Integrated Memory Controllers with Parallel Coherence Streams

Mainak Chaudhuri, *Member*, *IEEE*, and Mark Heinrich, *Senior Member*, *IEEE*

Abstract—Previous work in scalable hardware distributed shared memory (DSM) multiprocessors has established the critical and dominant role that protocol processing bandwidth (or its inverse, occupancy) plays in determining overall performance in architectures with standalone memory/coherence controllers. However, with recent architectural trends toward integrated (on-chip) memory controllers and the well-known fact that processor frequency is increasing more rapidly than memory systems', we must ask whether parallel coherence processing engines (either multiple integrated protocol processors/cores or multiple protocol threads) are needed in DSM machines constructed from modern processor architectures and, if so, when. We construct a useful analytical model to give the designer insight into when parallel coherence streams will improve performance and verify our model via detailed simulation on 64-threaded microbenchmarks and parallel applications and on single-node multiprogrammed workloads. Surprisingly, and contrary to related work, we find that, in these architectures, adding a second coherence engine has almost no impact on performance. Further, for less-tuned applications that suffer from hot spots (contentious requests to the same memory line), additional engines offer no benefit whatsoever. Even with double the memory bandwidth (or channels), an additional coherence processing stream yields only slight performance improvement. Only for a special class of DSM machines employing directoryless broadcast protocols over unordered interconnects does parallel "snoop" processing offer reasonable performance improvement for communication-intensive applications. Overall, given the architectural trends, this is good news for DSM designers who want to minimize the resources necessary (protocol threads or integrated protocol processor cores for maintaining internode coherence, respectively) to create SMTp-based or multi-CMP-based scalable DSM machines using directory protocols.

Index Terms—Distributed shared memory multiprocessor, directory protocol, multiple coherence controllers, coherence bandwidth, integrated memory controller.

1 Introduction

INTEGRATED memory controllers appearing in several high-end microprocessors such as the Alpha 21364 [6], the IBM POWER5 [11], the AMD Opteron [12], and the Sun UltraSPARC III and IV [30], [31] provide a direct solution to reducing the round-trip memory transaction latency. Multiprocessor systems built from microprocessors with integrated memory controllers are naturally distributed memory machines (each processor has its own local memory) as opposed to symmetric multiprocessors (SMPs). A "snoop-based" cache coherence protocol in such machines requires that a point-to-point message be sent to all processors in the system to complete even a simple remote read miss when the cache line is clean at the home node [2]. Therefore, a high-performance and bandwidth-thrifty solution would employ a scalable directory-based cache coherence protocol.

Research on directory-based coherence controller design has led to two different high-performance architectures. Hardwired coherence controllers, found in the MIT Alewife [1], the KSR1 machine [13], the SGI Origin 2000 [18], and the Stanford DASH [19], offer fast handling of coherence transactions but no flexibility in choice of the coherence protocol. On the other hand, customized programmable protocol processors embedded in memory controllers, found in the Piranha chip-multiprocessor [3], the Opteron-Horus [15], the Stanford FLASH multiprocessor [9], [17], the STiNG multiprocessor [20], and the Sun S3.mp [26], execute coherence protocol handlers in the form of optimized sequences of instructions or microcode. Due to higher protocol processor occupancy, these designs may offer degraded performance, depending on the speed of the protocol processor. However, this kind of design allows late binding of the protocol, flexibility in the choice of the protocol, and, in most cases, an easier and faster protocol verification phase. Despite these compelling reasons, lower performance has led the designers of commercial DSM multiprocessors not to consider programmable coherence controllers. In fact, prior research [9] has shown that there can be as much as a 12 percent performance gap between hardwired controllers and customized protocol processors and that coherence controller occupancy is the most important determinant of performance [4].

Designs employing multiple coherence engines offer low-occupancy coherence. What is surprising is that, with multiple coherence engines, even hardwired controllers show significant performance improvement in DSM multiprocessors built out of SMP nodes [24], [27]. In such systems, a single coherence engine in a standalone memory controller is not sufficient to handle all coherence transactions efficiently. However, we note that none of these studies have explored multiprocessors with integrated memory controllers, clocked at the main processor frequency, that have the potential of considerably reducing protocol processor occupancy.

- *M. Chaudhuri is with the Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016, India. E-mail: mainakc@cse.iitk.ac.in.*
- *M. Heinrich is with the School of Computer Science, University of Central Florida, Orlando, FL 32816. E-mail: heinrich@cs.ucf.edu.*

Manuscript received 23 Nov. 2005; revised 25 June 2006; accepted 22 Sept. 2006; published online 9 Jan. 2007.
Recommended for acceptance by A. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0488-1105.
Digital Object Identifier no. 10.1109/TPDS.2007.1044.

In this paper, we reconsider customized programmable protocol processors in light of integrated memory controllers and explore the extensions to memory controller hardware and coherence protocol software for enabling multiple coherence engines in such systems. Since simultaneous multithreading (SMT) [32], [33] is common in high-end microprocessors today [11], [14], [16], [22], in this study, we consider directory-based DSM multiprocessors of up to 16 nodes with each node built from an SMT processor capable of running four application threads. This aspect, along with integrated memory controllers, makes our work quite different from prior research in this direction. Although, in this paper, we squarely focus on SMT nodes, we believe that the trends presented here will apply equally to the internode coherence controllers in DSM multiprocessors built out of chip multiprocessor (CMP) nodes. We consider two classes of directory-based programmable coherence controller design. One class consists of one or more customized protocol processors embedded in the integrated memory controller, while the other exploits the recently proposed SMTp architecture [5], which executes the coherence protocol handlers on one or more of the hardware thread contexts of the main SMT processor, thereby entirely eliminating the embedded protocol processor. For both designs we present the architectural extensions needed to support concurrent handling of multiple independent coherence transactions, which we will refer to as parallel coherence streams.

Simulation results on two carefully crafted microbenchmarks, a selected set of shared memory scientific applications, and single-node multiprogrammed workloads show that the designs with SMTp extensions or with embedded protocol processors running at full processor frequency do not suffer from a shortage of protocol bandwidth. Even when the embedded protocol processor is operated at half the main processor frequency we find that adding a second protocol processor does not improve performance significantly, although it does reduce the average waiting time of coherence transactions. Only for a frequency ratio of 4 between the main processor and the protocol processor do we start seeing performance gains when adding a second protocol processor. To explain our results, we develop a simple, yet generic, analytical model. In the model, we derive the relationships between DRAM bank behavior of a batch of concurrent coherence requests, DRAM access latency, protocol processor occupancy, and memory channel bandwidth for which adding a second protocol handling engine may improve performance.

In this paper, we focus on scalable directory-based coherence processing. A different class of broadcast protocols over unordered interconnects, namely, AMD Hammer [2] and Token Coherence [21], eliminates the directory indirection, but increases the volume of coherence messages. While presenting the results, for completeness, we briefly discuss our experience with a Hammer-like protocol in the context of parallel snoop processing.

In summary, this paper contributes in the following two major ways:

- This paper presents a thorough evaluation of two different classes of customized programmable coherence controller architectures with integrated memory controllers and simultaneous multithreading. Surprisingly, the results suggest that customized protocol processors do not suffer from a shortage of protocol bandwidth in modern systems. This is very different from the previously published results on off-chip hardwired coherence engines [24], [27]. Parallel coherence stream processing turns out to be beneficial only for directoryless broadcast protocols over unordered interconnects.
- This paper develops a simple and generic analytical model to decide whether adding a second coherence engine will improve performance in a directory-based protocol.

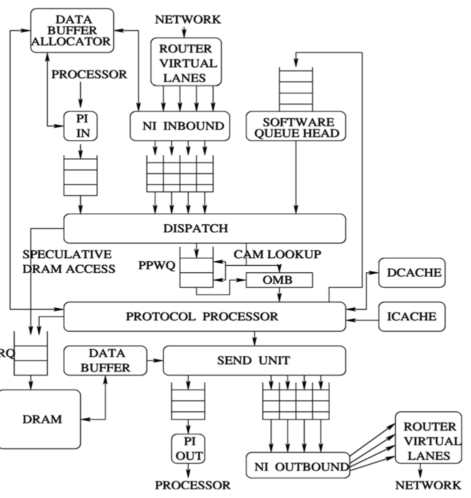


Fig. 1. Memory controller architecture. PI, NI, PPWQ, OMB, and MRQ stand for processor interface, network interface, protocol processor wait queue, outstanding message buffer, and memory reorder queue, respectively.

The next section describes the baseline architecture, and Section 3 explains the few necessary microarchitectural extensions and coherence protocol software modifications for enabling parallel coherence streams. We also present our analytical model in this section. In Sections 4 and 5, we present our simulation results. Section 6 briefly discusses prior research in this area, and Section 7 presents concluding remarks and future directions.

2 Baseline Memory Controller

In this section, we first discuss a DSM architecture with an embedded protocol processor and we then describe the salient architectural modifications needed to realize the SMTp architecture. In the next section, we extend this baseline architecture to enable multiple protocol processing streams.

Our memory controller architecture shown in Fig. 1 is derived from the Memory And General Interconnect Controller (MAGIC) of the Stanford FLASH multiprocessor [17]. However, its design is closer to the hub of the SGI Origin 2000 [18] with the exception that it is programmable and can execute any cache coherence protocol. Coherence messages arrive at the inbound processor interface (PI) or the inbound network interface (NI) and wait for the dispatch unit to schedule them. The dispatch unit carries out a round-robin scheduling among the PI queue, four NI input queues (corresponding to four virtual networks or lanes), and the software queue (see below) by examining the heads of the six queues. After selecting a message, a table lookup decides which protocol message handler should be invoked to serve the scheduled message. Also, the outcome of the table lookup speculatively initiates a DRAM access at the requested address if the message expects a data reply. The speculative DRAM access potentially hides the protocol processing latency under the memory access latency. The dispatched message waits in the Protocol Processor Wait Queue (PPWQ) until the protocol processor extracts it for execution. Each PPWQ entry contains the requested address, the starting PC of the handler, and the header of the message. The speculative memory request is sent to the memory reorder queue (MRQ) and can start accessing DRAM immediately if it does not have a bank conflict with any outstanding requests. After the memory read is completed, the cache line is transferred over a free memory channel to a data buffer, allocated by the PI or NI when the message arrived. The contents of the data buffer are sent to the requester only after the protocol processor instructs the outgoing interface to do so. The protocol processor starts serving a message by extracting a new entry from the head of the PPWQ, storing it in the Outstanding Message Buffer (OMB), marking the PPWQ entry free, and transferring control to the PC stored in the OMB. The software queue provides a self-scheduling mechanism for the protocol processor and plays an important role in the deadlock avoidance mechanism related to outgoing NI queue space needed to send invalidations to the sharers (see [17] for details).

Our reasonably aggressive memory controller is organized to have a four-stage macropipeline, namely, PI/NI inbound, handler dispatch, protocol processor, and the PI/NI outbound. These four units can operate concurrently on four different messages in a pipelined fashion. The protocol processor itself has a statically scheduled dual-issue five-stage in-order MIPS pipeline with some additional custom instructions to accelerate directory-specific bit manipulations. It has dedicated protocol instruction and data caches backed by main memory.

The full-map write-invalidate bitvector coherence protocol used in this study is derived from the SGI Origin 2000 protocol [18]. A typical protocol handler (e.g., a local read miss handler) starts by calculating the address of the directory entry corresponding to the requested cache line through a one-to-one hash function on the requested address. Next, the handler loads the directory entry in a register (the protocol processor has a MIPS-like ISA with 32 registers) and carries out some integer arithmetic on this register to decide appropriate coherence actions. The protocol handler uses *send* instructions to instruct the send unit to initiate message transfer operations via the PI or NI, as appropriate.

2.1 The SMTp Architecture

We now briefly discuss the salient features of the SMTp architecture [5] used in this study. The SMTp architecture replaces the embedded protocol processor in the baseline memory controller described above by a hardware thread context in the SMT processor core. The handler dispatch hardware sets the Protocol PC Valid (PPCV) bit in the fetcher when a message is ready to be serviced and the fetcher has completed fetching the current handler. At the same time the dispatch hardware sets the protocol handler PC. This lets the fetch policy (ICOUNT in this study) consider the protocol thread as a potential fetch candidate. When the fetcher completes fetching the current handler it probes the PPWQ to see if some message is waiting to be serviced at the head. If not, it turns off the PPCV bit. Thus, the protocol thread can start fetching a handler while the previous handler is still executing. This helps in hiding some of the latency imposed by long front-end pipeline on short critical handlers.

All protocol instructions go through the pipeline resources shared with the application threads, including the two-level cache hierarchy. Note that the protocol thread does not have a separate cache hierarchy and this is a major advantage in terms of complexity over embedded protocol processors. A protocol L2 miss does not invoke another protocol thread recursively. Instead, the dispatch hardware recognizes the protocol address space and enqueues the request into the MRQ directly.

In an SMTp architecture, the forward progress of an application thread's L2 miss depends on the forward progress of the protocol thread. This creates dependence cycles involving the shared pipeline resources such as the front-end queue buffers, branch stack space, integer registers, integer queue slots, load/store queue slots, speculative store buffers, and miss status holding registers (MSHRs). Since, in this design, we consider per-thread active lists (or reorder buffers), they are not involved in this kind of resource deadlock. We avoid any deadlock involving these resources by maintaining at least one reserved instance that is usable by the protocol thread only. For performance reasons more instances may be reserved. To break a subtle deadlock cycle involving cache index conflicts between the application and the protocol threads, a number of cache line-sized bypass buffers are reserved for the protocol thread [5].

The coherence protocol software requires no modifications when switching from an architecture with an embedded protocol processor to SMTp. However, we optimized the SMTp protocol code by eliminating the unnecessary NOPs generated by the static dual-issue scheduler of the embedded protocol processor.

3 Multiple Coherence Streams

In this section, we explain the necessary modifications to both the hardware and coherence protocol software for enabling multiple coherence streams. The modifications are essentially independent of whether the parallel coherence engines are multiple embedded protocol processors or multiple protocol threads. The basic difference between the two architectures is that the former has a much higher hardware cost due to the complete replication of protocol processor. SMTp only requires that multiple hardware thread contexts be reserved for protocol execution. In the following, we focus our discussion on multiple protocol processors only. We conclude this section with a simple performance model applicable to both embedded protocol processors and SMTp.

3.1 Multiple Protocol Processors

With multiple protocol processors, the memory controller's dispatch unit must be modified to ensure mutual exclusion for directory accesses. Before selecting a message from the heads of the six incoming queues, the dispatch unit compares the address requested by the message with addresses of the requests waiting in the PPWQ and OMB. The size of the OMB is increased to be equal to the number of protocol processors, and one OMB is logically assigned to one protocol processor. For a fast dispatch, we make the address CAMs in the PPWQ and OMB six-ported so that all six potential message addresses can be compared simultaneously and the winner chosen through round-robin priority. Once a message is selected, it is enqueued in the

PPWQ. The dispatch unit can write only one message to the PPWQ per cycle (it needs only one write port). Whenever a protocol processor is free, it arbitrates for the PPWQ read port. The read port is granted to the protocol processor having the lowest identifier among the contenders with the help of a priority encoder. Thus, the $k\mathrm{th}$ protocol processor gets a message to work on only if the protocol processors 0 to $k-1$ are busy. Thus, assignment of a message to a protocol processor is completely dynamic and is based solely on availability of protocol processors. In every cycle, a new protocol handler can be started on a free protocol processor.

Under pathological conditions, the dispatch unit may find that all the six potential messages are suffering from an address conflict with some outstanding request. This leads to idle dispatch cycles. As a way to reduce the number of idle dispatch cycles, we explore the potential of out-of-order dispatch by making five out of the six queues collapsible and organizing them as CAMs (the software queue cannot be made collapsible without significant performance penalty, since it is not implemented in hardware). The dispatch hardware considers all the requests (instead of only the head) in a queue before selecting one.

To enable concurrent execution of coherence protocol handlers, the coherence protocol code must be modified so that critical sections are protected via locks. We experimented with two types of spin locks, namely, atomic test and set (requires a special *test-and-set* instruction) and LL/SC. For high-throughput, the test and set lock bit is maintained on-chip in a special register instead of in memory. The LL/SC lock is implemented in the conventional manner by allocating the lock in the protocol address space. The access to the shared software queue data structure is the only major critical section in our protocol code. The running occupancy of the software queue is also a shared variable among the protocol processors. This variable is read by read-exclusive and upgrade handlers before starting to send invalidations, if any. Access to this variable is also protected by locks. In summary, the coherence protocol executed on a system supporting parallel coherence streams will exhibit increased dynamic instruction count leading to an increased average handler occupancy. Thus, we are trading the increase in occupancy of individual handlers with concurrency across different handlers (an usual latency versus throughput trade-off).

In the baseline (non-SMTp) architecture, the protocol instruction and data caches are not replicated per protocol processor due to coherence problems involving directory entries. The shared caches are made dual ported to support accesses from two protocol processors at the same time. However, each cache can have only one outstanding miss. While a miss is outstanding, the caches can continue to service hits to different lines. We chose such a design because the number of misses (other than cold misses) in the protocol instruction and data caches is small for the data sizes we consider. As a result, supporting one outstanding miss for each cache does not significantly affect performance but does obviate the need for complex circuitry associated with multiple outstanding misses. This modification does not apply to the SMTp architecture since the protocol threads already share the lockup-free cache hierarchy of the main SMT processor core.

The protocol processors (or the protocol threads) arbitrate for writing to the header and address registers in the send unit. We found that the frequency of multiple concurrent sends does not justify the complexity of having multiple header and address registers. The send unit has the best case occupancy of one cycle and, therefore, can initiate a new message every cycle.

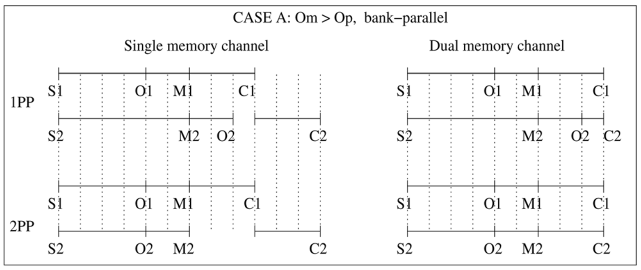


Fig. 2. Timing diagram of the performance model for bank-parallel requests.

Finally, the boot handler of the protocol code requires some minor modifications. The boot handler is responsible for initializing various memory controller state and the protocol processors' (or protocol threads') registers. With multiple protocol processors per memory controller only one protocol processor needs to initialize the memory controller states. Each of the remaining protocol processors executes a much shorter boot handler containing only the code to initialize its own architectural register file.

3.2 A Performance Model

In this section, we present a simple analytical model to help decide whether more than one coherence controller is helpful. We analyze the time spent in the memory system by a batch of concurrently arriving coherence requests with one and two coherence controllers. The applicability of the model is not limited to architectures with integrated memory controllers, but is equally applicable to any multiprocessor system exercising directory-based cache coherence. To keep the model simple, we focus on architectures with one or two coherence controllers, but it can be trivially extended to any number of coherence controllers. As pointed out in other work [4], the handlers running on the home node handling read, read-exclusive, and upgrade requests contribute the most toward the total occupancy of the coherence controller. Further, the number of upgrade requests is usually far fewer than the number of read and read-exclusive requests. Therefore, a system can potentially enjoy a considerable amount of performance improvement with two coherence controllers if there are multiple independent read or read-exclusive requests available at the home node at the same time. One common property of these request handlers is that they require cache line replies and, hence, DRAM reads. In the following discussion, we focus on these types of requests only. As we have already described, the DRAM access is initiated speculatively by the dispatch unit as soon as a request is selected and it can start long before the handler actually executes. Of course, at a later point, the directory lookup may reveal that the data is stale in memory, and the result of the speculative DRAM access may never be used.

There are three parts in the life of a request after it is dispatched, namely, the DRAM occupancy or the DRAM access latency (O_m), the protocol handler occupancy (O_p), and the memory channel occupancy or the transfer time of the cache line from DRAM to the memory controller (O_c). Fig. 2 (scale used: $O_m = 6$, $O_p = 4$, $O_c = 3$) shows the timing diagrams explaining the performance model. We show only the case $O_m > O_p$ since we did not encounter the case

$O_m < O_p$, even in the case of the slowest protocol processor that we consider. We show the timing of two requests scheduled back-to-back at the home node. Time increases on the x-axis from left to right. The starting time of the request i is Si. Here, we assume that $S1 = S2$, although, in practice, there may be a slight time gap due to scheduling delays. The time at which the DRAM access of the ith request finishes is Mi and the time at which the protocol handler for this request finishes is Oi. As soon as the DRAM access finishes, the cache line transfer can start if there is a free channel. We show timing diagrams for single-channel and dual-channel memory controllers. The finishing time of this transfer is denoted by Ci. The single protocol processor case is labeled 1PP while the dual protocol processor case is labeled 2PP. In the 1PP model, the handler for the second request cannot start until timestamp $O1$, but the two DRAM accesses can start at the same time (e.g., at $S1$ or $S2$) as long as they do not suffer from bank conflicts. In Fig. 2, we have assumed this to be the case. Bank conflicts are discussed later. Thus for a single-channel controller $O1 - S1 = O_p$, $M1 - S1 = O_m$, and $C1 - M1 = O_c$. Similar relations exist for the second request. The only difference is that for the 1PP model $O2 - O1 = O_p$, since the second protocol handler cannot start until the first one finishes. It is clear that there is no gain in adding a second protocol processor because in both the 1PP and 2PP models the total time required by two requests is the same (i.e., $C2 - S1$). However, for the 1PP model in cases where $O2 > C2$ (this is not the case in the figure shown), the 2PP model will enjoy performance improvement because the total protocol processing latency in 1PP can no longer be hidden under the memory access and transfer latency. For single-channel memory controllers this condition can be restated as $2O_p > O_m + 2O_c$. In this situation the amount of time saved by the 2PP model would be $2O_p - (O_m + 2O_c)$. For a dual-channel architecture the equivalent condition is $2O_p > O_m + O_c$ and the gain for the 2PP model would be $2O_p - (O_m + O_c)$.

Requests typically arrive at the memory controller in bursts. Therefore, it would be useful to extend this model to a larger number of concurrent requests. Let us assume that at a certain point in time there are k independent requests for read or read-exclusive misses present at the home memory controller. All these k DRAM accesses can be scheduled at the same time as long as k does not exceed the maximum MRQ occupancy and the number of DRAM banks. In the 1PP model, the k coherence protocol handlers must execute sequentially. Therefore, if we want all the protocol processing to take longer than the DRAM accesses and the cache line transfers (this is the situation when 2PP can offer improvement), we must have $kO_p > O_m + kO_c$ for a single-channel architecture. In this situation the k requests spend a total time of kO_p in the memory controller. With 2PP, we can execute the coherence protocol handlers in pairs and, therefore, the total time for k requests is $\max(\frac{k}{2}O_p, O_m + kO_c)$, which is less than kO_p if

$$O_p > \frac{1}{k}O_m + O_c. \quad (1)$$

Note that, for $k = 2$, this bound is exactly as discussed above. Similarly, for a dual-channel architecture, we can derive the inequality $kO_p > O_m + \frac{k}{2}O_c$ or, equivalently,

$$O_p > \frac{1}{k}O_m + \frac{1}{2}O_c. \quad (2)$$

The amount by which O_p actually exceeds the right-hand side will be referred to as the *occupancy margin*. The occupancy margin gives us an indication about the overall gain to expect from adding a second protocol processing unit. Interestingly, as k becomes large (e.g., in heavily banked memory systems with large MRQs running applications with high burstiness), the contribution of DRAM occupancy to these inequalities diminishes. As a result, the usefulness of parallel coherence streams becomes constrained by the proper balance between protocol bandwidth (characterized by O_p) and memory bandwidth (characterized by O_c). Further, observe that, if burstiness is low (i.e., small k), a large positive occupancy margin is unlikely and adding a second protocol processor would not be helpful. These inequalities also bring out the fact that, when adding extra memory bandwidth (or channels), the protocol processing bandwidth may become a bottleneck if the application actually uses the added memory bandwidth (the scaling factor of O_c in (2) is the reciprocal of the total number of memory channels). Thus, this performance model offers a simple way to determine protocol bandwidth requirements. From a designer's viewpoint, one can carry out simulations with single protocol processor, evaluate the average values of O_p and O_m and the maximum value of k (which captures the maximum burstiness), and find out if (1) or (2) holds (note that the value of O_c is fixed by the channel bandwidth). Unless the appropriate condition holds, there would be no benefit of adding a second coherence controller. We will rely on (1) and (2) for analyzing our simulation results.

We conclude the discussion of the model by commenting on cases where consecutive requests access the same DRAM bank, generating a bank conflict and delaying the second DRAM access. We note that, when a hot spot forms at the home node, this is possibly the most common case, since all requests will access the same cache line. Referring back to Fig. 2, we find that in this case the second memory request cannot start until timestamp $M1$ (in practice, a little later due to bank turnaround time). In such a situation, $M2$ will always be greater than $O2$ for 1PP, since $O_m > O_p$ on average. Further, we cannot start transferring the second cache line until the timestamp $M2$. Thus, in this case, the memory latency and transfer latency hide the entire protocol processing latency. Therefore, with bank-conflicting requests two protocol processors will not, in general, provide a visible performance improvement. However, in cases where the second request enjoys a row buffer hit, there may be some benefit if $M2 < O2 - O_c$. But since on average $O_m > O_p$, $M2$ is typically greater than $O2$.

In summary, for bank-parallel requests, having two protocol processors improves performance only if the average occupancy with a single protocol processor exceeds a certain limit determined by the average DRAM latency, memory bandwidth, and burstiness of the application. For applications with high burstiness, only the proper balance between the protocol and the memory bandwidth is important. On the other hand, for bank-conflicting requests, there is little or no benefit from adding a second protocol processor.

4 Evaluation Framework

This section discusses the simulation environment and the applications we use to evaluate the coherence controller architectures. We simulate DSM multiprocessors with

TABLE 1
Simulated Processor Configuration

Parameter	Value	Parameter	Value
Frequency	1.6 GHz	FP div. latency	12 (SP)/19 (DP) cycles
Thread contexts	4 app. + 2 protocol	ITLB, DTLB	128/fully assoc./LRU
Pipe stages	14	Page size	4 KB
Fetch policy	ICOUNT (2 threads)	L1 Icache	32 KB/64B/2-way/LRU
Front-end/Commit width	8/8	L1 Dcache	32 KB/32B/2-way/LRU
BTB	256 sets, 4-way	Unified L2 cache	2 MB/128B/8-way/LRU
Branch predictor	Tournament (21264)	MSHR	16+1 for retiring stores
RAS	32 entries (per thread)	Store buffer	32
Br. mispred. penalty	11 cycles (minimum)	L1 cache hit	2 cycles
Active list	128 entries (per thread)	L2 cache hit	10 cycles (round trip)
Branch stack	32 entries	Reserved (SMTp specific)	
Integer/FP Register	320/320	Front-end slots	1
Integer/FP queue	48/48 entries	Branch stack slots	1
Unified load/store queue	64 entries	Integer registers	16
ALU	8 (two for addr. calc.)	Integer queue slots	12
Integer mult./div. latency	6/35 cycles	LSQ slots	8
FPU	3	Res. MSHR	1
FP mult. latency	2 cycles	Store buffer	1
		Bypass buffer	16 each for instruction and data

TABLE 2
Memory System Configuration

Parameter	Value	Parameter	Value
For non-SMTp only		SDRAM bandwidth	6.4 GB/s (per channel)
Protocol processors	1 or 2	Number of channels	1 or 2 (64 bits each)
Protocol Icache	32 KB/128B/direct mapped	MRQ	16 entries
Protocol Dcache	512 KB/128B/direct mapped	PPWQ entries	1
Common to SMTp and non-SMTp		Virtual networks	4 (protocol uses 3)
System bus width	64 bits	Router ports	6 (SGI Spider)
System bus speed	Same as memory controller	Network topology	2-way bristled hypercube
SDRAM access time	80 ns (row buffer miss)	Hop time	10 ns
	40 ns (row buffer hit)	Link bandwidth	3.2 GB/s

16 nodes, each of which contains an out-of-order SMT processor with four application thread contexts and two additional protocol thread contexts that we enable only in our SMTp results. Therefore, the results present the performance of 64-threaded execution of the application programs discussed in this section. Table 1 shows the MIPS ISA-based simulated processor configuration along with SMTp-specific reserved or extra resources.

The number of physical registers is decided as follows: In addition to the number of active list entries, we provide 32 registers per thread. Thus, for an n-way SMT machine, we provide $32n + 128$ physical registers. Therefore, 128 extra registers are provided for renaming purposes. We configure the register file for a six-way SMT and do not scale it down even if the protocol threads are not active in the embedded protocol processor architectures. Given the large size of the register file, we conservatively select a shallow pipe (14 stages) clocked at 1.6 GHz and simulate a two-cycle pipelined register read. However, at this frequency we can accommodate a 32 KB L1 cache with a two-cycle hit latency, and we found this to be much more beneficial than having a faster clock, a deeper pipeline, and a smaller cache. We will briefly comment on results for higher frequencies in the next section. The number of reserved resources for SMTp shown in the table is decided through simulation.

The details of the memory system are listed in Table 2. The router is assumed to be integrated on chip. We simulate three different memory controller frequencies (reflecting its integration level) when studying the performance of embedded protocol processors: 1.6 GHz, 800 MHz, and 400 MHz. However, the absolute SDRAM access latency, the SDRAM bandwidth, the network hop time, and the network link bandwidth remain unchanged across these configurations. For SMTp, we always assume the integrated memory controller to be clocked at 1.6 GHz, the full processor frequency. We explore the performance of one and two embedded protocol processors. Note that adding a protocol processor does not increase either the SDRAM bandwidth or network bandwidth; it only increases protocol processing bandwidth. We found that more than one PPWQ entry does not affect performance and, therefore, we present results with a one-entry PPWQ.

TABLE 3
Applications and Problem Sizes

Applications	Problem Sizes
FFT	1M points, blocked for DTLB
FFTW	8192×16×16 points, 32×32 block
LU	512×512 matrix, 16×16 block
Radix-sort	2M keys, radix=32
Ocean	514×514 grid, tolerance 1e-5
Water	1024 molecules, 3 time steps

We simulate a 400 MHz SDRAM module and explore configurations with one and two logical channels each capable of transferring 64 bits on both edges of the clock (DDR), the critical 64 bits being the first transfer packet. This leads to an aggregate bandwidth of 6.4 GB/s per channel. A 40-bit physical address is divided into the following parts and decoded accordingly by the memory controller. The least significant 3 bits are offset into an 8-byte column, the next 12 bits are used as the column number, the next 4 bits are the bank address for simulated 16 banks, the next 13 bits are the row address. The upper 8 bits signify the home node number. The mapping of address bits to row, column, and bank is similar to the one suggested in [29]. To avoid bank conflicts between a miss request and a writeback originating from the eviction due to the same miss, the bank number is calculated by XORing the bits [18:15] with bits [21:18] of the address, the latter being the least significant four bits of the L2 tag [35].

Table 3 lists the six explicitly parallel shared memory programs that we use in this paper. FFTW is a 3D Fast Fourier Transform kernel operating on complex double points [8]. The other five applications are chosen from the SPLASH-2 suite [34]. The programs are chosen such that they represent a variety of important scientific computations with different communication patterns and synchronization requirements. The input sizes are chosen to capture realistic machine behavior for these highly scalable shared memory programs (recall we are running 64 threads). All the applications use proper page placement to minimize remote memory accesses, and, where possible, all applications other than Water and FFTW use hand-inserted prefetch and prefetch exclusive instructions to hide cache miss latency.

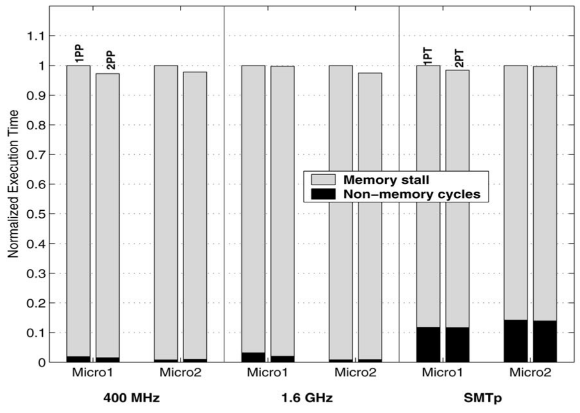
In addition to these applications, we use two microbenchmarks (we will refer to them as Micro1 and Micro2). These microbenchmarks read a matrix of size $4{,}096 \times 4{,}096$ with 8-byte elements. The matrix is divided among 64 threads so that each thread gets 64 contiguous rows. The matrix size is chosen so that the amount of data assigned to each node (i.e., four threads) does not fit in the L2 cache. In Micro1, all pages assigned to a node are allocated locally and the access pattern is such that every access suffers a cache miss. This puts maximum pressure on the coherence controllers. Micro1 accesses one element of consecutive cache lines leading to high amount of DRAM bank conflicts within each thread. However, consecutive accesses from a thread enjoy row buffer hits unless an intervening access from a different thread leads to a row buffer miss. A high amount of bank parallelism is observed across threads. Micro2 accesses data in the same way as Micro1, but pages of the matrix are placed round robin across the nodes. Therefore, only $1/N$ amount of accesses are local, where N is the number of nodes. Note that these microbenchmarks do not have any cross-thread data sharing. Also the data are read-only; therefore, there are no invalidations or interventions during the execution of these microbenchmarks. The microbenchmarks are intentionally kept simple so that we can get some insight into our performance model.

5 Simulation Results

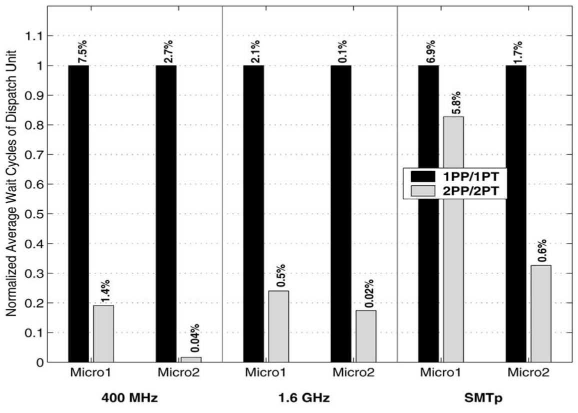
This section details the simulation results and explains them with the help of the performance model developed in Section 3.2. We first discuss the results for the two microbenchmarks and then present, in detail, the performance of the shared memory parallel applications. We close this section with some results on single-node multiprogrammed workloads and a directoryless broadcast protocol over unordered interconnects.

5.1 Microbenchmarks

In Fig. 3a, we present the results for Micro1 and Micro2 with single and dual coherence stream processing for systems with integrated protocol processors clocked at



(a)



(b)

Fig. 3. (a) Performance of single and dual coherence streams in single-channel memory systems. (b) Comparison of wait cycles spent by the dispatch unit with single and dual coherence stream processing.

TABLE 4
Measured Parameters for Single-Channel Systems (O_c Fixed at 20 ns)

Conf.	Param.	400 MHz	1.6 GHz	SMTp	Conf.	Param.	400 MHz	1.6 GHz	SMTp
Micro1					Micro2				
1PPU	O_p (ns)	27.5	6.9	15.0	1PPU	O_p (ns)	15.9	4.1	10.9
	O_m (ns)	65.6	65.5	45.7		O_m (ns)	46.8	46.8	44.3
	$k_{\max}$	14	16	16		$k_{\max}$	10	16	16
2PPU	O_p (ns)	27.5	7.2	25.9	2PPU	O_p (ns)	16.3	4.1	13.8
	$k_{\max}$	15	16	16		$k_{\max}$	14	16	16

400 MHz and 1.6 GHz (results with 800 MHz protocol processors do not add anything new). We also show the results for SMTp. In all cases, the main processor is clocked at 1.6 GHz. The execution time with two protocol processing streams (2PP for embedded protocol processor models and 2PT for SMTp) is normalized to one protocol processing engine (1PP or 1PT, as appropriate). The execution time is broken down into two parts, namely, memory stall cycles and nonmemory cycles. The latter includes busy commit cycles, synchronization cycles, and other resource stall cycles. Although across the coherence controller configurations we expect only the memory stall cycles to vary, the nonmemory cycles may also vary slightly due to different amount of synchronization cycles. Surprisingly, we observe only small gains when adding the second coherence processing engine. To explain this, we present the measured values of O_p, O_m, and $k_{\max}$ for one protocol processing unit (1PPU; can be 1PP or 1PT depending on the architecture), and O_p and $k_{\max}$ for 2PPU (can be 2PP or 2PT depending on the architecture) in Table 4. The values of O_m for 2PPU are similar to those with 1PPU. The value of O_c is fixed at 20 ns, which is the transfer time of a 128-byte cache line over a 6.4 GB/s memory channel. Here O_p is the handler occupancy averaged over all handler invocations across all of the 16 nodes. In Table 4, we notice that, while going from a 400 MHz memory controller to a 1.6 GHz memory controller, O_p decreases roughly by a factor of 4, as expected. Although the protocol threads in SMTp execute at 1.6 GHz, for such a system, O_p is midway between 400 MHz and 1.6 GHz protocol processors. Resource contention in the SMT core, especially involving the front-end buffers, is the main reason for this. In addition, we observe that O_p of a dual protocol thread (2PT) system is more than that of 1PT. This results from further increased resource contention. Recall that the amount of resources given to 1PT and 2PT systems is the same. The drop in the value of O_m seen in SMTp for Micro1 results from reordering of cache miss requests due to timing differences leading to a larger number of row buffer hits.

Table 4 shows that for Micro1, only the architecture with a 400 MHz protocol processor satisfies (1) of Section 3.2, and that too with a small occupancy margin (2.8 ns). Therefore, we expect only a small benefit from adding a second protocol processor in this case. But, in SMTp or 1.6 GHz protocol processor, the performance of Micro1 will not improve by adding a second protocol processor due to large negative occupancy margins. This corroborates the results shown in Fig. 3a. Micro2 does not satisfy (1) for any of the three architectures. However, we find that, if we calculated O_p by considering only the requests received by home nodes (Micro2 suffers from a large number of remote misses due to round-robin page placement), the situation becomes similar to Micro1. To further analyze the results, in Table 5, we show the percentage of coherence transactions handled by the second protocol processor or protocol thread in 2PP or 2PT configuration averaged across 16 nodes. The rest of the transactions are handled by the first protocol processing unit. Recall that the second protocol processor/thread gets a request only if the first protocol processor/thread is busy at the time of the scheduling decision. Except for Micro2 at 1.6 GHz, we find that, in all cases, the second protocol processing unit is moderately loaded. In SMTp for Micro1, the second protocol thread handles almost 50 percent of the requests, but still we do not see much performance improvement with 2PT. This is because, as predicted by our performance model, the low absolute value of O_p is the important determinant of overall performance, and not how often the second protocol processor is invoked.

Fig. 3b shows the number of cycles spent by a request in the PPWQ of the dispatch unit waiting for a free protocol processor or a protocol thread. We present the average across all the nodes normalized to 1PP or 1PT. On top of each bar, we also show this wait cycle count as a percentage of the total execution time. We make two important observations from this figure. First, as expected, the wait cycle count decreases dramatically after adding a second protocol processing unit. Second, even with a single protocol processing unit, the percentage of the wait cycles is always less than 10 percent of the total execution time; Micro1 on a 400 MHz controller is the largest at 7.5 percent. This essentially means that even these microbenchmarks (note from Fig. 3a that these applications spend almost their entire execution time waiting for memory) cannot make protocol processing the bottleneck in the architectures with integrated memory controllers.

Finally, we present the performance results for a dual-channel memory system in Fig. 4. Note that both 1PP/1PT and 2PP/2PT execute with two memory channels. Compared to a single-channel system, Micro1 clearly benefits more from 2PP or 2PT for all three architectures (e.g., 3.7 percent reduction in execution time compared to 2.7 percent in Fig. 3a for 400 MHz memory controllers). Referring back to Table 4 we find that O_p now satisfies (2) by a larger margin for 400 MHz. For SMTp, (2) is also now satisfied by Micro1. Further, Micro2 satisfies this relation

TABLE 5
Coherence Transactions Handled by the Second Protocol Processing Unit (Percentage of All Transactions)

App.	400 MHz	1.6 GHz	SMTp
Micro1	35.4%	33.5%	47.5%
Micro2	23.3%	3.3%	30.0%

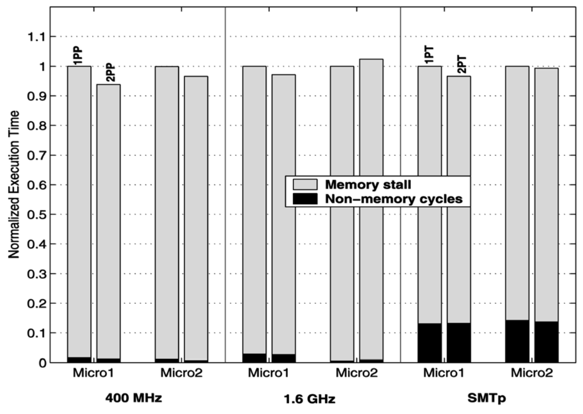


Fig. 4. Performance of single and dual coherence streams in dual-channel memory systems.

for 400 MHz. All these are correctly reflected in the improved performance, as shown in Fig. 4, thereby validating our model. Intuitively, a larger aggregate memory bandwidth can make protocol processing bandwidth a bottleneck more rapidly, but even when doubling the memory bandwidth there is only a minor performance improvement in using multiple coherence processing engines.

5.2 Shared Memory Parallel Applications

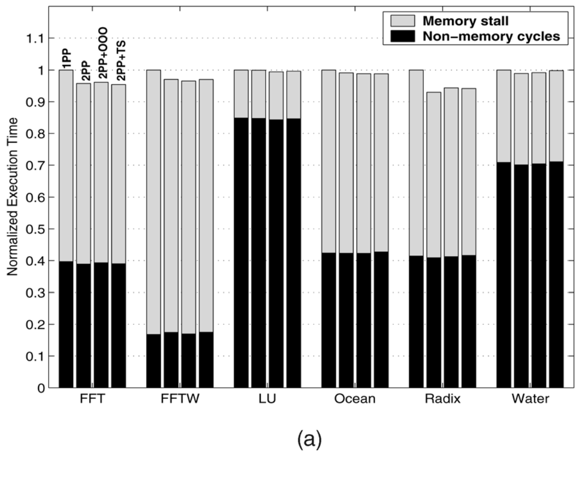
In this section, we present and analyze the simulation results for the six shared memory parallel applications presented in Table 3. First, we present the simulation results for architectures with embedded protocol processors running at 400 MHz, 800 MHz, and 1.6 GHz, and for SMTp. We close this section with an analysis of the results through our performance model.

Fig. 5a shows the execution time for each of the six applications normalized to 1PP for 400 MHz integrated memory controllers. For each application we present results for 1PP, 2PP, 2PP with out-of-order scheduling of coherence requests (labeled 2PP+OOO), and 2PP with test-and-set lock (labeled 2PP+TS). The embedded protocol processors in the 2PP configuration, by default, use a conventional LL/SC lock to protect the critical regions in the coherence protocol code. Radix-Sort enjoys the maximum benefit from multiple protocol processors, although execution time decreases only by 8 percent compared to 1PP. FFT and FFTW observe less than 5 percent improvement in execution time. There is no noticeable gain with out-of-order memory request scheduling. Out-of-order scheduling would be beneficial when the messages at the heads of all the six incoming dispatch queues are suffering from an address conflict with some other outstanding request and there is at least one request in one of these queues that accesses a different address. However, we found that the number of idle dispatch cycles resulting from address conflicts is negligible. The test-and-set lock only nominally improves the execution time of FFT, which means that no special support is needed beyond LL/SC locks.

Table 6 summarizes the percentage of coherence transactions handled by the second protocol processing unit for all four flexible coherence controller architectures. This percentage does not vary much between 2PP, 2PP+OOO, and 2PP+TS. The first column of Table 6 shows that for 400 MHz controllers, in FFT and Ocean the second protocol processor handles roughly 40 percent of all the coherence transactions. These two applications seem to put maximum pressure on the protocol processors, with FFTW and Radix-Sort following closely. Although FFT, FFTW, and Radix-Sort benefit from the addition of a second coherence controller (see Fig. 5a), the overall gain in Ocean is almost zero. We found that, in Ocean, the most frequent coherence transaction is a writeback coming from its large data footprint. As a result, with two protocol processors, even though the overall protocol occupancy decreases, the requests in the critical path do not get accelerated. The exact mix of requests that execute

TABLE 6
Coherence Transactions Handled by the Second Protocol Processing Unit (Percentage of All Transactions)

App.	400 MHz	800 MHz	1.6 GHz	SMTp
FFT	36.5%	30.9%	23.5%	36.5%
FFTW	30.0%	26.0%	20.0%	31.2%
LU	14.1%	11.1%	4.4%	18.8%
Ocean	38.8%	37.2%	34.5%	39.3%
Radix	29.2%	22.0%	16.4%	23.8%
Water	21.5%	15.2%	10.1%	19.6%



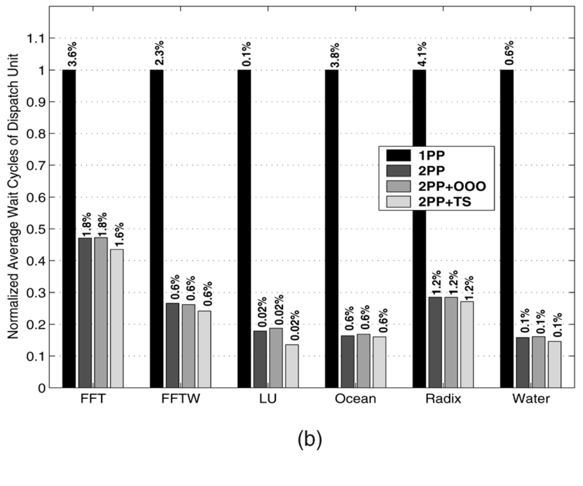


Fig. 5. (a) Performance of one and two protocol processors in single-channel memory systems with 400 MHz integrated memory controllers. (b) Dispatch wait cycles for 400 MHz integrated memory controllers.

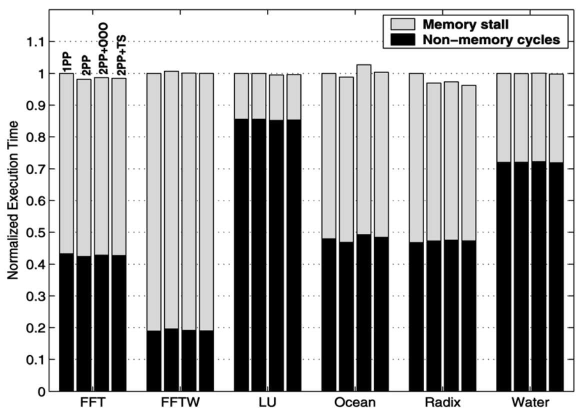


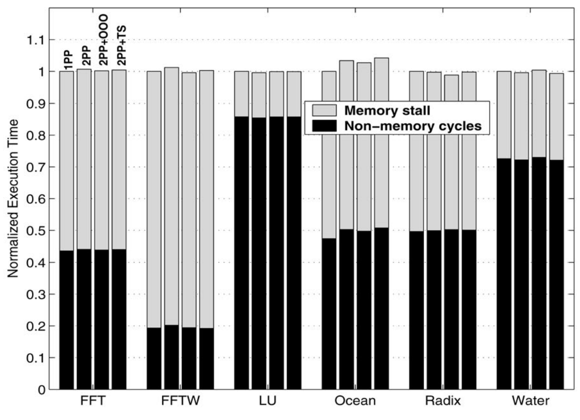
Fig. 6. Performance of one and two protocol processors in single-channel memory systems with 800 MHz integrated memory controllers.

concurrently on the protocol processors is also important for determining the end-performance. Fig. 5b presents the wait cycle count of the dispatch unit normalized to 1PP for 400 MHz memory controllers. Interestingly, the test-and-set lock does reduce the wait cycles noticeably for all applications. This results from reduced protocol occupancy when LL/SC locks are replaced by test-and-set locks. However, this reduction in protocol occupancy fails to make a noticeable impact on execution time. The percentage of wait cycles as shown in Fig. 5b is always less than 5 percent and is much less compared to Micro1 in Fig. 3b. Radix-Sort shows the maximum percentage of wait cycles (4.1 percent) with 1PP, and Ocean (3.8 percent) and FFT (3.6 percent) follow closely. This clearly explains why Radix-Sort enjoys the maximum gain when a second protocol processor is introduced. Radix-Sort is known to experience bursty read-exclusive requests during the histogram permutation phase.

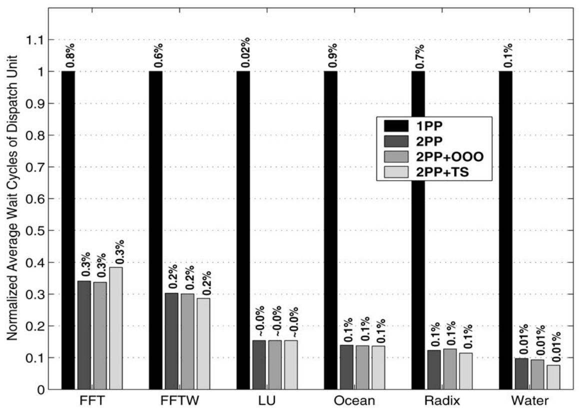
Fig. 6 presents the results for integrated memory controllers running at 800 MHz. Compared to the 400 MHz memory controller configuration (as was shown in Fig. 5a), we observe that the speedup of Radix-Sort has dramatically decreased. This is expected because a faster protocol processor naturally reduces the protocol occupancy leading to a lower occupancy margin. Other than Radix-Sort, only FFT observes a small speedup after introduction of a second coherence controller. Out-of-order coherence request scheduling and test-and-set locks fail to improve performance. In fact, due to an increase in the synchronization time, out-of-order scheduling hurts the performance of Ocean. Although not shown, we found that, as expected, the wait cycle count percentages were roughly half of that shown in Fig. 5b. In fact, for Radix-Sort the wait cycle percentage went down by more than half indicating a nonlinear improvement in protocol occupancy with frequency of the protocol processor.

Figs. 7a and 7b show the results for integrated memory controllers running at 1.6 GHz, which is the processor core frequency. From Fig. 7a, it is clear that there is no need for a second protocol processor in such an architecture. The performance of Ocean degrades after introduction of a second coherence controller due to an altered synchronization timing leading to an increase in the nonmemory cycles. Referring back to Table 6, we see that, with 1.6 GHz controllers for all applications, the pressure on the protocol processing unit has decreased dramatically compared to 400 MHz. This is directly visible from the percentage of requests handled by the second protocol processor. In fact, for compute-bound LU and Water, the second protocol processor remains idle most of the time and handles 5 percent and 10 percent of all transactions, respectively. Ocean continues to be the most heavyweight application in terms of protocol processing bandwidth requirements. However, addition of a second protocol processor does not improve its execution time. Fig. 7b shows the average dispatch wait cycles normalized to 1PP. Although the introduction of a second coherence controller decreases the wait time significantly, the absolute wait cycle percentages are near-zero (less than 1 percent of total execution time for all applications). This clearly explains the performance results of Fig. 7a. Also, we observe that LL/SC locks result in the same level of protocol occupancy as the test-and-set locks at 1.6 GHz.

Figs. 8a and 8b show the results for SMTp. Fig. 8a shows that a second protocol thread does not offer any performance advantage. Table 6 shows that the second protocol thread in SMTp is moderately busy for all the applications. In fact, it is busier than the second protocol processor in 800 MHz embedded protocol processor architecture. This higher occupancy in SMTp results from resource contention among the threads. Fig. 8b shows the average dispatch wait cycles normalized to 1PT. The test-and-set lock offers

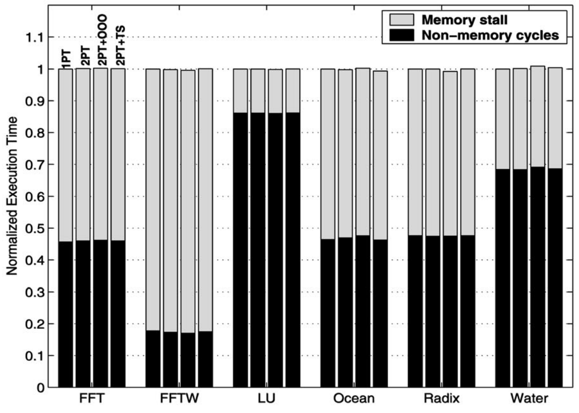


(a)

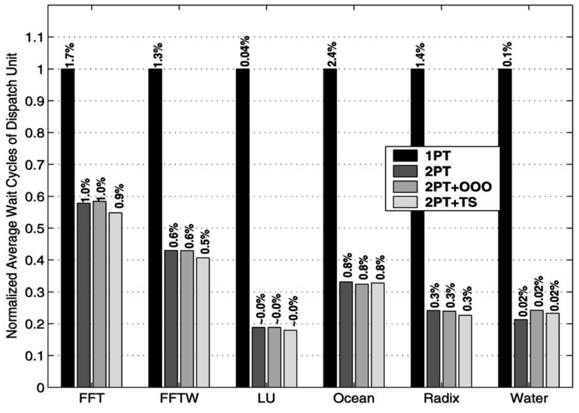


(b)

Fig. 7. (a) Performance of one and two protocol processors in single-channel memory systems with 1.6 GHz integrated memory controllers. (b) Dispatch wait cycles for 1.6 GHz integrated memory controllers.



(a)



(b)

Fig. 8. (a) Performance of one and two protocol threads in single-channel memory systems employing SMTp. (b) Comparison of wait cycles spent by the dispatch unit with one and two protocol threads in SMTp.

TABLE 7
Measured Parameters for Single-Channel Systems

Conf.	Param.	400 MHz	800 MHz	1.6 GHz	SMTp	Conf.	Param.	400 MHz	800 MHz	1.6 GHz	SMTp
		FFT						Ocean			
1PPU	O_p (ns)	30.3	15.3	8.4	15.3	1PPU	O_p (ns)	36.6	18.1	9.4	17.5
	O_m (ns)	54.6	54.4	54.4	55.2		O_m (ns)	54.4	54.7	54.6	55.9
	$k_{\max}$	5	8	12	10		$k_{\max}$	7	16	16	16
2PPU	O_p (ns)	39.1	19.7	10.6	23.1	2PPU	O_p (ns)	39.1	19.4	10.0	24.7
	$k_{\max}$	7	11	14	11		$k_{\max}$	11	16	16	16
		FFTW						Radix-Sort			
1PPU	O_p (ns)	29.4	14.7	7.5	15.6	1PPU	O_p (ns)	33.8	16.9	8.8	16.6
	O_m (ns)	54.3	54.1	53.9	57.6		O_m (ns)	45.8	45.8	45.8	51.8
	$k_{\max}$	7	13	16	14		$k_{\max}$	4	6	8	11
2PPU	O_p (ns)	35.0	17.8	9.4	22.8	2PPU	O_p (ns)	40.0	19.1	9.7	20.6
	$k_{\max}$	10	15	16	15		$k_{\max}$	6	7	8	12
		LU						Water			
1PPU	O_p (ns)	25.3	12.5	6.3	13.1	1PPU	O_p (ns)	28.1	14.1	7.2	14.4
	O_m (ns)	40.1	40.1	40.2	42.3		O_m (ns)	40.0	40.0	40.0	40.1
	$k_{\max}$	6	14	16	16		$k_{\max}$	4	7	8	8
2PPU	O_p (ns)	29.4	14.4	7.2	15.9	2PPU	O_p (ns)	32.2	15.9	8.1	18.1
	$k_{\max}$	12	16	16	16		$k_{\max}$	4	7	8	7

slightly reduced dispatch wait cycle count in FFT and FFTW only. The percentages of the wait cycles for 1PT are larger than those with 1.6 GHz embedded protocol processor (compare with 1PP in Fig. 7b), with Ocean showing the maximum of 2.4 percent, still too small to justify the need for a second protocol thread. However, introduction of a second protocol thread does reduce the wait cycles significantly, as is clearly visible in Fig. 8b.

5.2.1 Model Validation

Having presented all the simulation results, we proceed to analyze these with the help of our performance model. Table 7 presents the measured parameters for all four flexible coherence controller architectures with a single-channel memory controller. In these applications we found quite uneven distribution of $k_{\max}$ across nodes. So we present average $k_{\max}$ in this table. As protocol bandwidth increases (either by frequency doubling or by addition of protocol engines), $k_{\max}$ increases resulting from reduced backpressure on the application threads. We observe that FFT, FFTW, and Ocean show similar values of O_m while LU and Water have O_m close to the row buffer hit latency (40 ns). Due to extremely small data footprints per node and regular sequential accesses in LU and Water, a few row buffer reads are sufficient to satisfy all the memory requests with high hit rates. The value of O_p with one protocol processing unit is consistently smaller than that with two protocol processing units. The main reason for this is a larger number of executed instructions resulting from synchronization overhead in the protocol for dual coherence streams. As discussed in Section 3.1, write contention at the send unit also slightly contributes to this overhead.

Now, we turn to see which applications and configurations satisfy our model relations (1) and (2). From Table 7, we find that only FFTW, Ocean, and Radix-Sort satisfy (1) for a 400 MHz memory controller with one protocol processor while none of the applications satisfy the relation

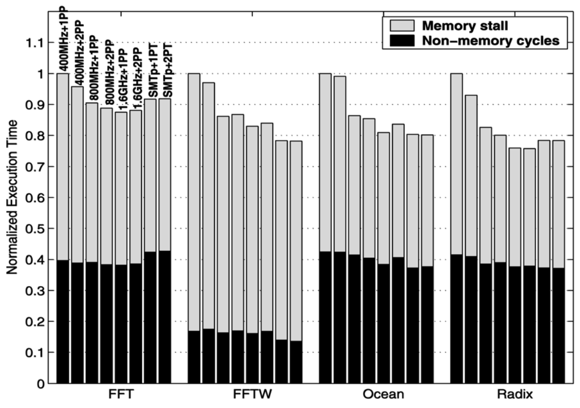


Fig. 9. Summary of results for four communication-intensive applications on single-channel memory controllers.

for memory controllers faster than that, or for SMTp. Further, at 400 MHz, Ocean satisfies the relation by the largest occupancy margin (8.8 ns), followed by Radix-Sort (2.3 ns) and FFTW (1.6 ns). However, we have already mentioned that Ocean fails to benefit from dual protocol processing units due to the presence of a large number of writebacks in the request mix which does not follow the assumed request mix of our performance model. Among Radix-Sort and FFTW, as predicted by the model, Radix-Sort achieves the maximum performance gain from dual coherence streams (as was shown in Fig. 5a). Interestingly, FFT is able to convert a near-zero *negative* occupancy margin (-0.6 ns) into a performance gain of roughly 4 percent (see Fig. 5a). Compared to this, LU and Water show relatively large negative margins (-1.4 ns and -1.9 ns) and fail to get any performance improvement from 2PP, as expected. Although according to the model, Radix-Sort should not benefit from dual protocol processing at 800 MHz, the simulation results show a nominal 3 percent reduction in execution time for 2PP (see Fig. 6). The dominant request type in Radix-Sort is remote read-exclusive. The small amount of parallelism achieved by the concurrent short remote handlers at the requester accounts for this speedup. Our model is not yet equipped to capture this concurrency at nonhome nodes. In summary, it is encouraging to note that the trends dictated by our performance model and the simulation results closely track each other.

5.2.2 *Summary of Results*

In Fig. 9, we summarize all the results for four communication intensive applications. For each application, we show the execution time for one and two protocol processing units corresponding to each of the four architectures (400 MHz, 800 MHz, and 1.6 GHz embedded protocol processors, and SMTp). For each application, the execution time is normalized to the 400 MHz one protocol processor case. We observe that doubling the protocol processor frequency is always more beneficial than adding a second protocol processor. This essentially means that reducing the absolute protocol occupancy of every request is much more important than reducing the total occupancy of a burst of requests depending on the mix of the burst. The most interesting case is Ocean, which does not get any benefit from multiple coherence streams but enjoys large gains from doubling of frequency. This result brings out the importance of proper burst mix.

Interestingly, in the SMTp architecture, FFT and Radix-Sort do not perform as well as the 1.6 GHz protocol processor case, though the performance gap is within 4 percent. The major bottleneck in SMTp comes from contention for one specific front-end queue (between the renamer and the issue queue allocator) and an increased volume of L1 data cache misses. What is noteworthy, however, is that there is never any benefit from adding a second protocol thread in SMTp systems.

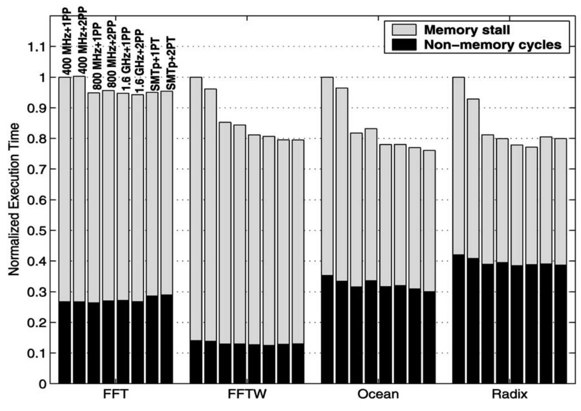
In summary, we observe that as the frequency of the protocol processor increases, the relative gain saturates. This is clearly seen in the case of FFT. To investigate this trend further, we doubled the processor core frequency to 3.2 GHz and experimented with two architectures, one with a 1.6 GHz integrated memory controller and the other with a 3.2 GHz integrated memory controller. We found that the speedup achieved by the 3.2 GHz memory controller compared to the one with 1.6 GHz was, at most, 3 percent (for Radix-Sort). Therefore, at this point, replacing the customized programmable protocol processor or the protocol thread in SMTp by a more powerful hardwired protocol engine (with lower O_p) will not improve overall performance. The option of programmable protocol processing therefore becomes even more attractive in emerging chip-multiprocessor designs where a core can be dedicated to protocol processing.

We also explored the performance of these applications with a dual-channel memory controller (results not shown). There was no major impact of two channels on dual coherence stream architectures compared to single coherence stream architectures. As predicted by our model, FFT, FFTW, and Radix-Sort showed slightly better speedup for 2PP over 1PP at 400 MHz when the second channel is enabled.

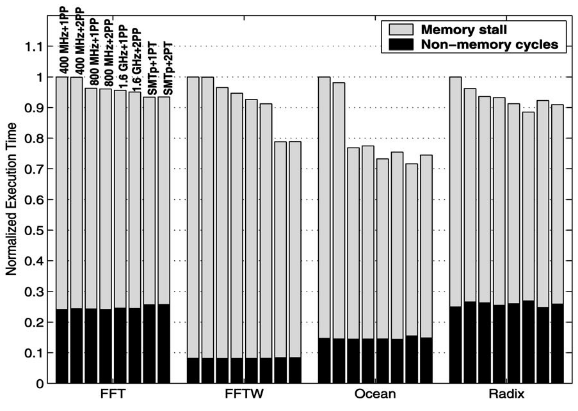
Until now we have focused on 16-node systems running 64 application threads. For completeness, in Fig. 10, we present the results for eight nodes (32 application threads) and one node (4 application threads). We make one important observation from these figures: The protocol processor or protocol thread occupancy is much less a problem in single-node systems compared to 8-node or 16-node systems (see FFTW and Radix-Sort). Although a single-node system puts more pressure on the available processor cache capacity and, hence, incurs more misses compared to an eight-node system, the memory access latency hides the protocol processing occupancy. But, in a multinode system, the occupancy of processing dataless coherence requests cannot be hidden. This is most prominent in Radix-Sort, as seen in Fig. 10.

5.3 Single-Node Multiprogramming

In the previous section, we have seen that protocol processing is not a bottleneck for single-node systems. In this section, we present some results on single-node 4-way multiprogrammed workloads to reinforce this finding. These workloads have much larger volume of cache misses than 4-way threaded parallel applications. We prepared all possible fifteen 4-way multiprogrammed workloads from the six applications presented in Table 3, namely, FFT (F), FFTW (FW), LU (L), Ocean (O), Radix-Sort (R), and Water (W). The same problem sizes shown in Table 3 are used here. We present the results for six selected workloads in Fig. 11. The other workloads do not add anything extra to the explanation of the results. The workload time (shown on the y-axis normalized to 400 MHz 1PP model) is the total time to complete the workload. Among all 15 workloads,

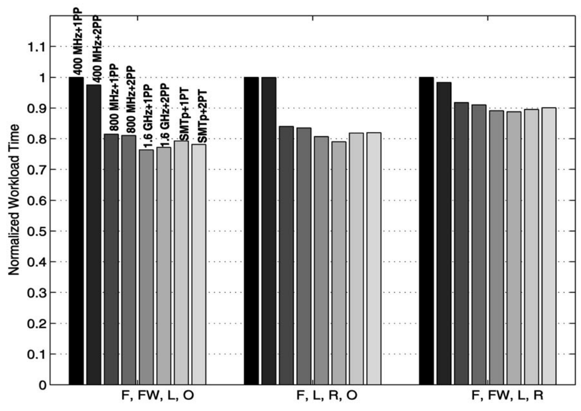


(a)

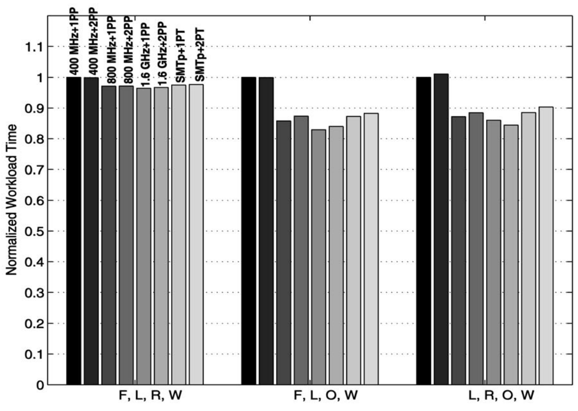


(b)

Fig. 10. Summary of results for four communication intensive applications on single-channel memory controllers: (a) Eight nodes (32 application threads). (b) One node (4 application threads).



(a)



(b)

Fig. 11. Summary of results for selected six multiprogrammed workloads on single-channel memory controllers.

SMTp performs worst on the mix of FFT, LU, Ocean, Water (the second workload in Fig. 11b) when compared to a 1.6 GHz protocol processor configuration. We make two major observations from these results: First, SMTp is performing satisfactorily compared to 1.6 GHz protocol processor models. Second, as expected, a second protocol processing unit (processor or thread) does not help improve the performance significantly. However, increasing the protocol processor's frequency up to 800 MHz continues to be beneficial for all of the workloads except the mix of FFT, LU, Radix-Sort, Water (the first workload in Fig. 11b).

5.4 Directoryless Broadcast Protocols

For completeness, in this section, we present some results on the impact of parallel snoop engines for a Hammer-like broadcast protocol [2], which sends every request to the home node and the home node broadcasts the request to all other nodes. Every node sends a reply to the requester indicating its local snoop result, which may carry data if the cache block is dirty in that node. The home node also sends a reply either from its local cache (if dirty) or from main memory. The requester combines all these replies appropriately (i.e., uses home's reply if no other node replies with data or uses data from the dirty node). We simulate a 16-node DSM machine running this protocol, each node executing four application threads. Both the main processor and the snoop engines are clocked at 2.4 GHz (mimicking the recent models of the AMD Opteron).

We find that, averaged over the four communication-intensive applications (FFT, FFTW, Radix-Sort, Ocean), this system experiences 13.9 times more coherence processing invocations (including requests, snoop responses, and other coherence messages) than the directory-based protocol. Addition of a second snoop engine, on average, improves performance by 16.1 percent compared to one snoop engine. However, out-of-order request scheduling does not offer much performance improvement in this protocol either. Interestingly, the number of endpoint messages per L2 cache miss in the broadcast protocol is, on average, 26.1, as opposed to 2.5 in the directory-based protocol for these four applications. Therefore, as expected, due to a reasonable amount of message concurrency, this class of protocols finds parallel coherence streams much more beneficial than the traditional directory-based protocols.

6 Related Work

Programmable coherence controllers have been studied and designed by several research groups. Some of these are customized protocol processors e.g., the Piranha chip multiprocessor [3], Opteron-Horus [15], Stanford FLASH multiprocessor [17], Sequent STiNG [20], and Sun S3.mp [26], while others use commodity off-the-shelf processors e.g., Typhoon [28]. The Stanford FLASH team reported a 12 percent performance loss compared to a hardwired controller while for the Wisconsin Typhoon the corresponding gap was less than 20 percent. None of these studies, however, considered an integrated memory controller with embedded protocol processors.

An exhaustive study of several address partitioning schemes for static and dynamic mapping of transactions to multiple off-chip coherence controllers in DSM multiprocessors with SMP nodes has been presented in [27]. A performance comparison of multiple hardwired coherence controllers and multiple off-the-shelf protocol processors has been presented in [24]. Finally, the advantage of pipelining [24], superpipelining, nonblocking execution, and directory/tag prefetches has been explored in the context of coherence controller microarchitecture [25]. All these studies consider coherence controllers much slower than what can be achieved if they are integrated on-chip. Multiple coherence engines with local/remote address partitioning are studied in [23], [24] and are implemented in the Opteron-Horus [15], the Sequent STiNG [20], and the Sun S3.mp [26]. Some of these studies include a large L3 cache per node to confine the capacity-related remote traffic to local nodes as much as possible. Our analytical model continues to hold for such systems, where the value of k may decrease depending on the burstiness of the remaining requests leading to a smaller occupancy margin compared to a similar system without L3 caches.

A parallel dispatch queue programming model is presented in [7] for expressing the mutual exclusion among shared resources in fine-grain software coherence protocols at dispatch time as opposed to in-handler synchronization with spin locks. In our study, we explore the use of two standard spin lock mechanisms for protecting the critical sections in coherence handlers.

7 Conclusions

For the first time, this paper presents and validates through simulation a useful analytical model to determine protocol processing bandwidth requirements in contemporary distributed shared memory multiprocessors with integrated memory controllers and programmable coherence protocol engines (either protocol cores or protocol threads) that implement directory-based coherence protocols. The analytical model reveals two important insights. First, a system suffers from a protocol processing bandwidth shortage only when the average protocol occupancy exceeds a value determined by the average DRAM access latency, the memory bandwidth, and the burstiness of the applications. For applications with high burstiness, the proper balance between protocol processing bandwidth and memory bandwidth is the important determinant of performance. In such cases, the DRAM latency is not a critical factor. Second, request streams with large amounts of DRAM bank parallelism benefit most from parallel coherence handling provided the first condition holds, i.e., the application is already in need of more protocol processing bandwidth. Interestingly, bank-conflicting request streams enjoy little benefit from parallel coherence processing. Unfortunately, this is the most frequent case when a hot spot arises at a node due to accesses to a single cache line possibly holding a "hot" variable such as a lock or a flag. Parallel coherence engines cannot reduce this hot spot in modern systems with integrated memory controllers.

Simulation results with 64-threaded parallel applications validate these hypotheses and show that, in a 16-node DSM multiprocessor built from 1.6 GHz 4-way SMT nodes with integrated memory controllers, there is little or no benefit from adding a second protocol processing engine when the memory controller is clocked at the same frequency or at half the frequency of the processor core. Only when the processor core's frequency is at least four times the memory controller's do we see minor performance benefits (at most, 8 percent) from adding a second protocol engine. Further, when the balance between the protocol bandwidth and the memory bandwidth is slightly upset by adding an extra DRAM channel, we also observe some minor performance improvement, but only when the application itself can take advantage of the added memory bandwidth. Finally, as expected, we find that systems running directoryless broadcast protocols enjoy a significant performance benefit from parallel snoop engines (on average, 16.1 percent) for four of our communication-intensive applications.

In summary, the major contribution of this work is that it shows a directory-based DSM designer a systematic, simple, and abstract way to determine protocol bandwidth requirements when designing future parallel architectures. Since, in modern architectures running scalable directory-based coherence protocols, one protocol engine clocked at the main processor frequency is sufficient in terms of protocol processing bandwidth, a DSM designer can optimize the amount of resources devoted to protocol processing by using a single SMTp-based or CMP-based protocol engine.

We would like to mention two important extensions to this work that are worth exploring. Inclusion of embedded DRAMs on the die may require reengineering of the proposed model depending on the position of the coherence controller(s) in the path of a cache/coherence miss. More importantly, our model will find applications to determining the appropriate number of coherence controllers in a directory-based shared nonuniform L2 cache of a chip multiprocessor with private L1 caches. As far as the analytical model is concerned, one major difference in such an architecture compared to what we have presented in this paper is that the L2 bank access latency is much smaller than the DRAM bank access latency (O_m). Similarly, O_c is also much smaller. According to our model, in such architectures the occupancy margin per L2 bank may be quite large depending on the value of k, thereby requiring more than one coherence controller per L2 bank.

Acknowledgments

Mainak Chaudhuri would like to thank Anshuman Gupta and Varun Khaneja for help with the initial simulations and development of the broadcast protocol, respectively. The authors extend special thanks to the Security Center of IIT Kanpur for offering a quad Opteron used to run some of the simulations.

REFERENCES

[1] A. Agarwal et al., "The MIT Alewife Machine: Architecture and Performance," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 2-13, June 1995.

[2] A. Ahmed et al., "AMD Opteron Shared Memory MP Systems," *Proc. 14th Hot Chips Symp.,* Aug. 2002.

[3] L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Int'l Symp. Computer Architecture,* pp. 282-293, June 2000.

[4] M. Chaudhuri et al., "Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation," *IEEE Trans. Computers,* vol. 52, no. 7, pp. 862-880, July 2003.

[5] M. Chaudhuri and M. Heinrich, "SMTp: An Architecture for Next-Generation Scalable Multi-Threading," *Proc. 31st Int'l Symp. Computer Architecture,* pp. 124-135, June 2004.

[6] Z. Cvetanovic, "Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor," *Proc. 30th Int'l Symp. Computer Architecture,* pp. 218-228, June 2003.

[7] B. Falsafi and D.A. Wood, "Parallel Dispatch Queue: A Queue-Based Programming Abstraction to Parallelize Fine-Grain Communication Protocols," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture,* pp. 182-192, Jan. 1999.

[8] M. Frigo and S.G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proc. 23rd Int'l Conf. Acoustics, Speech, and Signal Processing,* pp. 1381-1384, May 1998.

[9] M. Heinrich et al., "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 274-285, Oct. 1994.

[10] Z. Hu, M. Martonosi, and S. Kaxiras, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," *Proc. 29th Int'l Symp. Computer Architecture,* pp. 209-220, May 2002.

[11] R. Kalla, B. Sinharoy, and J.M. Tendler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro,* vol. 24, no. 2, pp. 40-47, Mar.-Apr. 2004.

[12] C.N. Keltcher et al., "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro,* vol. 23, no. 2, pp. 66-76, Mar.-Apr. 2003.

[13] "KSR1 Technical Summary," technical report, Kendall Square Research, 1992.

[14] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro,* vol. 25, no. 2, pp. 21-29, Mar.-Apr. 2005.

[15] R. Kota and R. Oehler, "Horus: Large-Scale Symmetric Multiprocessing for Opteron Systems," *IEEE Micro,* vol. 25, no. 2, pp. 30-40, Mar.-Apr. 2005.

[16] D. Koufaty and D.T. Marr, "Hyperthreading Technology in the Netburst Microarchitecture," *IEEE Micro,* vol. 23, no. 2, pp. 56-65, Mar.-Apr. 2003.

[17] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.

[18] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 241-251, June 1997.

[19] D. Lenoski et al., "The Stanford DASH Multiprocessor," *IEEE Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.

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

[21] M.M.K. Martin, M.D. Hill, and D.A. Wood, "Token Coherence: Decoupling Performance and Correctness," *Proc. 30th Int'l Symp. Computer Architecture,* pp. 182-193, June 2003.

[22] C. McNairy and R. Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," *IEEE Micro,* vol. 25, no. 2, pp. 10-20, Mar.-Apr. 2005.

[23] M.M. Michael et al., "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 219-228, June 1997.

[24] A.K. Nanda et al., "High-Throughput Coherence Controllers," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* pp. 145-155, Jan. 2000.

[25] A.-T. Nguyen and J. Torrellas, "Design Trade-Offs in High-Throughput Coherence Controllers," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 194-205, Sept.-Oct. 2003.

[26] A. Nowatzyk et al., "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. 24th Int'l Conf. Parallel Processing,* vol. 1, pp. 1-10, Aug. 1995.

[27] I. Pragaspathy and B. Falsafi, "Address Partitioning in DSM Clusters with Parallel Coherence Controllers," *Proc. Eighth Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 47-56, Oct. 2000.

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

[29] S. Rixner, "Memory Controller Optimizations for Web Servers," *Proc. 37th Int'l Symp. Microarchitecture,* pp. 355-366, Dec. 2004.

[30] "An Overview of UltraSPARC III Cu," white paper, Sun Microsystems, http://www.sun.com/processors/whitepapers/USIIICuoverview.pdf, Sept. 2003.

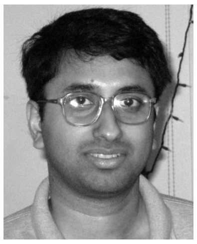
[31] "UltraSPARC IV Processor Architecture Overview," white paper, Sun Microsystems, http://www.sun.com/processors/whitepapers/us4_whitepaper.pdf, Feb. 2004.

[32] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 392-403, June 1995.

[33] D.M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Int'l Symp. Computer Architecture,* pp. 191-202, May 1996.

[34] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

[35] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," *Proc. 33rd Int'l Symp. Microarchitecture,* pp. 32-41, Dec. 2000.



Mainak Chaudhuri received the PhD degree from Cornell University in 2004. He is an assistant professor of computer science and engineering at the Indian Institute of Technology, Kanpur. His primary research interest is in parallel computer architectures. He is a member of the IEEE and the IEEE Computer Society.

Mark Heinrich received the PhD degree in electrical engineering from Stanford University in 1998, the MS degree from Stanford in 1993, the BSE in electrical engineering and computer science from Duke University in 1991. He was a principal designer of the FLASH multiprocessor. He is an associate professor and the associate director of the school of electrical engineering and computer science at the University of Central Florida (UCF), and a founder of its Computer Systems Laboratory. His research interests include novel computer architectures, parallel computer architecture, high-throughput computing, scalable cache coherence protocols, and active memory and I/O subsystems. He is the recipient of an NSF CAREER Award, an IBM Faculty Award, and he is a senior member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.