Journal of Parallel and Distributed Computing **57**, 14–32 (1999)
Article ID jpdc.1998.1524, available online at http://www.idealibrary.com on IDEAL®

Coherence and Replacement Protocol of DICE—A Bus-Based COMA Multiprocessor

Sangyeun Cho* and Jinseok Kong

Department of Computer Science and Engineering, University of Minnesota, 4-192 200 Union Street SE, Minneapolis, Minnesota 55455
E-mail: {sycho,jkong}@cs.umn.edu

and

Gyungho Lee

Division of Engineering, University of Texas–San Antonio, 6900 North Loop 1604 West, San Antonio, Texas 78249-0665
E-mail: glee@voyager1.eng.utsa.edu

Received June 2, 1997; revised December 29, 1997; accepted November 16, 1998

As microprocessors become faster and demand more bandwidth, the already limited scalability of a shared bus decreases even further. DICE, a shared-bus multiprocessor, utilizes cache only memory architecture (COMA) to effectively decrease the speed gap between modern high-performance microprocessors and the bus. DICE tries to optimize COMA for a shared-bus medium, in particular to reduce the detrimental effects of cache coherence and the "last memory block" problem on replacement. In this paper, we present the coherence and replacement protocol of the DICE multiprocessor and its design trade-offs. We describe a four-state write-invalidate coherence protocol in detail. Replacement, which poses a unique overhead problem of COMA, requires that a victim block with ownership be relocated to a remote node in order not to discard the last cached memory block. We show that the relocation process can be efficiently implemented by using a temporary storage called relocation buffer and a priority-based selection algorithm. We present performance results that show a drastic reduction in global bus traffic compared to a traditional shared-bus multiprocessor architecture. © 1999 Academic Press, Inc.

Key Words: distributed shared memory (DSM); symmetric multiprocessor (SMP); shared bus.

* Corresponding author. Current address: Sangyeun Cho, Samho 3rd Apt. B-604, Seocho-gu Panpo-1-dong, Seoul 137-041, Korea.

1. INTRODUCTION

Shared-bus multiprocessors such as Sequent Symmetry [24] or SGI Challenge [8] represent a mainstream of accepted and commercially viable computer systems. However, as microprocessors become faster and demand aggressive data bandwidth, the already limited scalability of the shared bus decreases even further. The effective machine size for shared-bus multiprocessors is fairly limited, typically to less than 20 processors, and a cache miss can cost up to a few hundred processor cycles for high-performance microprocessors today. A recent measurement using a real machine shows that a four-processor shared-bus multiprocessor with 1-MB L2 caches experiences more than 60% longer memory latency compared with the same machine with one processor [14]. The situation is exacerbated when new, high-bandwidth memory technologies, such as Rambus DRAM [7], are used in main memory. To bridge the speed gap between high-performance microprocessors and a backplane bus, it is important to reduce global bus traffic and to increase local memory utilization, together with efforts to develop a high-speed wide data-path backplane bus.

The DICE (direct interconnection of computing elements) project at the University of Minnesota [17, 18] utilizes *cache-only memory architecture* (COMA) to bridge the gap. COMA improves utilization of local memory by decoupling the address of a datum from its physical location, allowing it to move dynamically beyond the level provided by traditional caches. This decoupling is achieved by treating the memory local to each node, called *attraction memory* (AM), as a cache to the shared address space without providing traditional physical main memory [9]. COMA is similar to *shared virtual memory* (SVM) that allows sharing of virtual memory space through migration and replication of pages [23], but it is a more hardware-oriented approach with a sharing unit called *memory block* that is of finer granularity, rather than the page in SVM.

In a COMA machine, most of the capacity misses in processor caches will hit in the AM due to its large size, leading to reduced miss penalty. Also, there will be no write-back traffic in the global interconnection on cache replacements since a write-back operation is performed locally. The higher utilization of local memory can lower the average memory access time and global traffic. Unlike the previous examples of scalable COMA machines, including DDM [9] and KSR-1 [28], DICE focuses on improving a bus-based *symmetric multiprocessor* (SMP) via an efficient realization of COMA with little provision for scalability for larger scale multiprocessing. While we expect that many problems associated with scalable COMA machines will become less serious with a bus, shared-bus multiprocessors benefit from COMA in three ways: (i) less bus contention due to lower global traffic; (ii) shorter average memory latency due to high local memory utilization; and (iii) more processors in the machine due to less bandwidth requirement on the bus.

DICE tries to optimize COMA for a shared-bus medium, in particular to reduce the detrimental effects of cache coherence and the last memory block problem on replacement. This paper presents the coherence and replacement protocol of DICE in detail with our priority-based relocation scheme [19]. We show how the relocation process can be efficiently implemented by using a temporary storage called *a* *relocation buffer* and a priority-based selection algorithm. DICE demonstrates that an efficient shared-bus multiprocessor based on COMA can be realized with very little additional hardware complexity. In addition, we compare the potential performance of the DICE multiprocessor with a traditional shared-bus multiprocessor model based on program-driven simulations. Our performance study confirms the observation that COMA provides an excellent opportunity to significantly reduce the global bus traffic and the average memory access latency [20]. We observed global bus traffic reduction of up to almost 80% with an average of 68% for 16 processors in our performance study. A shared-bus COMA SMP model has also been studied by other researchers [16] and their results also support our claim of its effectiveness over traditional shared-bus SMPs.

The rest of this paper is organized as follows. Section 2 gives a brief background necessary for our discussions. Section 3 and 4 describe the coherence protocol and the replacement protocol of the DICE multiprocessor, respectively. We present a simulation study and the results in Section 5, followed by Section 6 which concludes the paper.

2. BACKGROUND

2.1. COMA and AM Coherency

The AM of a COMA machine seems to fit very well into a traditional memory hierarchy. When a memory reference misses both in the processor cache(s) and the local AM, a copy of the block containing the data is fetched over the network from a remote AM. It is then placed in the block frame of the local AM that the block maps to. When a page fault occurs, the page will be brought from the backing store of disks into the AM of the particular node that caused the page fault. All data, therefore, tend to replicate and migrate to the nodes accessing the data.

Although the problem of maintaining coherence in a COMA multiprocessor is similar to that in a traditional shared-memory multiprocessor, there are a few aspects that differentiate the AMs from the cache memory in traditional cache-coherent multiprocessors. First, the backing store of the AMs in a COMA machine is usually composed of (slow) disks of secondary storage. Thus, unlike a traditional multiprocessor cache, write-backs to the backing store on memory block replacement should be avoided. This creates a unique problem in a COMA machine; the replaced block may need to relocate to some other AM which has space for it to avoid a disk write-back, since it may be the last valid copy in the system. Finding a node whose AM has space for the replaced block can cause a serious overhead [13, 25]. Second, since a COMA machine tries to maximize the utilization of the memory local to a processing node by caching the whole working set, not a portion of it, a typical AM is much larger than a traditional cache memory. Although this huge size of AM will remove capacity misses, it can create more coherence activities [18]. Third, the AM plays its role at a different level of memory hierarchy. The memory requests that reach the AM are the ones missed at a traditional cache memory of substantial size, even perhaps of a multilevel structure. Further, it is important to have some of the physical storage space in the AMs left unallocated, i.e., not utilized as a part of the physical address space. For example, a COMA machine of 16 nodes with 64 MB of AM per node may have only 512 MB for its physical address space, leaving 512 MB unallocated for data replication. Without enough unallocated space, excessive migration and replacement of memory blocks between the AMs can result. Proper reservation of the unallocated space needs to be done in consideration with the set-associativity of the AM, which can be handled by an operating system with appropriate hardware support [12, 13].

With dynamic replication and migration of data through the AMs, a COMA machine can provide higher utilization of local memory than is otherwise possible, which will result in lower average memory access latency and network traffic. As the processor technology is progressing much faster than the bus or interconnection network technology, this potential reduction in latency and bandwidth requirement can be a crucial advantage.

2.2. *DICE—a Bus-Based COMA Multiprocessor*

The DICE architecture described here is a basis for later discussions. Our discussions are not restricted to a specific implementation; however, possible bus and node designs of a bus-based COMA multiprocessor can be found in [16, 21, 22].

Figure 1 shows a high-level structure of a bus-based COMA multiprocessor. A processor node is composed of a high-performance microprocessor, two levels of cache memory, and the local memory managed as the AM. The local memory tag, which includes "state" information and uses fast SRAMs, is duplicated so that local tag access and global bus snooping will not conflict too often. The inclusion property [1] is maintained in the memory hierarchy.

As in a traditional shared-bus machine, each node snoops all global bus traffic. In dealing with the large AM, it can be challenging to have a snoop control logic that can keep up with a modern backplane bus with a high clock frequency, especially if the memory access model is based on sequential consistency [15]. For example, in the SGI POWERpath-2 bus [8], each transaction takes five clock cycles of the 47.6 MHz clock, and the snooper has about two cycles to search the state and tag memory for its AM and update the state if necessary. With the fast SRAMs currently available, the snooper can manage to keep the AMs of a COMA machine coherent. However, if the snooper cannot keep up with the fast clock of the bus, one can relax the memory model and delay the snooping activity with

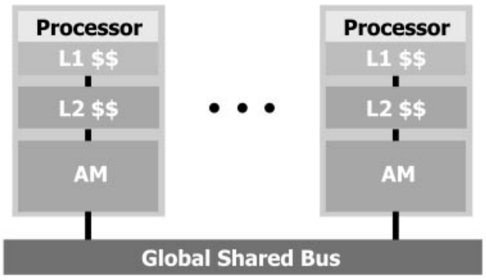

FIG. 1. High-level structure of a bus-based COMA multiprocessor.

request buffering [5]. The global bus supports split-transactions and distributed arbitration for various transactions. Distributed arbitration is important not only for greater scalability but also for our priority-based relocation algorithm.

The impact of COMA on the shared bus was first studied by Lee and Kong [20], where it was shown that on average COMA had about 40% of the bandwidth requirement on the global bus when compared with a conventional UMA model, assuming 16 processors. A bus-based COMA multiprocessor was studied also by Landin and Dahlgren [16]. Using detailed execution-driven simulations, the study reported a significant traffic reduction of up to 70%, with an average of 46%, and an average execution time reduction of 32% for the benchmark programs examined. A global bus design for a bus-based COMA multiprocessor based on the Futurebus+ standard backplane bus was presented by Lee *et al.* [21], where they showed that a bus-based COMA multiprocessor can be built efficiently using off-the-shelf components with little additional hardware complexity.

3. COHERENCE PROTOCOL

In this section, we describe a four-state write-invalidate coherence protocol for the AM in the DICE multiprocessor. We give the definitions of states and events in the first subsection. A detailed description of the state transitions and the actions taken to enforce coherence will be given in the second subsection, followed by a discussion.

3.1. Preliminaries

Figure 2 depicts the coherence protocol. An AM block can be in any one of the four states: *invalid* (INV), *shared non-owner* (SHN), *shared owner* (SHO), and *exclusive* (EXL). The INV state tells that the block contains no valid data. The SHN state is a nonowner state and guarantees that the block in this state is not the only copy in the system. The SHO state is an owner state and carries an ambiguity that there

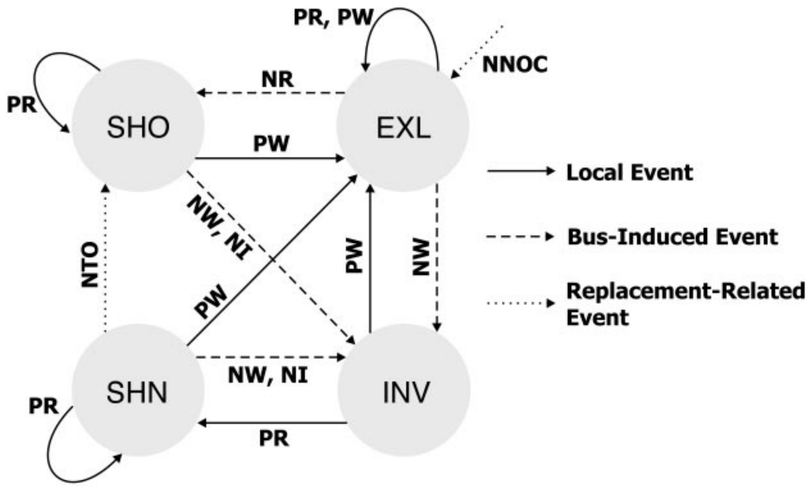

FIG. 2. A four-state write-invalidate coherence protocol of DICE.

may be no other copy in the system. The EXL state guarantees that the block in the state is the only copy in the system and ownership is implicitly assumed. A reference is said to "hit" if the target block exists in the local AM in a valid state other than INV. Note that there is no "dirty" bit associated with any of the states, since there is a unique owner block for every cached block in the system. The ownership associated with a block in the SHO or EXL state entitles the caching node with a responsibility to supply requesting nodes with data.

The following local events are considered: *processor read* (PR) and *processor write* (PW). We assume that a page fault will be handled by the operating system, and when a page is fetched from the disk, all the blocks in the page will be initially in the EXL state. Depending on how one wants to handle I/O operations and lock operations such as *test-and-set*, PR and PW may be classified further to distinguish normal memory operations from I/O or lock operations. In this section we limit our discussion to normal memory operations for simplicity. However, extending the discussion to allow I/O or lock operations on top of normal memory operations should be straightforward.

Bus-induced events are as follows: *network read request* (NR), *network write request* (NW), and *network invalidation* (NI). NR and NW are data requests to satisfy a remote miss. NI is incurred when a processor writes to a shared block, i.e., a block in the SHN or SHO state. Upon receiving an NI transaction specifying a block, each node that cached the corresponding block invalidates its copy. To maintain sequential consistency, an NI transaction can complete only when every node finishes the operation and responds to it, while in relaxed consistency models, an NI transaction completes as long as each and every node latches the address.

Replacement in a COMA multiprocessor may lead to data transfer and state transition. The following events are related to replacement: *network transfer of ownership* (NTO) and *network no other copies* (NNOC). When a block in the SHO state is replaced and relocated to a remote node that contains a shared copy of the block, an NTO event is said to occur to the shared copy of the remote node. When a block in the SHO or EXL state is replaced, and if there is no other copy in the system (which is always the case for the EXL state), an NNOC event is said to occur to the node which accepts the replaced block. The SHO state has ambiguity on replacement, since it is not known if there is a shared copy. However, the DICE replacement protocol distinguishes between the two cases using distributed arbitration of the shared bus.

3.2. *State Transitions and Actions*

A read miss generates a read request on the bus and the owner of the block will supply the data. The new block will always be in the SHN state unless there was a page fault because the block is supplied by a remote AM. In the process, the owner that supplies the data will move to the SHO state if its original state was EXL.

A write reference that hits in the local memory may or may not cause an explicit invalidation depending on the state of the block. If the block is in the SHN or SHO state, an NI transaction will be launched on the bus to invalidate all other copies and the block will be in the EXL state thereafter. A write miss will generate a write request on the bus and the owner of the block will supply the data. All the other copies of the block including the owner's are invalidated, and the block supplied by the owner will be in the EXL state.

When a read or write request is seen on the bus, every node will snoop the request and look up its state and tag memory to see if it has the corresponding block with ownership. As the result of this look-up, the node in charge of supplying the data is uniquely determined.

When a block is replaced and relocated to a remote AM, NTO and NNOC events determine the actions taken and the resulting state of a relocated block. An NTO event occurs when a block in the SHO state is replaced and there exists at least another copy in the SHN state. The relocation process will effectively transfer ownership to an "already-there" copy in a remote node which is chosen on a priority basis. The shared block (in the SHN state) now changes to the SHO state, and the replaced block is not written in the remote AM.

An NNOC event is said to occur when there is no shared copy of a block to be relocated. There are two cases for NNOC: (i) when a block in the EXL state is replaced and (ii) when a block in the SHO state is replaced and no other copy in SHN exists. The second case is possible since blocks in the SHN state can be replaced locally at any node. A replaced block in the SHO state, with no shared copy elsewhere, should move to a remote node and is required to be written in the AM. The resulting state will be EXL.

3.3. *Discussion*

The ownership associated with each memory block is very important to the coherence mechanism in the DICE architecture. Because the ownership per block uniquely determines which node is responsible for providing data, the "selection" phase of DDM [9] is unnecessary.

Although we have presented an invalidation-based coherence protocol in this paper, an update-based or hybrid protocol can become attractive when applications show a lot of write sharing. Our simulation results in Section 5.2 show that coherence misses due to previous invalidations can constitute a large portion of or even dominate the overall misses in a bus-based COMA multiprocessor.

As hinted in the previous subsections, the coherence protocol of bus-based COMA multiprocessors is closely connected with the replacement problem. The mechanism of the replacement and associated coherence maintenance will differentiate the DICE architecture from conventional shared-bus multiprocessors, as discussed in greater detail in the next section.

4. REPLACEMENT PROTOCOL

4.1. *Victim Block Selection*

On a reference miss, a (victim) block in the set to which the reference maps needs to be selected to accept the incoming data. Let us assume that the AM is set-associative and there are more than one victim candidates. The states of the candidate blocks are used to choose the victim prioritized in the following order: INV, SHN, SHO, EXL.

Our first choice is naturally a block in INV state. The second choice is a block in the SHN state, which can be overwritten safely. Note that in the first two cases relocation to a remote node is not necessary. A block in the SHO state is the third choice that will incur a relocation process. If there is a copy of the block in the SHN state in a remote node, the relocation activity will effectively transfer the ownership possessed by the original block (NTO event in Fig. 2). It is possible, however, that there is no other copy residing in the system due to replacement of SHN blocks. A block in the EXL state always needs to be relocated.

There are two advantages in preferring a block in the SHO state to a block in the EXL state when selecting the victim. First, relocating a block in the EXL state may force a node which has nothing to do with the block into receiving it. This may again replace a valid block in the remote node, possibly decreasing the hit rate. However, if a block in the SHO state is selected, there is a high probability that a shared copy exists in a remote AM and thus a third party will not be dragged into participation. Second, if there is a shared copy in the system, the relocation of a block in the SHO state is effectively a transfer of the ownership only, and data write (AM update) can be avoided. The relocation of a block in the EXL state, however, always updates a remote AM usually composed of (slow) DRAMs, and it can be on the critical path of the remote processor.

Figure 3 shows the results of this priority-based victim selection scheme, assuming a four way set-associative AM. Victim candidates after the priority-based selection are marked with darker block, and the victim is selected randomly, should there be more than one candidate. Note that this selection and data movement activity can be done in parallel with the request for the missing block. While the processor sits idle waiting for the missing block, the victim block can be fetched from the AM and placed in the relocation buffer, thus hiding a portion of the latency due to the

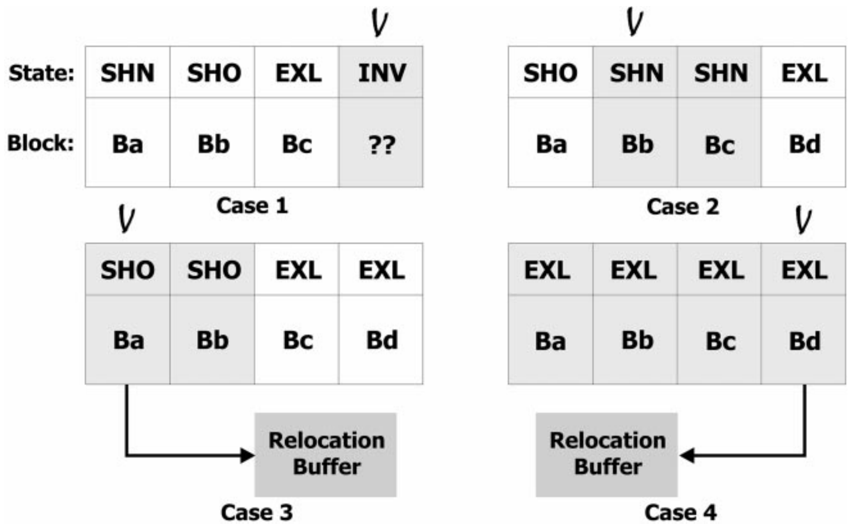

FIG. 3. Priority-based victim block selection.

relocation process. The replaced block in the relocation buffer will be relocated to a remote AM on a priority basis, as will be described in the next subsection.

4.2. Block Relocation

When a block in the SHO or EXL state is replaced, it is required to relocate the victim to a remote AM to keep the last copy of a datum in the system. A four-level priority scheme is again used in choosing which node to accommodate the replaced block. Figure 4 briefly depicts our priority scheme.

A node with a shared copy of the replaced block is given the highest priority. It is clear that this case is possible only when a block in the SHO state is replaced. The shared copy now takes ownership, and no data update in the AM is necessary. The resulting state of the block is SHO.

The second priority is given to a node with at least one block in the INV state and no shared copy of the replaced block. The data will be stored in the block, and the resulting state is EXL, regardless of the state that the original replaced block had. This is because (i) if the original state of the replaced block was SHO and there exists a shared copy in a node, this case falls into the priority 1 case, and (ii) if, however, there is no shared copy of the replaced block (which is always the case for the replaced block in the EXL state), the resulting state should be EXL. It may seem that relocation to the node with a block in the INV state is preferable to the node having a shared copy of the replaced block. However, our scheme favors a node with a shared copy because (i) relocation incurs ownership transfer only and (ii) better performance can be achieved from the efficient use of memory space [11].

A node with a block in the SHN state which is not a shared copy of the replaced block gets the next priority. As in the victim selection process, a block in SHN state can be overwritten. Last, the lowest priority is given to the node with all owner blocks in the set. In this case, a new victim is selected and needs to be relocated to

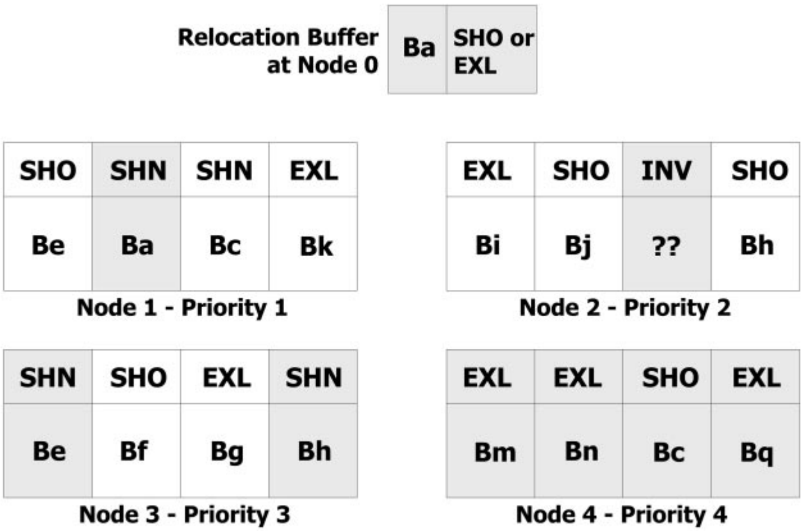

FIG. 4. Priority-based remote node selection for relocation.

a different AM again. One observation is that in the presence of unallocated space [12], it is very rare to come down to this lowest-priority case. As the worst-case scenario, however, we also need to handle the last case with care in order not to have a chain of replacement actions which can happen in previous COMA machines [11, 13]. A simple solution is given in the next subsection incorporated in our priority scheme.

Figure 5 illustrates a conceptual mechanism of the relocation. On a reference miss, the node decides whether a relocation action is necessary (**1a**). It sends a data request on the bus while fetching the victim block from the local memory (**2a**). It puts the fetched block into the relocation buffer with the state (**3a**). Upon the arrival of missed data, it begins the relocate transaction and the processor resumes its execution (**4a**).

From the viewpoint of a remote node, when a relocate transaction is seen on the bus, the node buffers the data with its address and state (**1r**). The node looks up the AM tag and state to decide its priority in accepting the block it has just received (**2r**). Based on the result of the tag and state look-up, it generates and sends to the arbiter a priority vector, which is the 2-bit priority concatenated with its node ID (**3r**). In case of a tie in the 2-bit priority, the node ID, the lower bits in the vector will help decide the winner. After arbitration, the result will be passed back to the controller, which will either update the AM and the tag with the buffered data and state or simply discard them (**4r**). The distributed arbitration determines the unique winner that will accommodate the block, and all other nodes will discard the block, thus achieving our goal. If needed (as in the priority 4 case), a node performs a similar relocation process using its relocation buffer.

4.3. *Ownership Relinquish*

Swapping [13] is beneficial in the lowest priority case to avoid a chain of replacement actions. The swapping technique forces two nodes—the node which has a reference miss and needs to replace a block and the node which supplies the data—into exchanging the blocks. With some modification to our original protocol, swapping can be easily implemented.

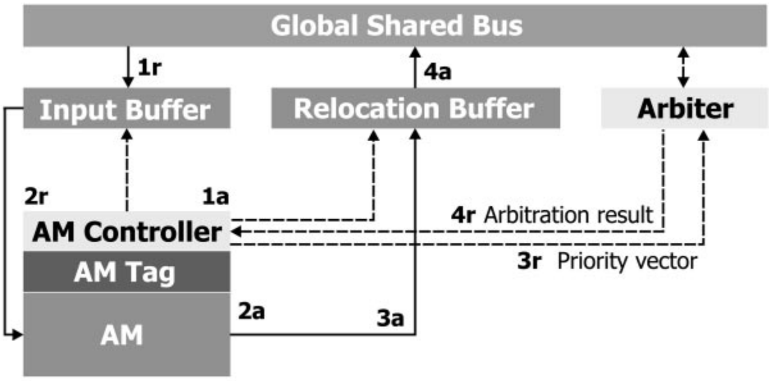

FIG. 5. Block relocation mechanism.

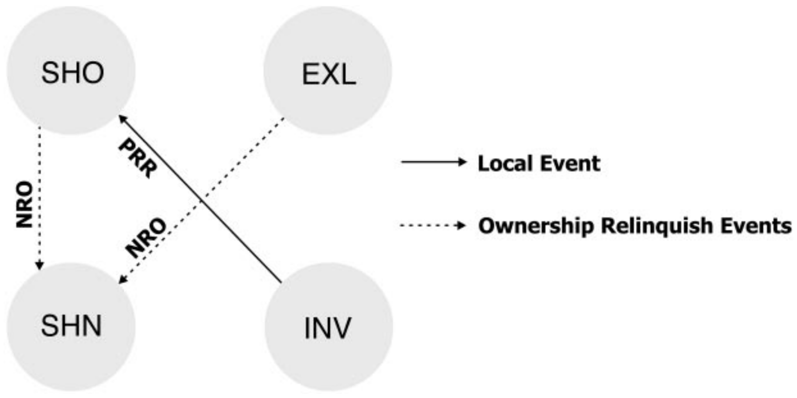

FIG. 6. Ownership relinquish.

We introduce a local event called *processor read with relocation* (PRR), which occurs when there is a read reference miss and relocation is necessary due to replacement. We introduce a new bus-induced event, *network read request with ownership* (NRO), which is a request for data with the ownership (in Fig. 6). When PRR occurs, the node will request the missing block and ask for the ownership also by issuing NRO. The owner node, upon receiving the NRO event, provides the data and relinquishes the associated ownership, leaving the block now in SHN (dotted lines). The other party, the requesting node, will set the state of the new block SHO.

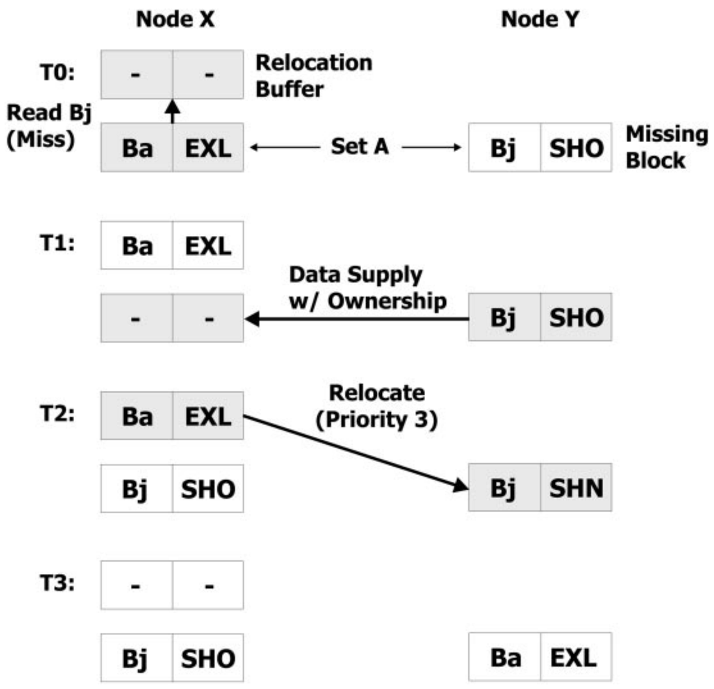

FIG. 7. Swapping.

Now the replaced block in the relocation buffer is guaranteed not to go down to the priority 4 case in the relocation process, because the node which just supplied its data to the requesting node, has an SHN block in the same set that the replaced block will be in. Thus the original priority 4 case now does not occur, avoiding a chain of relocation actions (Fig. 7). For simplicity, a direct-mapped AM is assumed in the figure, and only one set (set A) is shown. With this modification, we made swapping a natural subset of our priority-based relocation.

5. PERFORMANCE

5.1. Experimental Setup

We use a set of program-driven simulations to compare two shared-bus multiprocessors with different memory architectures: UMA (*uniform memory access*), a traditional SMP similar to SGI Challenge [8] and DICE, a COMA machine with the above coherence and replacement mechanism. Our simulator consists of two parts: the MINT front end [26] which simulates the execution of the processors and a back end that simulates the memory system and the bus. The front end calls the back end on every data reference and synchronization operation, and the back end decides which processors block waiting for memory and which continue execution. Since the decision is made on-line, the back end affects the timing of the front end, so that the control flow of the application and the interleaving of instructions across processors can depend on the behavior of the memory system and the bus.

The simulator is capable of capturing contention within the bus and in the state and tag memory due to conflicting accesses from the processor and the bus. The simulated processor is MIPS R4000 [10] with a 200-MHz clock. We assume no stalls for instruction fetching, and an instruction can be executed in a processor clock cycle (pclock) if there is no data cache miss. The L1 cache is 2 KB and the access time is hidden if an access hits in the cache. It has a 6-pclock block fill time. The 4-way set-associative L2 cache is 32 KB for UMA and 16 KB for DICE and has a cycle time of 30 ns (6 pclocks) and a 10-pclock block fill time. UMA's L2 cache is made larger than that of DICE for a fairer comparison. With this cache configuration, UMA achieves node hit rates of 96–99 %. Main memory is fully interleaved with an access time of 120 ns. The block size in the memory hierarchy is 32 bytes.

For DICE, the *memory pressure*, or the ratio of the data size to the total AM size, is adjusted to be around 50–75 %. An AM is four-way set-associative. Note that we avoided a fixed memory pressure by arbitrarily setting the memory size to a number that is not a power of two times the set-associativity, e.g., 385 KB. The relationship between the memory pressure and the performance of a COMA machine has been studied in other works [12, 13, 16]. For example, Jamil and Lee [12] have shown that a memory pressure of 50 % is needed to keep the rate of replacement per reference miss to less than 30 %.

The backplane bus simulated for both the models is similar to POWERpath-2 [8]. It supports a split-transaction protocol to decouple memory requests and responses, is clocked at 50 MHz, and can have up to eight outstanding read

TABLE 1

Summary of Benchmark Programs

Program	Description	Input
Barnes	Barnes-Hut algorithm for the many-body problem	4 K bodies for 5 time steps
Cholesky	Cholesky factorization of a sparse matrix	bcsstk14
FFT	Complex 1D version of radix $-\sqrt{N}$ six-step FFT	256 K points
LU	LU decomposition of a dense matrix	300×300 matrix, 16×16 block
Ocean	Ocean basin simulator	130×130 ocean, 10^{-7} tolerance
Radix	Radix sorting	400,000 integers
Water	Simulates evolution of a system of water molecules	512 Molecules, 3 time steps

requests waiting for responses. Each bus transaction consumes five bus cycles, and a cache block can be transferred in one bus transaction.

We use seven programs from the SPLASH–2 benchmark suite [27] to drive our simulator. Program descriptions and inputs are summarized in Table 1. The programs are all written in C using the ANL macros to express parallelism [2] and are compiled by gcc with -O2 optimization flag. For all measurements, we gathered statistics during the parallel sections only.

5.2. *Simulation Results*

5.2.1. *Bus utilization.* We compare the bus utilization (i.e., the fraction of time when the global bus is busy) of each model when 16 processors are used. Figure 8 shows the result where each bar comprises two components: coherence traffic due to invalidation transactions and all the others including read and write memory requests and memory write-backs. DICE achieved a significantly lower global bus utilization in all the programs studied. Programs with heavy bus usage like FFT and Ocean are likely to benefit from this traffic reduction.

In terms of the absolute number of bus transactions which is not shown in the figure, DICE could reduce the global bus traffic, by as much as 80% for Ocean, with an average of 64%. The reduction of traffic was consistent over all the

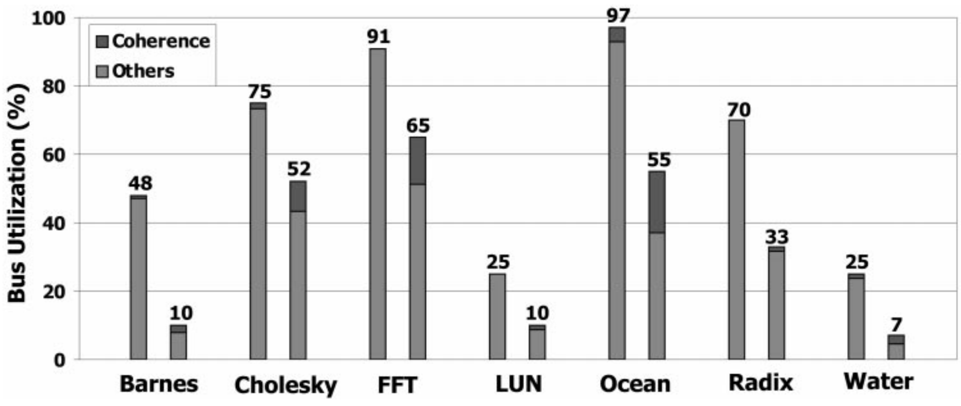

FIG. 8. Bus utilization when $P = 16$ (left bar: UMA and right bar: DICE).

TABLE 2

Block State on a Read Reference

	UMA				DICE			
Program	**M**	**E**	**S**	**I**	**EXL**	**SHO**	**SHN**	**INV**
Barnes	71.5%	2.6%	25.9%	0.0%	73.5%	4.4%	22.0%	0.1%
Cholesky	38.4%	4.4%	57.2%	0.0%	45.0%	29.3%	25.6%	0.1%
FFT	56.5%	42.4%	1.1%	0.0%	80.6%	18.9%	0.2%	0.3%
LU	52.9%	3.2%	43.9%	0.0%	55.0%	18.9%	26.0%	0.1%
Ocean	42.9%	39.4%	17.5%	0.2%	80.3%	2.5%	16.8%	0.4%
Radix	56.3%	5.5%	38.2%	0.0%	59.4%	5.4%	35.2%	0.0%
Water	51.7%	0.8%	47.5%	0.0%	52.4%	3.9%	43.6%	0.0%

benchmark programs by about 40% (FFT)–80% (Ocean). The highest reduction in Ocean is due to its high local data traffic rate per instruction [27] which is captured in the attraction memory of COMA.

As predicted in [18], an interesting phenomenon is that the invalidation traffic was not reduced in DICE; rather, more invalidation transactions were generated because the large caching space allows and exposes more sharing of memory blocks in DICE. This increases the relative portion of coherence traffic in DICE, implying that it can become a performance degrading factor; in Ocean and Water the invalidation traffic was more than 30% of all the bus traffic. To further cut down the bus traffic, techniques such as the adaptive protocol for migratory blocks [6] or self-invalidation [4] can be integrated with the DICE coherence protocol.

5.2.2. *Block state distribution.* Table 2 shows the frequency of block states on a read hit. It is observed that DICE, with its large local memory, generally allows more sharing of blocks; i.e., the percentage of the "SHO" and "SHN" states exceeds that of the "S(hared)" state of UMA. In FFT, the L2 cache of UMA fails to capture the blocks across program phases, resulting in few read hits in the "S(hared)" state,

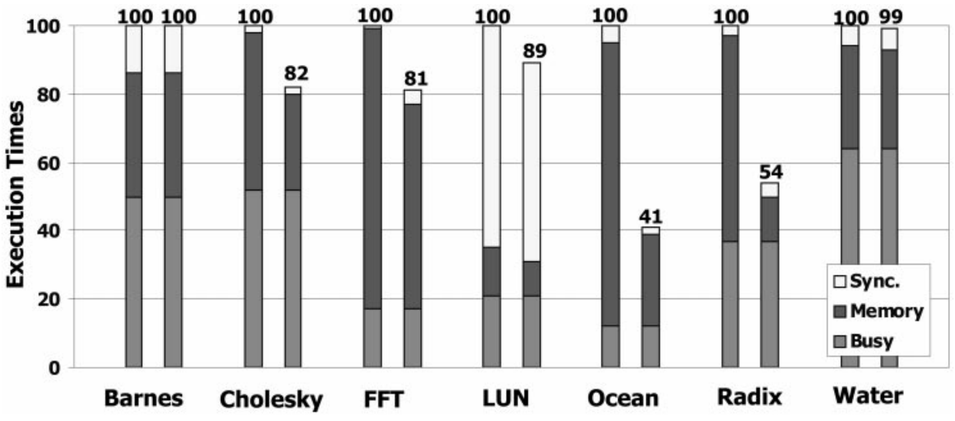

FIG. 9. Execution times when $P = 16$ (left bar: UMA and right bar: DICE).

while DICE still retains some shared blocks for later read hits. It is also noted that there are relatively more coherence misses in DICE. In fact, coherence misses constitute more than half of all the misses in Barnes, Ocean, and Water under the machine configuration studied.

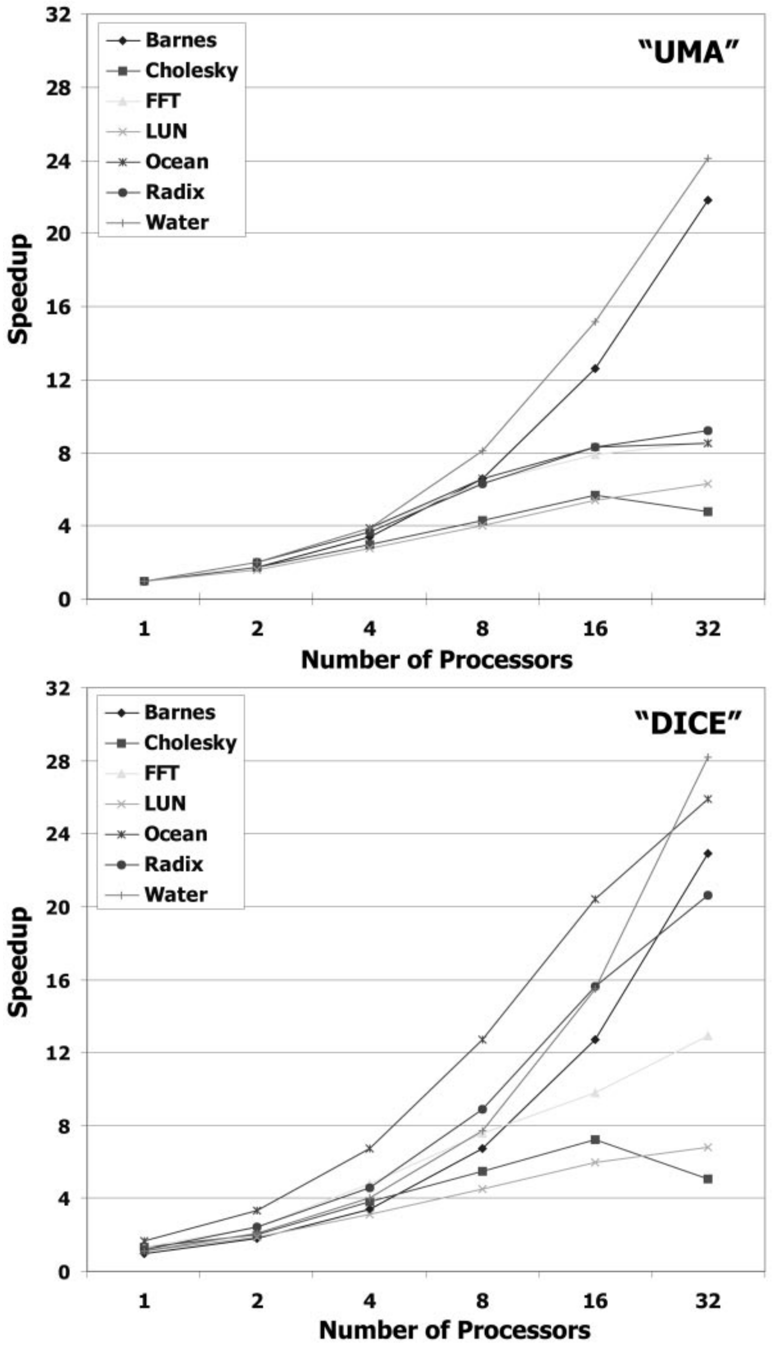

FIG. 10. Speedups over a single-processor UMA machine.

5.2.3. *Execution time.* Figure 9 shows the execution times (normalized to those of UMA) of benchmark programs when 16 processors are used. Each execution time is divided into three components: busy time, time spent for memory (both read and write), and synchronization time. For five programs out of seven, DICE had reasonably or significantly shorter execution times (by about 10% to 60%) than UMA, and the geometric mean of the execution times of all the programs on DICE was 75%. The difference in execution time mainly came from the difference in time spent on waiting for memory. High local hit ratio and less bus contention due to low bus utilization contribute to shorter execution times in DICE.

For Barnes and Water, two machine models achieved virtually the same performance, although the bus traffic for them was significantly lower in DICE. The reasons include (i) Barnes and Water have relatively low bus utilization (40% and 20%, respectively) meaning that there was little contention in the bus while other programs exhibit very high bus utilization (for example, 96% for Ocean and 70% for Radix), and (ii) both the programs have relatively high percentage of busy time. Their relatively weak dependence of performance on the global bus is indicated in Fig. 10, where they show good scalability even on UMA. Traffic generated per instruction or *communication-to-computation ratio* for both the programs is comparatively small [27].

5.2.4. *Scalability.* To measure the scalability of UMA and DICE, experiments were conducted on both models with varying number of processors. Fig. 10 shows the speedups for all the configurations over the single-processor UMA model.

Cholesky experienced speed-down in either model with 32 processors, as it requires very high bandwidth [27] which is not handled well by the global bus model simulated. Ocean and Radix showed good scalability on DICE while their performance suffered on UMA due to bus saturation after 16 processors. The results show that the advantage of the DICE architecture becomes clear as more processors are introduced.

6. CONCLUDING REMARKS

DICE is a shared-bus multiprocessor utilizing cache-only memory architecture (COMA) to effectively decrease the gap between modern high-performance microprocessors and the bus. DICE tries to optimize COMA for a shared-bus medium, in particular to reduce detrimental effects of cache coherence and the "last memory block" problem on replacement. We presented the coherence and replacement protocol for the DICE multiprocessor in this paper. With careful use of ownership some unnecessary coherence and replacement actions are avoided. Although replacement in local memory presents a unique problem to coherence, our replacement algorithm dynamically chooses an optimal location for data relocation.

As the bus performance falls further behind the microprocessor speed, shared-bus multiprocessors will no longer be able to capitalize on the advances in microprocessors. We have shown in this paper that the DICE machine can be efficiently implemented using a shared-bus medium to reduce the dependency microprocessors have on bus bandwidth. Our experimental results with seven SPLASH–2 benchmark programs show a drastic reduction of the global bus traffic up to 80% with an average of 68% for 16 processors. Execution times of DICE were also improved due to reduced bus traffic and higher local memory utilization, by as much as about 60% with an average of 25%. The reduced bus traffic helps DICE achieve a better scalability than a traditional SMP. A bus-based COMA multiprocessor like DICE can become a viable alternative for future shared-bus SMP implementation.

With the current progress of VLSI technology, a system-on-a-chip with several hundred million transistors will be a reality within a few years [3]. A node of a DICE machine with a decent AM (32 MB or 64 MB) can be fabricated on a single chip with such technology, which will make a nice building block for multiprocessors. The DICE architecture described in this paper can be implemented on a single board with such building blocks for cost-effective general-purpose multiprocessing.

ACKNOWLEGMENTS

We would like to thank the former members of the DICE project: Manu Agarwal, Sujat Jamil, Bland Quattlebaum, and Professor Larry Kinney in the Electrical and Computer Engineering Department, University of Minnesota. We also appreciate the constructive comments made by anonymous referees, which greatly helped improve the quality of this paper. The DICE project was supported by a funding from Samsung Electronics, Seoul, Korea and by a DoD/AFOSR grant under Contract F49620-96-1-0472. Sangyeun Cho was supported in part by a fellowship from the Korea Foundation for Advanced Studies.

REFERENCES

1. J.-L. Baer and W.-H. Wang, On the inclusion properties for multi-level cache hierarchies, *in* "Proc. 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, June 1988," pp. 73–80.
2. J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, "Portable Programs for Parallel Processors," Holt, Rinehart, & Winston, New York, 1987.
3. D. C. Burger and J. R. Goodman, Guest Editors' Introduction: Billion-transistor architectures, *IEEE Comput.* **30**, 9 (Sept. 1997), 46–48.
4. S. Cho and G. Lee, Reducing coherence overhead in shared-bus multiprocessors, *in* "Proc. Euro-Par '96, Lyon, France, Aug. 1996," pp. 492–497.
5. S. Cho, J. Kong, and G. Lee, On timing constraints of snooping in a bus-based COMA multiprocessor, *Microprocessors and Microsystems* **21**, 5 (Feb. 1998), 313–318.
6. A. L. Cox, and R. J. Fowler, Adaptive cache coherency for detecting migratory shared data, *in* "Proc. 20th International Symposium on Computer Architecture, San Diego, CA, May 1993," pp. 98–107.
7. R. Crisp, Direct RAMBUS Technology: The New Main Memory Standard, *IEEE Micro* (Nov./Dec. 1998), 18–28.
8. M. Galles and E. Williams, Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor, *in* "Proc. 27th International Conference on System Sciences," Vol. 1, pp. 134–143, Jan. 1994.
9. E. Hagersten, A. Landin, and S. Haridi, DDM–A Cache-Only Memory Architecture, *IEEE Computer* (Sep. 1992), 44–54.
10. J. Heinrich, "MIPS R4000 Microprocessor User's Manual," Prentice–Hall, Englewood Cliffs, NJ, 1993.

11. S. Jamil, "Block Replacement in Cache-Only Memory Architecture Multiprocessors," M.S.E.E. thesis, Department of Electrical Engineering, University of Minnesota, 1994.

12. S. Jamil and G. Lee, Unallocated Memory Space in COMA Multiprocessors, *in* "Proc. 8th International Conference on Parallel and Distributed Computing Systems," pp. 228–235, Orlando, FL, Sep. 1995.

13. T. Joe and J. L. Hennessy, Evaluating the Memory Overhead Required for COMA Architectures, *in* "Proc. 21st Annual International Symposium on Computer Architecture," pp. 82–93, Chicago, IL, Apr. 1994.

14. K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, Performance characterization of a Quad Pentium Pro SMP using OLTP workloads, *in* "Proc. 25th Annual International Symposium on Computer Architecture, Barcelona, Spain, June 1998," pp. 15–26.

15. L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. Computers* **C-28**, No. 9 (Sep. 1979), 690–691.

16. A. Landin and F. Dahlgren, Bus-Based COMA–Reducing Traffic in Shared-Bus Multiprocessors, *in* "Proc. 2nd International Symposium on High-Performance Computer Architecture," pp. 85–105, San Jose, CA, February 1996.

17. G. Lee, "Common Platform: A Case of Distributed Shared Memory Multiprocessors," DICE Project Technical Report No. 2, Department of Electrical Engineering, University of Minnesota, June 1992.

18. G. Lee, An Assessment of COMA Multiprocessors, *in* "Proc. 9th International Parallel Processing Symposium," pp. 388–392, Santa Barbara, CA, Apr. 1995.

19. G. Lee, "Block Replacement Method in Cache Only Memory Architecture Multiprocessor," U.S. Patent No. 5,692,149 (1997).

20. G. Lee and J. Kong, Prospects of Distributed Shared Memory for Reducing Global Traffic in Shared-Bus Multiprocessors, *in* "Proc. 7th IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems," pp. 63–67, Washington, DC, Oct. 1995.

21. G. Lee, B. Quattlebaum, S. Cho, and L. Kinney, Global Bus Design of a Bus-Based COMA Multiprocessor DICE, *in* "Proc. IEEE International Conference on Computer Design," pp. 231–240, Austin, TX, Oct. 1996.

22. G. Lee, B. Quattlebaum, and L. Kinney, "Protocol Mapping in Bus-Based COMA Multiprocessors," DICE Project Technical Report No. 10, Department of Electrical Engineering, University of Minnesota, Mar. 1994.

23. K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Computer Systems* **7**, 4 (Nov. 1989), 321–359.

24. T. Lovett and S. Thakkar, The Symmetry Multiprocessor System, *in* "Proc. 1988 International Conference on Parallel Processing," pp. 303–310, University Park, PA, Aug. 1988.

25. P. Stenström, T. Joe, and A. Gupta, Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures, *in* "Proc 19th Annual International Symposium on Computer Architecture," pp. 80–91, Gold Coast, Australia, May 1992.

26. J. Veenstra and R. Fowler, Mint: A Front-End for Efficient Simulation of Shared-Memory Multiprocessors, *in* "Proc. 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94)," Durham, NC, Jan. 1994.

27. S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH–2 Programs: Characterization and Methodological Considerations, *in* "Proc. 22nd International Symposium on Computer Architecture," pp. 24–36, Santa Margherita Ligure, Italy, June 1995.

28. "KSR-1 Technical Summary," Kendall Square Research, Waltham, MA, 1992.

SANGYEUN CHO is a Ph.D. candidate in computer science and engineering at the University of Minnesota in Minneapolis. His current research interests are in microprocessor architectures, compilation techniques, and their performance evaluation. Cho received a B.S. in Computer Engineering from Seoul National University, Seoul, Korea in 1994 and an M.S. in Computer Science from the University of Minnesota in 1996. He is a student member of the IEEE Computer Society and the ACM.

JINSEOK KONG earned a Ph.D. degree in computer science and engineering at the University of Minnesota in 1998. His research interest is in the area of computer architecture. He received a B.S. and an M.S. in Computer Science from Seoul National University, Seoul, Korea.

GYUNGHO LEE is an associate professor of electrical engineering in the University of Texas, San Antonio. Prior to joining the University of Texas, he was with the electrical engineering faculty at the University of Minnesota in Minneapolis from 1988 to 1996 and worked as an assistant professor at the Center for Advanced Computer Studies, University of SW Louisiana in Lafayette from 1986 to 1988. While he was on a leave of absence from the University of Minnesota from 1990 to 1992, he worked as the principal architect of SSM7000, the first commercial shared-memory multiprocessor in Korea. He was responsible for the design of coherence protocol and two-level cache memory in addition to the overall architecture of SSM7000. His research interests are in computer architectures, switch architectures for multiprocessor interconnection and ATM networks, and compiler optimizations. He was a recipient of the Outstanding Paper Award from the 1986 International Conference on Parallel Processing for his work on a combining switch. He holds a U.S. patent on bus-based cache-only memory architecture multiprocessors.