bandwidth as pipelined PEs or superpipelined PEs. Overall, comparing the *2PEs* bars of Figure 7 with the *NoBlk+2PEs* bars of Figure 9, we see that nonblocking execution offers less dramatic improvements to CCs with two PEs. For example, FFT exhibits speedups of 1.3 with *2PEs* and 1.6 with *NoBlk+2PEs* on 1 SMP/CC. As node size increases to 2 SMPs/CC and 4 SMPs/CC, CCs with two PEs underperform the two pipelining techniques more.

The split request-response streams optimization does not match pipelining or superpipelining in throughput either. Consequently, it underperforms these optimizations when combined with the nonblocking execution optimization.

Overall, among all the configurations, nonblocking superpipelined PEs offer the highest performance. A machine with these CCs and 8 2-SMP nodes (*2 SMPs/CC*) typically delivers the highest speedups. For instance, the speedups of Radix and Tomcatv are 2.4 and 2.3, respectively. Note also that a machine with these CCs and 4 4-SMP nodes (*4 SMPs/CC*) is very competitive. It is usually about as fast as a machine with the same CCs and 16 1-SMP nodes (*1 SMP/CC*). However, it has 4 times fewer nodes, a smaller chassis, and a smaller interconnection fabric, which is likely to reduce the overall cost of the machine significantly.

5.2.2 CC Occupancy

Figure 10 shows the relative CC occupancy with the same optimizations as in Figure 9. The figure shows that, as we move to machines with large nodes (*2 SMPs/CC* and *4 SMPs/CC*), combining nonblocking execution with another optimization significantly reduces CC occupancy. This is in marked contrast to the high occupancies observed with a single optimization (Figure 8).

Nonblocking execution and superpipelining provide the highest reduction in CC occupancy. For example, in systems with 2 SMPs/CC or 4 SMPs/CC, the CC occupancy in FFT, Radix, Ocean, and Tomcatv drops to below that of the baseline CCs with 1 SMP/CC. The reason is that the nonblocking optimization allows superpipelining (and pipelining as well) to maintain high CC throughput by minimizing stalls from directory cache and tag cache misses.

For LU, Water-Nsq, and Water-Sp, the occupancy reductions are modest. The reason is that, in these applications, processors communicate much less.

6 Related Work

Multiple PEs were employed in the Sun S3.mp machine [15]. Each CC in the machine dedicated one PE to handle transactions to local addresses and one to handle remote addresses. The architects of the Sequent STiNG system [11] also considered a similar approach as a way to reduce CC occupancy. However, the performance impact of this optimization on these systems was not evaluated.

Michael *et al.* [13] evaluated systems with one PE for local addresses and one for remote ones, while comparing the performance of hardwired and programmable CCs. They showed significant improvements in the performance of such systems over those with single-PE CCs. They also found load imbalance in the coherence traffic to the PEs, indicating the potential for more improvement if the imbalance problem were resolved.

Falsafi and Wood [2] studied the performance of software coherence handlers on a shared-memory multiprocessor. They utilized several idle processors in the machine to run the software coherence handlers. Falsafi and Pragaspathy [1] evaluated four different schemes to assign coherence traffic to multi-PE CCs. They found that a dynamic partitioning scheme performed best by balancing load among multiple PEs.

The MAGIC chip [5] is a CC built using a pipelined two-way superscalar RISC core with special bit-manipulation instructions useful for coherence operations. The MAGIC pipeline is very similar to a processor pipeline, which fetches and executes instructions from memory to manipulate coherence transactions. In contrast to the programmable MAGIC chip, our CCs are hardwired, and are custom architected for coherence protocol handling. Except for a prior study [14], the performance impact of pipelining in coherence controllers has not been discussed in the literature. We focus on hardwired CCs because they have been shown to yield higher performance than programmable protocol processors [6, 13].

7 Conclusion

Based on a comparison between general-purpose microprocessors and coherence controllers (CCs), we have proposed three optimizations to enhance the throughput and reduce the occupancy of hardwired CCs: nonblocking execution, early fetching, and Protocol Engine (PE) superpipelining. Nonblocking execution in the CC reduces stalls by processing subsequent coherence transactions when the current one suffers a miss in the directory cache or tag cache; early fetching hides the miss in the directory cache or tag cache by fetching necessary information in advance; finally, superpipelining subdivides the pipeline of the PEs into more, finer stages.

We evaluated combinations of these CC enhancements and previously proposed optimizations such as multiple PEs per CC, PE pipelining, and split request-response streams. Our results showed that using CCs that combine nonblocking execution and superpipelining boosts the performance of machines substantially. With

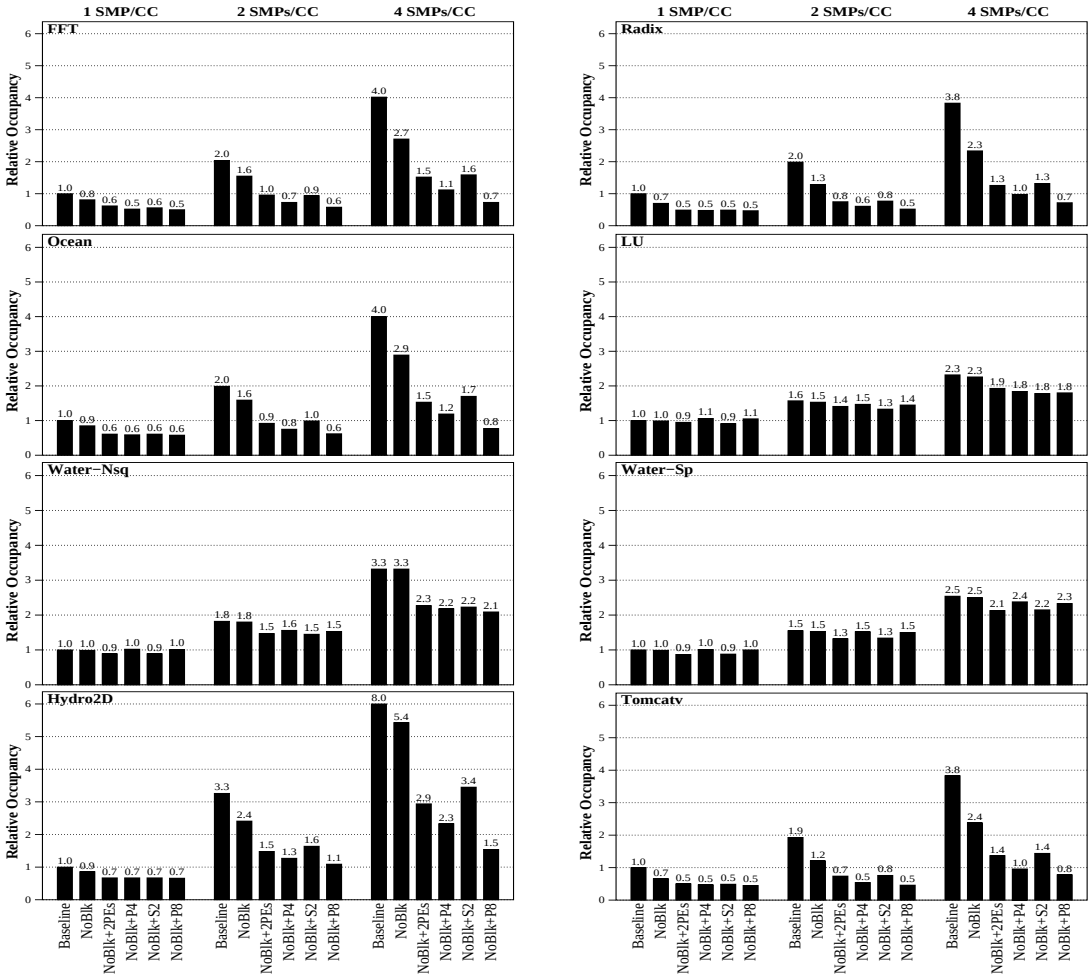


Figure 10. Relative CC occupancy for different machine organizations and combined CC optimizations. Each bar in the figure corresponds to a different CC design.

these CCs, a 64-processor machine with four nodes of four SMPs per node runs on average 3.56 times faster than if it used conventional CCs. In addition, the machine runs about as fast as a more costly 64-processor machine with sixteen nodes of one SMP per node and the same advanced CCs. This is despite using much less network, chassis, and node hardware. Consequently, with our proposed advanced CCs, we can reduce the system cost significantly without affecting performance.

References

[1] B. Falsafi and I. Pragaspathy. Address Partitioning in DSM Clusters with Parallel Coherence Controllers. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2001.

[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] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing*, pages 312–321, August 1990.

[4] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[5] 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.

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

[7] IBM. IBM eServer pSeries and IBM RS/6000 Facts and Features. Armonk, New York. http://www.ibm.com/servers/eserver/pseries/.

[8] D. Joseph, M. Michael, and A. Nanda. Split Multiported Pending Buffer. Patent pending, 1999.

[9] 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 FLASH Multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[11] 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.

[12] 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.

[13] 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 24th International Symposium on Computer Architecture*, pages 219–228, June 1997.

[14] A. Nanda, A.-T. Nguyen, M. Michael, and D. Joseph. High-Throughput Coherence Controllers. In *Proceedings of High-Performance Computer Architecture (HPCA-6)*, January 2000.

[15] 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.

[16] Standard Performance Council. The SPEC95 CPU Benchmark Suite, 1995.

[17] Sun Microsystem Inc. Sun Enterprise 10000 Server: Dynamic System Domains. Palo Alto, California. http://www.sun.com/servers/white-papers/domains.html.

[18] 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.