Switch MSHR: A Technique to Reduce Remote Read Memory Access Time in CC-NUMA Multiprocessors

Laxmi Narayan Bhuyan, *Fellow*, *IEEE*, and Hujun Wang

Abstract—A remote memory access poses a severe problem for the design of CC-NUMA multiprocessors because it takes an order of magnitude longer than the local memory access. The large latency arises partly due to the increased distance between the processor and remote memory over the interconnection network. In this paper, we develop a new switch architecture, called Switch MSHR (SMSHR), which provides the cache block to the requesting processors without those requests having to go to the home memory. The SMSHR idea is based on providing a few miss status holding registers (MSHRs) in each switch that keep track of read requests to the memory. The SMSHR blocks secondary requests to the same memory block and provides them with a copy of the block when the primary reply returns. The SMSHR design is then extended to include a switch cache, which can temporarily save a copy of the data block for later use. We provide basic block designs for the SMSHR and SMSHR+cache architectures in this paper. We explore the design space by modeling the new switch architectures in a detailed execution-driven simulator and analyze the performance benefits. Our simulation results show that applications with a high degree of data sharing benefit tremendously from the SMSHR and SMSHR+cache techniques.

Index Terms—CC-NUMA multiprocessor, memory latency problem, interconnection network, miss status holding register, execution-driven simulation.

1 Introduction

As the gap between processor and memory speeds increases, finding memory latency hiding techniques has become an important area of research. The problem is particularly acute for Cache-Coherent Non-Uniform Memory Access (CC-NUMA) servers where remote memory accesses take an order of magnitude longer compared to the local memory accesses [16], [18]. Most of this time is consumed in communication over the interconnection network of the machine. The design of high bandwidth and low latency networks has been addressed well in the literature. The aim of this research is to develop a technique where a remote memory access request can be satisfied at the interconnection network without the request having to go all the way to the remote memory. Such a technique will considerably reduce the current memory latency problem in CC-NUMA multiprocessors.

The idea presented in this paper is to make some of the memory read requests wait in the network if another request to the same data block has been issued before by another processor. When the block returns from the memory, it is supplied to the waiting processors. Such a technique has many advantages. First, the waiting requests may be satisfied earlier because they do not have to travel to the remote memory. Second, suppressing these subsequent requests will reduce the traffic at the network and memory system, which will make the current reads faster. The approach has some similarity to the combining approach [8] that was suggested in the early 80s, which aimed to reduce the synchronization traffic in shared memory multiprocessors. The combining technique relies on combining the requests, which arrive at a network switch at the same time as the synchronization variable. The combining technique can be extended to handle all the memory read requests. However, as we will show through execution-driven simulations, the arrival of the requests really does not happen at the same time, so the performance gain is marginal. Rather, we propose to create a hardware mechanism where we can remember the identities of the requests that have already passed so that subsequent requests to the same variable or cache line can be blocked.

We take a clue from the lockup-free cache design. The organization of a lockup-free cache with support for nonblocking loads was first given by Kroft [15]. In Kroft's implementation, registers called MSHRs (miss status holding registers) are used to hold information on outstanding misses. The MSHRs save enough information on a miss so that, when a requested cache block arrives from the next lower level in the memory system, the load instruction for the data in that block can be completed. Farkas and Jouppi [5] extend the design to allow more than one request to the same cache line. These requests wait in the MSHR until replies to the primary misses are returned. The MSHR then provides these data to the secondary requests which are waiting. This is data sharing among various independent instructions in an ILP processor. We would like to create

- *L.N. Bhuyan is with the Computer Science and Engineering Department, University of California, Riverside, CA 92521. E-mail: bhuyan@cs.ucr.edu.*
- *H. Wang is with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. E-mail: hwang@cs.tamu.edu.*

Manuscript received 24 Oct. 2001; revised 16 May 2002; accepted 29 July 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115245.

such a mechanism where cache lines are shared among various processors in a CC-NUMA machine.

Switches are basic building blocks for interconnecting CC-NUMA multiprocessors. Because remote misses from different processors flow through switches, these switches provide ideal places for data sharing among the processors. Therefore, we propose embedding MSHRs into switches in an interconnection network so that subsequent requests for the same cache block which flow through a switch do not have to access memory modules. We call the mechanism a *Switch MSHR (SMSHR)*. Our simulation results indicate a substantial gain in performance by including only a few MSHRs at the switch. A novel switch cache architecture was proposed in [10], where a small data cache is provided in a switch to store the recently read data for use by other processors in the system. In this paper, we examine the combined architecture of an SMSHR and a switch cache, called an SMSHR+cache, and measure the performance improvement.

In this paper, we present the detailed architectures of the SMSHR and SMSHR+cache for a CC-NUMA multiprocessor with a mesh interconnection network. Block level diagrams are provided to explain the implementation and hardware complexity of the new switches. Necessary changes to the cache coherence protocol are presented to maintain coherence among the shared blocks. We then explore the design space of our proposed architectures by modifying an execution-driven simulator, RSIM [24], and perform detailed simulations to evaluate the performance. The simulator is capable of simulating CC-NUMA multiprocessors with ILP processors, lock-up free cache, branch prediction, and speculative execution.

The paper makes the following contributions:

- It presents two new switch architectures (SMSHR and SMSHR+cache) to improve the memory system performance of CC-NUMA systems.
- The hardware architecture and cache coherence protocols are developed to ensure correct and fast read operations through the switches.
- To illustrate the performance of the new switches, we implement the new switches in RSIM and compare the results with those from the basic switch, combining technique, and switch cache in detail. It is shown that an improvement in the execution time of up to 38 percent for some applications is possible.
- Finally, the paper presents sensitivity studies related to the new switches and determines the parameters that have significant impact on the performance.

The rest of the paper is organized as follows: Section 2 presents the related work and Section 3 describes our basic approach. In Section 4, we present the switch architecture with MSHRs, explain its operation, and show the implementation in detail. Section 5 describes the SMSHR+cache architecture. Section 6 describes the simulation environment, presents the performance of our new switches, and compares the results of our new switches with other existing techniques. It also presents a sensitivity study to quantify the effect of switch design parameters on the system performance. Finally, Section 7 presents the conclusions of this study.

2 Related Work

The communication overhead due to remote memory accesses is a significant performance bottleneck for a CC-NUMA multiprocessor. Several scientific applications exhibit a wide sharing pattern [12], where many processors generate read requests to the same memory block within a small interval of time. This causes network and memory congestion and significantly degrades the application execution time. Some techniques, such as replication and dynamic page migration, which are used in current machines like SGI Origin [16] and HAL S1 [28], have been proposed to reduce remote memory accesses by placing the required memory pages in the local memory. Also, techniques like prefetching and multithreading have been proposed to hide the memory latency. However, in this paper, we concentrate on CC-NUMA interconnection networks and caching techniques.

2.1 CC-NUMA Interconnection Networks

CC-NUMA multiprocessors provide a unified view of the memory for easy programming and are capable of providing significant performance benefits both for scientific and commercial applications. Examples include the Stanford DASH [17], Sequent STiNG [18], HP Exemplar [1], SGI Origin [16], and HAL S1 [28] machines. Most of these systems implement a network cache at each cluster, which is described in the next subsection. However, none of the networks has caching or combining inside the network.

Stanford DASH: The Stanford DASH [17] uses a pair of wormhole routed meshes to implement the interconnection network, one mesh for requests and the other for replies. The interconnect for the system can utilize point-to-point connections which do not have the physical length limitations of multidrop buses. The system employs a four-processor SMP cluster as its node. The advantage of an SMP-based node is the potential for cache-to-cache sharing within the node. On the other hand, the disadvantage of an SMP-based node is to cause the local memory latency to be longer due to the snoopy bus.

Sequent StiNG: The Sequent STiNG [18] uses multiple four-processor SMP nodes, each with a high bandwidth bus with the low memory access latency. Scalability to system sizes with more than four processors uses a second-level interconnect based on the Scalable Coherent Interface (SCI) specification. This standard point-to-point interconnect breaks the length restriction of the single shared bus since signals no longer need to propagate within a single clock cycle. The system is architecturally similar to the Stanford DASH, albeit with a different processor, a simpler network topology, and a different coherence protocol.

HP Exemplar: The HP Exemplar [1] implements a crossbar connected set of hypernodes that are then connected by parallel ring interconnects. The implementation is a modified version of the SCI protocol. The use of a crossbar intraconnect for the hypernode reduces the bandwidth penalty of using an SMP node compared with the Sequent STiNG.

SGI Origin: The SGI Origin 2000 [16] employs SPIDER routers to create a bristled fat hypercube interconnection topology. The network topology is bristled in that two

nodes are connected to a single router instead of one. The main features of the SPIDER are the low latency wormhole routing, DAMQ buffer structures with global arbitration to maximize utilization under load, and congestion control allowing messages to adaptively switch between two virtual channels.

HAL S1: The HAL S1 [28] consists of Intel Pentium Pro processor-based SMP nodes linked together with the Mercury interconnect. The interconnect implementation supports up to 32 nodes (128 processors), of which any 4-node subset can make up 16-processor cache-coherent systems. The attribute of the interconnect architecture is to support both shared-memory and message-passing, as well as hybrid systems. It also has enough reliability, availability, and serviceability features to build commercial servers.

2.2 Caching Techniques

Suggested caching techniques for CC-NUMA multiprocessors can be classified into several categories, namely, network cache, switch cache, page cache, and directory cache. These schemes are briefly described below.

Network Cache: Network caches have been suggested to provide an additional layer of shared cache for multiple processors within a single cluster in the system. This approach is used in current machines like the HP Exemplar [1], Sequent's StiNG [18], and Sun's S3.mp [23]. Nayfeh and Olukotun [22] study the performance of a cluster-based multiprocessor architecture for various processor-cache configurations in which processors within a cluster are tightly coupled via a shared cluster cache. Bennett et al. [2] investigate potential performance benefits of adding a shared cache to the network interface and provide an analysis of how processor cache misses are satisfied for various clustered multiprocessor configurations. The network cache stores the data in the working set that is allocated in remote memory modules and, therefore, minimizes the cost of cache conflict misses. However, since the network cache is implemented with a large DRAM, the slow data access increases the latency for network cache hits. By relaxing inclusion for the clean blocks [6], the utilization of the network cache is improved and the size of the network cache can be reduced. Moga and Dubois [20] explore the use of small SRAM network caches as a means to reduce the remote stalls and capacity traffic of multiprocessor clusters.

Switch Cache: The first problem with the network cache is that it is not helpful for multiprocessors with no cluster or for clusters with a smaller number of processors. The second problem is that the network cache cannot benefit from the data locality between clusters. Iyer and Bhuyan [10] embed a small fast SRAM, called a switch cache, within each switch in the network. The cache in a switch of the interconnection medium captures and stores shared data as they flow from the memory module to the requesting processor. This stored data acts as a cached block for subsequent requests, thus reducing the need for remote memory accesses. These requests can come from the same cluster or different clusters. A necessary cache coherence protocol is designed so that the inclusion property is not required. The technique provides very good performance for applications that have large data sharing. However, it cannot serve the requests to the cache blocks that arrive at the switch cache before the reply to the previous request to the same block returns. The proposed SMSHR technique is meant to take care of such requests.

Page cache: A page cache, called a *Stache* [25], employs part of each processor's local DRAM memory as a large, fully associative "level three cache." The *Stache* also provides support to allow the explicit page migration. In [26], a page cache holds remote data aliased under local addresses. The page cache can be stored in the main memory, just like the local data. The architecture features the automatic data migration and replication capabilities of cache-only memory architecture (COMA) machines. Page caches for remote data are an attractive way to reduce the number of remote misses for they redirect some references to the local main memory. However, for applications withe large and sparse remote working sets or with low spatial locality, the utilization of page caches is poor and the overhead of page relocation is high.

Directory Cache: The performance of communication intensive applications such as FFT on CC-NUMA multiprocessors is hindered by the large communication latency for cache-to-cache transfers. The data caching techniques, presented above, do not help. Cache-to-cache transfers require directory lookups at the home and multiple message transfers over the interconnection network. The multistage interconnection network with a directory (MIND) scheme [21] embeds full-map directories within the switches of the MIN, and presents a hierarchical directory protocol to maintain the cache coherence. The scheme suffers from restrictions imposed by the inclusion property requiring larger directories toward the root of the hierarchy. Kaxiras et al. [12], [13] present two extensions to the SCI protocol to provide scalable reads and writes. The characteristic of the GLOW extensions is that they create sharing trees very well mapped on the top of the network topology of the system, thus exploiting the *geographical* locality. However, widely shared data are statically identified by the user (the programmer or potentially the compiler). Such a static method of identifying widely shared data is a major drawback. Michael and Nanda [19] present the design issues for SRAM directory caches and show some performance benefits. Iyer and Bhuyan [11] embed a small SRAM directory cache at each crossbar switch. These switch directories capture the ownership information as a data block passes through the network and reroute subsequent requests directly to the owner cache, thus avoiding the expensive DRAM directory lookups.

3 The Basic Idea

Consider the timing diagram, as shown in Fig. 1. Assume that there are two requests, A and B, for the same memory block that arrive at switch i at times t_0 and t_1, respectively. Request A leaves the switch at time t_2 and its reply comes back to the switch at time t_3. These two requests can be combined [8] if request B arrives at the switch before request A leaves that switch, that is, $t_0 < t_1 < t_2$. Here, $\mid t_2 - t_0 \mid$ is called a switch latency time and it is only about four cycles in commercial CC-NUMA switches [7]. The combining technique will not work if $t_1 > t_2$. On the other hand, a switch cache captures

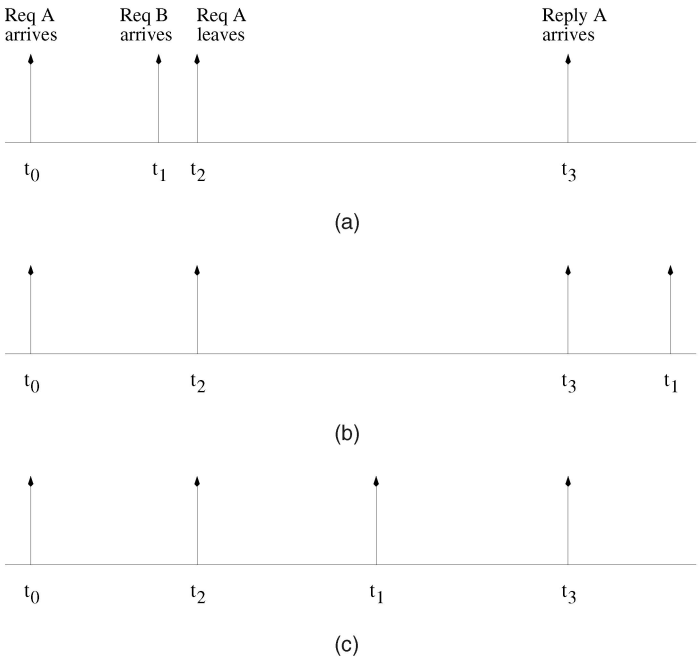


Fig. 1. Timing diagrams for three techniques: (a) Combining works ($t_0 < t_1 < t_2$), (b) switch cache works ($t_1 > t_3$), (c) switch MSHR works ($t_1 < t_3$).

and stores shared data as they flow from the memory module to the requesting processor [10]. This stored data acts as a cache for subsequent requests, thus reducing the need for remote memory accesses. If the interarrival time between requests A and B is large enough such that $t_1 > t_3$, it is possible for request B to find the data in the switch cache. From Fig. 1a and Fig. 1b, request B still has to access a memory module if it arrives at the switch after request A leaves and before the reply for request A comes back to the switch. Here, $(t_3 - t_2)$ is the memory latency time from the switch to the memory and back to the switch. Usually, this time is much higher than either the switch latency or switch cache residence times. The proposed SMSHR will work when request B arrives between t_2 and t_3. The timing diagrams for the case of the combining, switch cache, and the proposed SMSHR are depicted in Fig. 1.

Our basic idea is to make request B wait at the switch instead of passing it to its destination until the reply for request A returns. Switch i records request A before it leaves the switch. When another request B passes through the switch, it checks the record to see whether there is already a request pending for the same cache block. If so, request B waits in the switch until the reply for request A comes back to the switch. Then, the switch supplies the data block to request B. We can relate the scenario to multiple outstanding requests to the memory from a superscalar processor. The Miss Status Holding Register (MSHR) is a mechanism that is proposed to design lock-up free caches [15]. The MSHR consists of registers that keep a record of outstanding cache miss requests and corresponding information. The MSHR can also record several instructions that cause a data miss to the same block. Whenever the data is fetched, the MSHR is checked and the data is supplied to the instructions waiting for the data. By considering a processor in a CC-NUMA machine as equivalent to an instruction in an ILP processor, we propose to design an MSHR inside the interconnect switch to keep track of the outstanding memory requests. A request to a new memory block is recorded separately in a register, whereas subsequent requests to the same memory block are recorded inside an already existing register. The operation is very similar to the MSHR, hence we call the technique a *Switch MSHR* or an *SMSHR*.

3.1 Data-Sharing Tree Structure

The SMSHR technique is applicable to any interconnection network (IN). In this paper, we choose a mesh network to demonstrate our idea. However, other networks, like hypercubes and switch-based networks, are even more suitable because of their inherent tree properties. Fig. 2a shows a typical mesh architecture, where processors are connected to the nodes of a two-dimensional grid. The neighboring nodes are connected by point-to-point links. The mesh IN is a special case of *k*-ary *n*-cube networks in which the number of dimensions, n, is two. Many existing multiprocessor systems including the Intel Paragon, Stanford Dash [17], and SGI Origin 2000 [16] use such low-dimensional direct networks to interconnect the processors (IBM Blue Gene supercomputer [9] uses a three-dimensional torus). The XY routing is the most commonly used routing scheme in existing mesh systems. It is a nonadaptive routing scheme where the path between every source-destination pair is fixed. However, it is possible for a reply to traverse through a different path from its corresponding request. For example, the path for a request from node 9 to node 4 is $9 \rightarrow 8 \rightarrow 4$ using the XY routing scheme. The path for its reply from node 4 to node 9 is $4 \rightarrow 5 \rightarrow 9$ using the XY routing scheme.

For the realization of the SMSHR architecture, it is essential that the reply packet (consisting of a data block) return on the same path so that the waiting secondary requests to the same block are satisfied. Thus, we modify the traditional XY routing

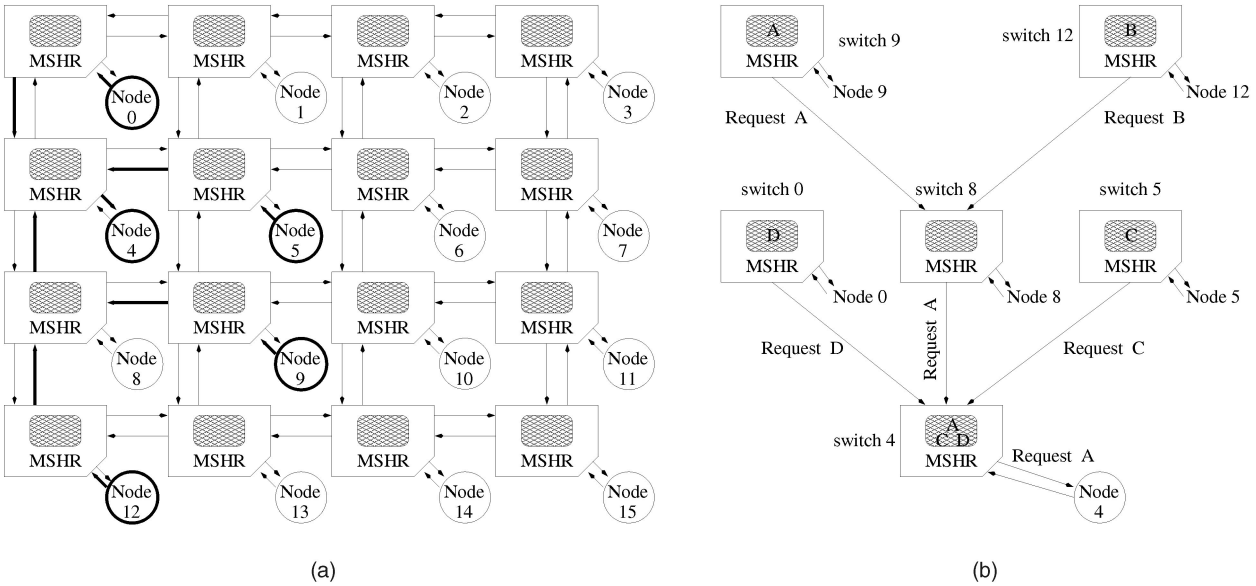


Fig. 2. MESH architecture and data-sharing tree structure. (a) 16-node mesh structure. (b) An example for a shared structure.

into: 1) the XY routing scheme for messages (such as read requests) from processors; 2) the YX routing scheme for messages (such as read replies) from memory modules. We call this routing algorithm an XY-YX routing algorithm. We have provided virtual channels in the network to make it deadlock-free [4]. The detailed proof that the proposed XY-YX routing is deadlock-free is contained in [27]. To better understand the data-sharing in mesh INs, we give below an example of the shared tree structure.

Assume that the requests A, B, C, and D from processors 9, 12, 5, and 0 read a block at memory module 4. A data-sharing tree structure is shown in Fig. 2b for this situation. Request A arrives at switch 8 earlier than request B and it arrives at switch 4 earlier than requests C and D. When request A arrives at switch 8, the switch keeps a record of the request and passes the request to its destination. When request B arrives at switch 8, it waits at the switch till the reply for request A returns. When request A arrives at switch 4, the switch keeps a record of the request and passes it on. When requests C and D arrive at switch 4, they wait at the switch till the reply for request A returns. The requested block is supplied to requests C and D after the reply for request A returns to switch 4 and to B after it returns to switch 8. It may be noted that the data sharing of the same block forms a tree with the home node at the root of the tree and the requesting processors at the leaves.

4 Switch MSHR (SMSHR) Architecture

In this section, we describe the SMSHR organization and operation in detail followed by its implementation.

4.1 SMSHR Organization and Operation

Fig. 3 shows the basic organization of a SMSHR for a 16-node CC-NUMA multiprocessor. It consists of a number of registers to hold the *primary* memory requests. The first remote miss to a block passing through a SMSHR is called the *primary* request. The subsequent remote misses to the same block through that SMSHR are called the *secondary* requests. If an SMSHR does not have enough registers to record a primary request, the situation is called a *structure miss*. The MSHR registers have sufficient bits to store the information about secondary requests. Each register consists of: 1) one bit for valid or invalid, indicating whether or not the respective MSHR entry is in use; 2) 32 bits for the block address; and 3) 16 source node bits corresponding to 16 source processors. Assuming a 64-bit virtual address architecture machine with a 40-bit physical address and a 32-byte block size, only 32 bits need to be stored as the block request address. Each MSHR entry has its own comparator so that a collection of MSHR entries can be searched associatively when a request arrives to find out whether it is a primary or secondary request. For a secondary request, only the corresponding source node bit is set. When a requested block returns from the memory, replies for the secondary requests are generated.

Read Request: Fig. 4 shows the flow chart for handling the messages which access an SMSHR. Each read request which enters an interconnection network checks the SMSHRs along its path. For a primary request, the SMSHR module selects a free MSHR entry, records the memory block address, and marks the requesting processor bit. The original message continues to the destination memory. If the SMSHR module doesn't have any free register, we drop the request from the SMSHR and it proceeds to the memory without any record. For a secondary request, the corresponding MSHR entry records the requesting processor address, marks the header as an SMSHR hit, and the request continues to the destination memory. It is necessary for this request to continue to the memory in order to maintain the full-map directory protocol. However, a memory access for this request is not needed and no reply is generated at the memory. An already marked secondary request does not access subsequent SMSHRs in its future path to the memory. This situation is illustrated in Fig. 2b, where the request B is not marked in the SMSHR at switch 4, even though it travels through node 4.

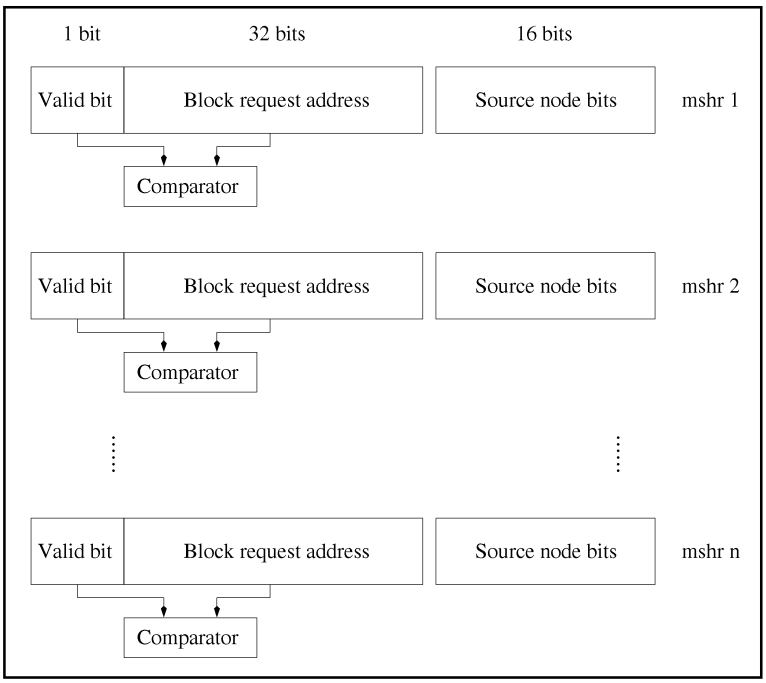


Fig. 3. Basic MSHR structure.

Cache Protocol Design: The cache operation is based on the *full-map* directory protocol. At the destination memory, a primary request can find the memory block in three states: *uncached*, *shared*, or *dirty*. For uncached or shared states, the request updates the full-map directory protocol and a reply is sent back to the requester along the requesting path. If the directory state is *dirty*, then a cache-to-cache transfer occurs. In this situation, it is possible for the owner of the block to send a reply to the source node through a different path while the copies of the primary read request are still waiting in the SMSHRs along the original path. Moreover, subsequent secondary requests for the same memory block may also be waiting for the reply in the SMSHRs. In order to avoid the *starvation* problem of those secondary requests, the home node of the block sends an ACK message to the requester when a cache-to-cache transfer is initiated. The ACK message checks the SMSHRs at switches along the original path and invalidates all copies of its primary request. If there are secondary requests in the same MSHR entry, the SMSHR module has to create a request for one of them and pass it to its destination memory as if it were a primary request. Due to false sharing or for an application that allows data race conditions to exist, it is possible that a write has been initiated to the same cache line by a different processor and, consequently, the directory sends an invalidation only to the processor which sent the primary request. The processor which sent the secondary request will get the data but not the invalidation. If this occurs, the secondary request observes a *transient* state at the directory. In such an event, the directory sends invalidation to the processor which sent the secondary request and waits for an additional acknowledgment before committing the write. Fig. 5 presents a change in the directory protocol.

Read reply: Each read reply which enters an interconnection network checks the SMSHRs along its path. If it does not hit in an SMSHR module, it is just passed through the switch. If it hits an MSHR entry and the MSHR entry only holds the *primary* request, this entire MSHR entry is invalidated. Otherwise, the SMSHR module generates replies for all *secondary* requests and puts them in a *reply* queue for transmission of the data block to the secondary source processors. The MSHR entry is then invalidated. Once the data block reaches the source node, it is handled by the local MSHR in the processor's cache.

Message Format: Each message contains a header micro-packet, as shown in Fig. 6. Each header micro-packet reserves 32 bits for the block address and 8 bits to the word address. A 6-bit source identifier specifies one of 64 possible source nodes. A 6-bit destination identifier specifies one of 64 possible system destinations. These identifiers map into the routing tables. There are five bits which specify the number of flits in the message. The *age* field, which is 4 bits, specifies 16 levels of message ages which may be used in arbitration. The header *Swmshr Hit* field provides 1 bit of SMSHR hit or miss information for use by the subsequent SMSHRs along the request path. Finally, two bits are encoded to denote the type of requests, such as read, write, ACK, and others.

4.2 Implementation of an SMSHR

The size of a crossbar switch for the mesh architecture, as shown in Fig. 2, is 5×5. Each switch has four connections to its neighbor and one to the local processor. The basic switch design considered here is similar to the SGI Spider [7]. In addition, an SMSHR consists of a flit processing unit, a request generator unit, a reply generator unit, and an MSHR unit, as shown in Fig. 7. The outputs of the request generator and reply generator units are fed to the crossbar for transmission of these messages, as explained below. Hence, the size of the crossbar has to be extended to 7×5.

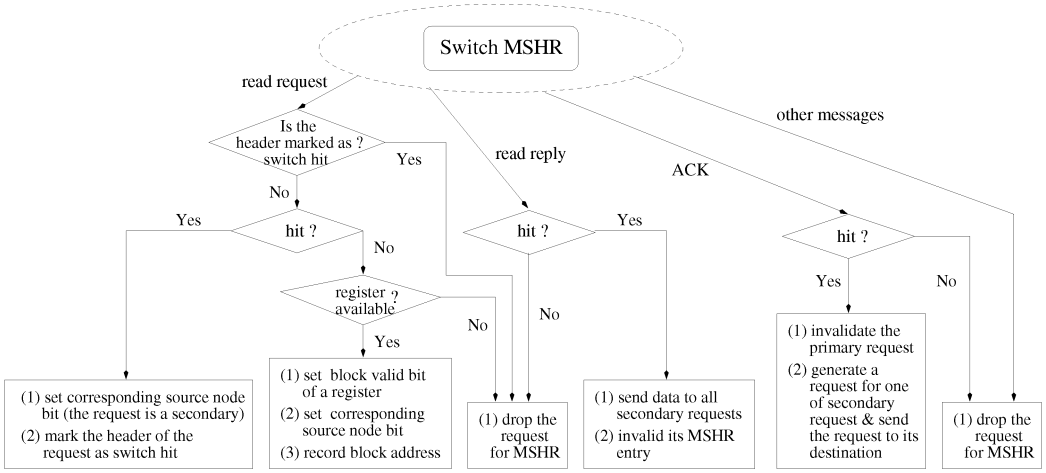


Fig. 4. Flow chart for messages accessing an SMSHR.

SMSHR Access: The SMSHR operates at the same frequency as the internal switch core. Whenever a request arrives at an input block of the switch, it is propagated to the SMSHR module. As shown in Fig. 7, there are five ports to allow five independent requests to access the SMSHR unit. The flit processing unit masks out write and coherence flits and passes Read, ACK, and Reply messages to the MSHRs. If the *Swmshr Hit* bit of a read request is marked, it is also blocked. This is done by reading *Reqtype* (2 bits) from the header vector and the *Swmshr Hit* bit. The read request header flits are copied to the snoop registers. Due to the possibility of a maximum five read requests entering the SMSHR unit in a single arbitration cycle, we need five snoop registers which hold the requested block address. Although this would rarely happen, we have to make a provision for handling such a situation. The ACK header and the read reply flits requiring MSHR processing are passed to the separate ACK and Read Reply buffers, as shown in Fig. 7.

For read requests, the SMSHR must determine a hit or miss, and mark the *Swmshr Hit* bit before the flit is transmitted to the next switch. Assume that the switch core delay is four cycles, similar to the SGI Spider [7]. We can complete the snoop operation in the MSHRs within one cycle by providing the parallel snooping (comparison) of the tag bits. The hardware design is really not complex because the number of MSHR entries is limited to only four or eight. Therefore, the SMSHR latency can be broken down as: one cycle for the buffer synchronization, one cycle for moving flits from the input buffer to the SMSHR, one cycle for the MSHR access, and one cycle for the bit update in the header transmitted at the output link. For other requests, they are chosen from the ACK buffer and read reply buffer in Fig. 7. It is not necessary for the requests to update the hit/miss information in their headers, so the snooping can be done when there are less than five read messages. Finite size queues exist at the input of the SMSHR (ACK buffer and Read reply buffer) and at the reply unit and request unit queues. When any limited size buffer gets full, we block the requests until a free space is available in the buffer. To implement the blocking of flits at the input, the SMSHR needs to inform the arbiter of the status of all its queues (in the form of space available) at the end of each cycle.

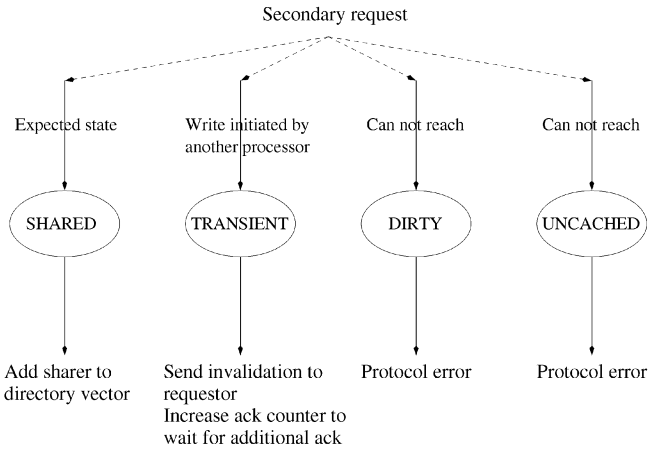


Fig. 5. Change in directory protocol.

Request generation unit: When a cache-to-cache transfer occurs, the owner of a block sends an ACK to the source processor. While an ACK request arrives at switches along its path, it needs to be serviced by invalidating the *primary* request information and checking whether there are *secondary* requests in the MSHR entry. If so, the SMSHR module has to generate a request for one of the secondary requests and pass it to its destination memory. The ACK header contains all required information for the generated request except the *source* node information. The source node information can be read from the MSHR entry. Thus, the request generator unit gathers some information from the ACK header at the beginning of the MSHR operation. The read request is generated and sent to the SMSHR *request* output block as soon as the source node information has been obtained from the MSHR entry. The generated request from the SMSHR which enters a switch is similar to that of other messages.

Reply generation unit: While a read reply arrives at a switch along its path, it needs to be serviced by invalidating an MSHR entry and checking for the *secondary* requests in that entry. The SMSHR module has to send a reply for each

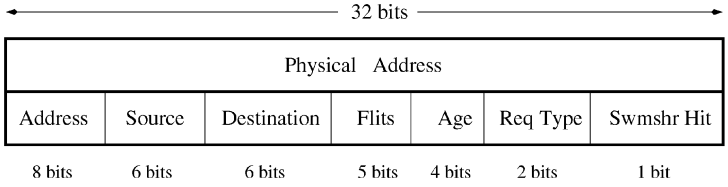


Fig. 6. Message format.

secondary request, if present. The original read reply and the MSHR entry contain all required information which can be used to generate a reply for each secondary request. The reply generator unit will generate a new reply for each secondary request and send it to the SMSHR *reply* output block. The generated reply from the SMSHR enters the crossbar and contends for the output along with other messages.

5 Implementation of an SMSHR+Cache

In the last section, we proposed the SMSHR architecture. This architecture will provide a benefit if a secondary request arrives at a switch before the reply for its primary request returns to the switch. If a secondary request arrives at a switch after the reply of its primary request returns to the switch, the secondary request still has to access memory modules and the SMSHR cannot show any benefit. The situation is depicted in Fig. 1c.

Iyer and Bhuyan [10] proposed a hardware caching technique, called a switch cache, to improve the remote memory access performance of CC-NUMA multiprocessors. By incorporating a small cache within each switching element, the shared data are captured as they pass from the memory to the processor. They organize the caching technique in a hierarchical fashion by utilizing the inherent tree structure of the multistage interconnection network (MIN). The stored data are provided for subsequent requests, thus reducing the need for remote memory accesses tremendously.

5.1 Switch Operation

Here, we will combine an SMSHR and a switch cache into a new switch architecture, called an SMSHR+cache, as shown in Fig. 8. All operations in the SMSHR+cache are very similar to those in the switch cache [10] or SMSHR architectures. Flits requiring the MSHR and cache processing are passed to a request queue in every cycle. The queue is organized as three buffers: an ACK buffer, a read reply buffer, and an invalidation buffer. The ACK buffer holds the header flits of ACK requests. The read reply buffer stores all read reply flits. The invalidation buffer holds all header flits of coherence messages. Five snoop registers hold the header flits of read requests.

The read requests and replies access the cache and MSHRs in parallel so that each access operation in the SMSHR+cache module can be finished in one cycle. Read requests are required to snoop both the MSHRs and cache to determine a hit or miss before their outgoing flits are transmitted to the next switch. For invalidation messages, it is not necessary to access the MSHR entries. The priority of an invalidation message which is chosen from the invalidation buffer is higher than that of an ACK chosen from the ACK buffer so that the data consistency is maintained. Both invalidation and ACK messages are processed in a maximum of four cycles in the absence of contention. It is

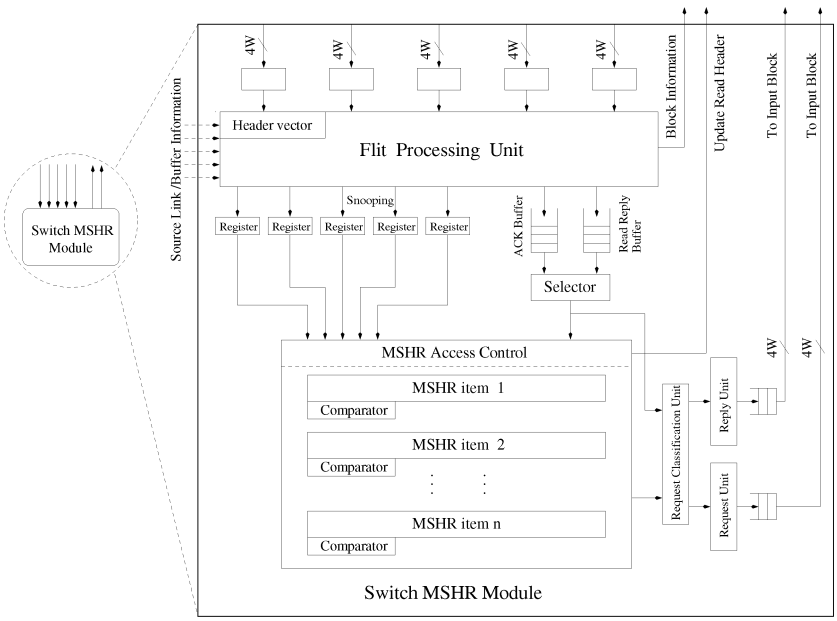


Fig. 7. Implementation of an SMSHR.

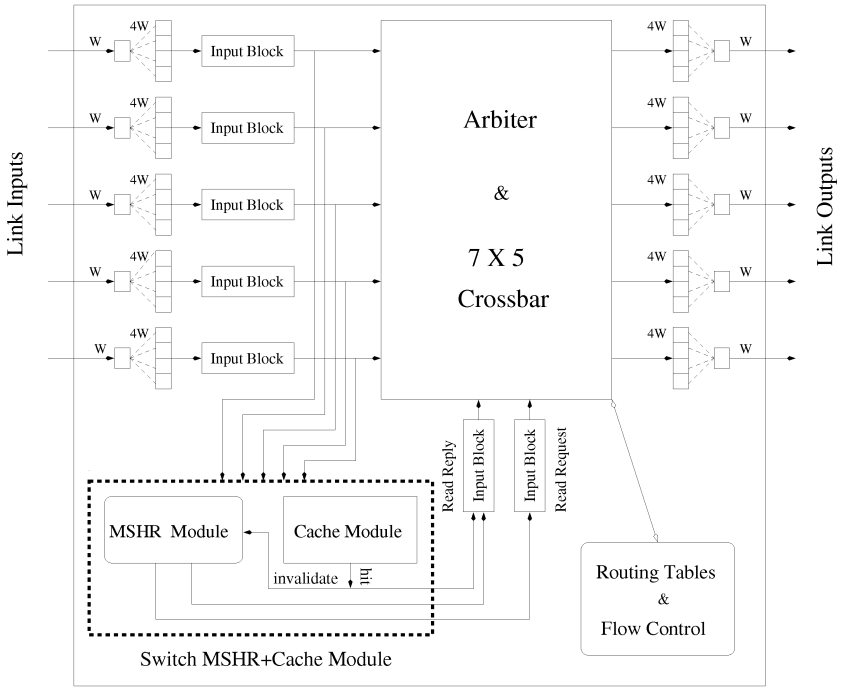


Fig. 8. Basic SMSHR+Cache organization.

not necessary for these messages to update the hit/miss information of their headers after they are completed.

Those requests which are unmarked by a previous switch are propagated to the SMSHR+cache module. Fig. 9 shows the flow chart of the switch operation for a read request. Whenever a read request arrives at a switch, the switch module checks in parallel to see whether its cache holds data and whether its MSHRs hold an entry for the same block. The information on the same block may be present either in the MSHRs or in the cache, but not in both at the same time. If the MSHRs do not contain an entry, it treats the read request as a primary request and allocates one of its registers to store the information. In the event of a switch cache hit for the same block, the cache is responsible for providing the data and sending the reply to the requester. The cache marks the header and signals to the SMSHR module that a cache hit occured. Once the SMSHR module receives this hit signal, it invalidates the MSHR entry that was just created. In order to maintain the full-map directory protocol, the original read request is marked as a switch hit and it continues to the destination memory. At the destination memory, this request only updates the directory if the directory state is SHARED. In the event of a switch cache miss, the SMSHR module handles the request.

In the event of an SMSHR hit, the request is a secondary request and the corresponding source node bit is set. The read request is marked as a hit and it continues to the destination memory to update the directory. In the event of an SMSHR miss, the SMSHR module identifies the request as a primary request or structure miss. If the request is primary, the SMSHR module sets the block valid bit and records the source node and block addresses. The entry may be invalidated in the next cycle if there was a switch cache hit. If the request is a structure miss and a cache miss occurs, the switch passes the request to the destination directly.

Despite the high performance potential of our proposed scheme, the key stumbling block which may prevent a switch cache from being used in practice is correctness. When a switch cache is used, we must make sure that the copies of a data block present in various switches are coherent. To ensure coherence, we adopt the same technique, as suggested [10]. First, the home directory is updated, even if there is a switch cache hit, by passing the read request to the home memory module. When a store operation finds the block in shared state, an invalidation signal is sent to all the processors along the same path as the requests, without flooding the network. The invalidation signal invalidates the copies in the switch caches on the way. The transient states in the directory are handled following either the technique for the SMSHR, described in Section 4, or the technique for the switch cache.

5.2 Message Access Type

Due to the embedding of a cache into the SMSHR, we must make sure that the copies of a data block present in various switches are coherent. This gives rise to identifying the coherence message in addition to other messages in the SMSHR architecture. Hence, we use three bits to encode the SMSHR+cache access type as follows:

- 000(read requests): For this case, we pass read requests to the SMSHR+cache module. They access the cache and MSHRs in paralle, and are handled as described before.
- 001(read reply): A read reply is passed to the SMSHR+cache module when it reaches a switch along its path. It also accesses the cache and MSHRs in parallel. For the cache, if the line is not present in a switch cache, the data is written into the cache. The state of the cache line is marked as *shared*. For the

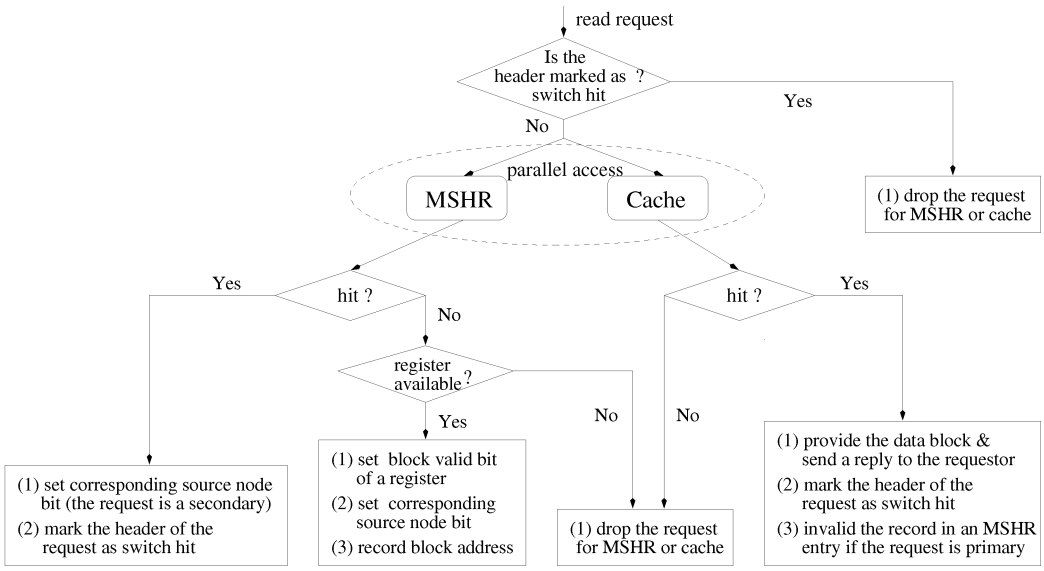


Fig. 9. Flow chart for a read request accessing an SMSHR+Cache.

MSHRs, the operations are similar to those described in Section 4.

- 010(acknowledgment messages): It is not necessary for an ACK to access the cache. The operations in the MSHRs are similar to those described in Section 4.
- 011(coherence messages): For this case, coherence messages check the cache and invalidate the cache line if present in the cache so that switch cache coherence is maintained.
- 100(other messages): These are write, write back, and other messages. We drop these messages at the SMSHR+cache and pass them to their destinations directly.

6 Performance Evaluation

To evaluate our proposed techniques, we use detailed simulations. In this section, we describe the architecture, applications, simulation results, and sensitivity studies.

6.1 Architecture and Applications

Our simulator is based on RSIM [24]. To evaluate our proposed techniques, we modified the simulator to incorporate the combining, switch cache, SMSHR, and SMSHR+cache into each switch of the mesh interconnection network. The system parameters used in the simulation are listed in Table 1. The system architecture under consideration is modeled as the processor, two-level cache, memory, network interface, and the interconnection network. This is a Non-Uniform Memory Access (NUMA) architecture where the memories are distributed among the processors, but all the processors share the same address space. The memory access time varies depending on the location of the memory word/block, hence the name NUMA. The interface provides services such as dividing a message into flits, initializing the header flit of every packet with the necessary routing information, etc. Each node contains a coherence controller, a memory controller, a directory controller, and a reply controller.

We simulated a 16-node system using a mesh interconnection network with 512 Kbytes of memory per node, which is enough to hold the shared data of applications. We chose a 16-node system to limit the simulation time since a larger system would need an appropriately larger data structure to obtain realistic results. The primary cache size is 16 Kbytes per processor with a line size of 32 bytes. It has a 2-way set associative organization and an access time of one cycle. There is a secondary cache of size 128 Kbytes per processor with a line size of 32 bytes, a 4-way set associative organization, and an access time of eight cycles. A flit size of 16 bits is considered, which is the same as the width of the link. The switch latency or delay is considered to be four processor cycles, similar to the SGI Spider [7] and Cavallino [3]. Messages are of two different lengths with 8 and 40 bytes for control and data messages, respectively.

We describe here the computational structure of six benchmark applications used in the paper. This information will be useful in later subsections for understanding the performance results. The selected applications are representative of algorithms used in an engineering computing environment.

1. The matrix multiplication (MATMUL) algorithm is done between two 128×128 double precision matrices. The principal data structures are four shared two-dimensional arrays of real numbers: two input matrices, one transpose matrix, and one output matrix. The shared data size is about 512 Kbytes. The problem is partitioned into square blocks of the output matrix and all the elements in a block are computed by one processor.
2. For the Floyd-Warshall's algorithm (FWA), we use a graph of 256 nodes with random weights assigned to the edges. The principal data structures are two shared two-dimensional arrays of integers: one distance matrix and one predecessor matrix. The distance matrix is initialized with the weights of the edges in the graph. The problem is partitioned by

TABLE 1
Simulation Parameters

Processor		Memory	
speed time	200MHz	access time	40 cycles
issue	4-way	interleaving	4
L1 cache and L2 cache		Interconnection network	
L1cache size	16 Kbytes	switch size	5×5 or 7×5
line size	32 bytes	core delay	4 cycles
set size	2	Core Frequency	200MHz
access time	1 cycle	link width	16 bits
L2 cache size	128 Kbytes	transfer frequency	200MHz
line size	32 bytes	flit size	2 bytes
set size	4	buffer size	4 flits
access time	8 cycles	virtual channel	1
SMSHR+cache		SMSHR	
size	32 bytes - 8 Kbytes	size	6 bytes - 1536 bytes
line size	32 bytes		

rows in the distance matrix. Accordingly, a set of vertices is statically scheduled to a processor to compute the shortest paths from them.

3. The FFT kernel is a 1D version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize the interprocessor communication. The data set consists of the n complex data points to be transformed and another n complex data points referred to as the roots of unity. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. The simulations are done on an input of 2^{14} points.
4. The Gram-Schmidt (GS) application performs a QR factorization of an $m \times n$ matrix using a parallel shared memory version of the popular modified Gram-Schmidt algorithm. The algorithm constructs an orthogonal set of vectors q, from the columns of a given matrix through an iterative process of subtraction and normalization. For our simulations, we perform the Gram Schmidt algorithm on a 96×192 input matrix.
5. The Gaussian elimination algorithm (GAUSS) is a method for solving matrix equations of the form (Ax = b). We perform the Gaussian elimination on a 128×128 matrix.
6. The Successive over-relaxation of a grid (SOR) application computes the steady state temperature of a metal sheet using a banded parallelization on a 512×512 grid. We run each application on an input size that could be simulated in a reasonable amount of time and that provided good scalability for a 16-processor system. These sizes are smaller than would occur on a real machine, but we have chosen similarly smaller caches.

6.2 Simulation Results

We used detailed simulations to evaluate the impact of our proposed architectures. As changes in technology continue to alter the landscape of what constitutes a major performance bottleneck, it is sometimes worth reexamining published architectural ideas to see whether they can be adapted to serve new purposes. Thus, in this paper, in addition to testing our new switch architectures, we reexamine two techniques: combining and switch cache architectures. We may recollect that the combining technique was proposed in the early 80s [8] to reduce the waiting time incurred by shared memory synchronization, but was not tested with real applications. The switch cache architecture [10] was evaluated with a multistage interconnection network (MIN) which has an inherent tree structure. We now reevaluate the architecture with a mesh network to show that the technique is still very effective. Two performance metrics, hit/miss ratio and execution time of the six applications, are analyzed and studied. We compare these two metrics for the base switch, combining, and switch cache architectures against those on two new switch architectures: SMSHR and SMSHR+cache. In order to perform a fair comparison, we use an overall caching space of 256 bytes (switch cache or MSHR or total) at each switch for each scheme. The SMSHR+cache has eight MSHRs (48 bytes) and seven blocks/lines (224 bytes) of cache. Also, because the switch cache size is small, we only consider a fully associative organization for the cache.

Fig. 10 shows the comparison of the hit/miss ratio of these five schemes (base, combining, switch cache, SMSHR, and SMSHR+cache) for six applications. In the x-axis, *base*, *comb*, *cache*, *mshr*, and *mcache* represent base, combining, switch cache, SMSHR, and SMSHR+cache schemes, respectively. The y-axis represents the hit/miss ratio, normalized to *base*, where no combining, MSHR, or switch cache is used. The first observation from the figure is the great impact of these three schemes on the hit/miss ratio for the first group of applications, where a high percentage of requests get satisfied in the switch cache, SMSHR, and SMSHR+cache schemes. For the combining scheme, the hit ratio is almost 0 percent because this scheme requires that requests for the same block should meet at a switch in a small interval time (four cycles), which rarely happens. For the switch cache scheme, we find that the average hit ratio improves by as much as 35 percent, 45 percent, and 37 percent for the FWA, GS, and GAUSS applications, respectively. This is due to the high degree of read sharing characteristics for these three applications. For the SMSHR

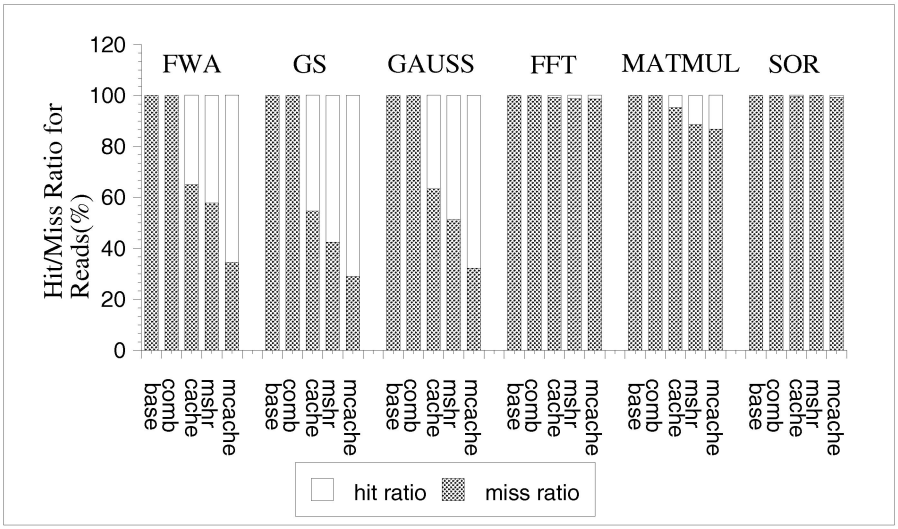


Fig. 10. Hit/miss ratio in remote memory reads.

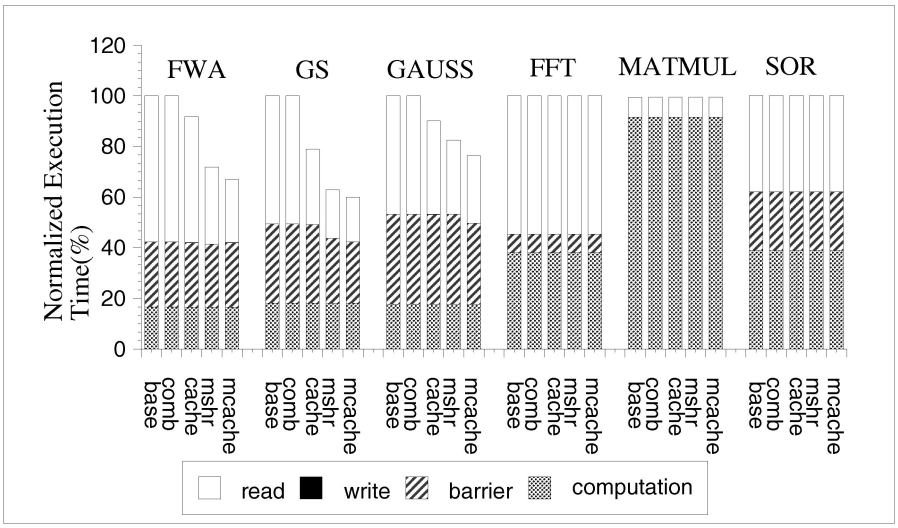
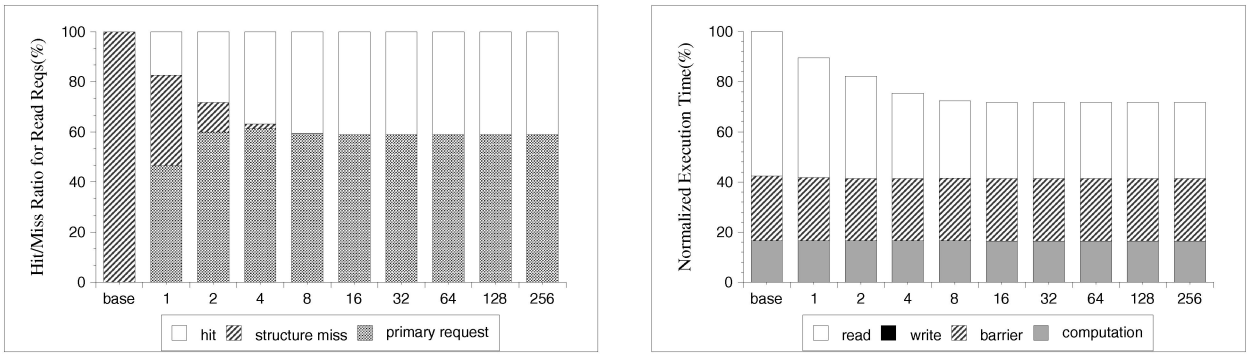


Fig. 11. Execution time improvements for six applications.

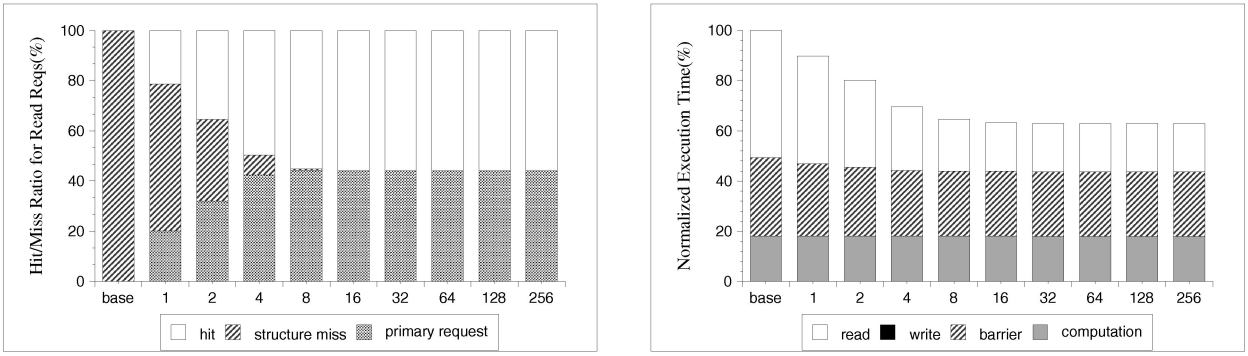
scheme, the average hit ratio can be improved to about 42 percent, 57 percent, and 48 percent for the FWA, GS, and GAUSS applications, respectively. So, the SMSHR technique performs better than the switch cache technique and also with much less buffer space. For the SMSHR+cache scheme, the average hit ratio can improve by as high as about 65 percent, 71 percent, and 67 percent for these applications. Thus, our proposed techniques (SMSHR and SMSHR+cache) should provide excellent performance with low cost. The second observation from the figure is that the impact of these four schemes (combining, switch cache, SMSHR, and SMSHR+cache) on the hit/miss ratio for the second group of applications (FFT, MATMUL, and SOR) is negligible. This is because of their low degree of read sharing and high cache-to-cache transfers. In another paper, we have designed a switch directory scheme [11], which is particularly suitable for these applications.

Let us now concentrate on the impact of these techniques on the execution time, as shown in Fig. 11. The execution time is normalized to *base* and divided into the execution of instructions (computation), the processor stall due to read misses in the memory hierarchy (read stall), the processor stall due to write misses in the memory hierarchy (write stall), and the stall due to the synchronization (barrier). We can verify that the write stall time is negligible because the RSIM simulator uses the release consistency [24]. We do not observe any improvement in the barrier or execution time due to the combining technique. The stall time for the FWA, GS, and GAUSS applications reduces by about 8 percent, 21 percent, and 10 percent by using the switch cache technique and by about 28 percent, 36 percent, and 17 percent by using the SMSHR technique. Using the SMSHR+cache technique, the stall time for the FWA, GS, and GAUSS applications reduces by about 33 percent, 38 percent, and 25 percent, respectively. These results are impressive for the execution time improvement. It is obvious that a high percentage of requests are satisfied by the SMSHR or SMSHR+cache at the interconnection network and avoid accessing memory modules. Also, the network traffic is reduced, which in turn reduces the



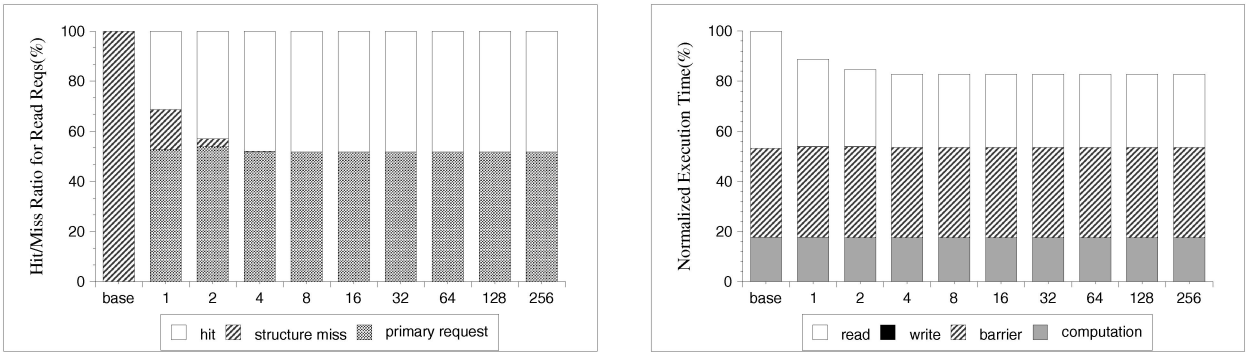
(a) (b)

Fig. 12. Impact of MSHR size on FWA application. (a) Hit/miss ratio. (b) Execution time.



(a) (b)

Fig. 13. Impact of MSHR size on GS application. (a) Hit/miss ratio. (b) Execution time.



(a) (b)

Fig. 14. Impact of MSHR size on GAUSS application. (a) Hit/Miss ratio. (b) Execution time.

memory access times for other requests. The switch design with an MSHR or MSHR+cache increases the complexity and cost of the switch only slightly, but the performance improvements are very good. Moreover, current switches like the Spider [7] already have enormous input buffers, which can be better used for the proposed architectures. Again for the FFT, MATMUL, and SOR applications, the reduction of the execution time is negligible due to the low degree of read sharing.

6.3 Sensitivity Results

We conducted extensive sensitivity studies to test this architecture [27]. For brevity, we present only a few results in this subsection. In order to determine the impact of MSHR size on performance, we varied the number of MSHR entries at each switch from 1 to 256. Figs. 12, 13, and 14 present the impact of changing the number of MSHR entries on the hit/miss ratio and execution time of the FWA, GS, and GAUSS applications. The leftmost bar in each figure corresponds to *base*, where no MSHR is used. We observe that the improvement from one to eight MSHR entries is significant for these applications. As can be seen in Fig. 12, the SMSHR architecture improves the hit ratio by as much as 41 percent when we increase the number of MSHR entries to eight. Also, the SMSHR architecture helps in reducing the execution time by as much as 25 percent when

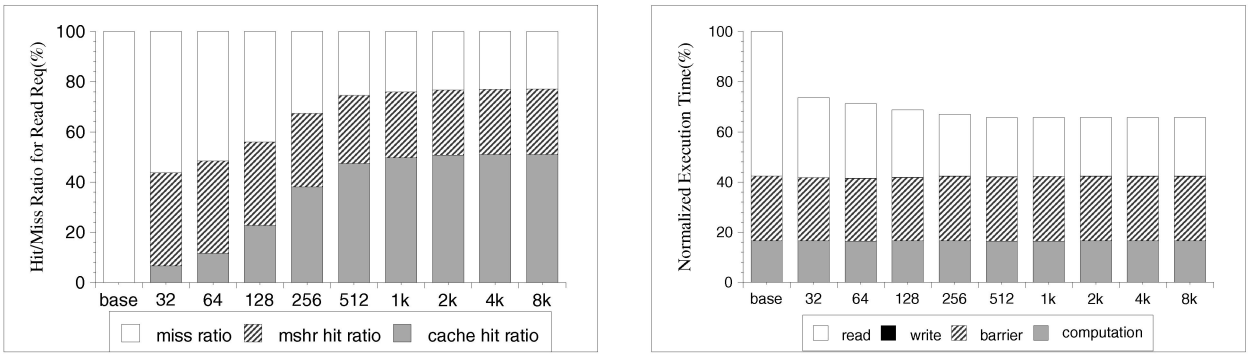


Fig. 15. Impact of cache size on SMSHR+Cache for FWA application. (a) Hit/miss ratio. (b) Execution time.

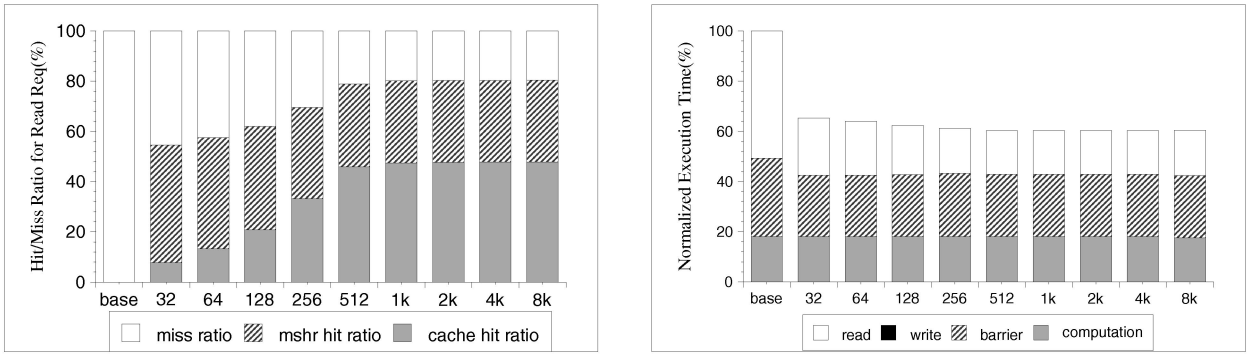


Fig. 16. Impact of cache size on SMSHR+Cache for GS application. (a) Hit/miss ratio. (b) Execution time.

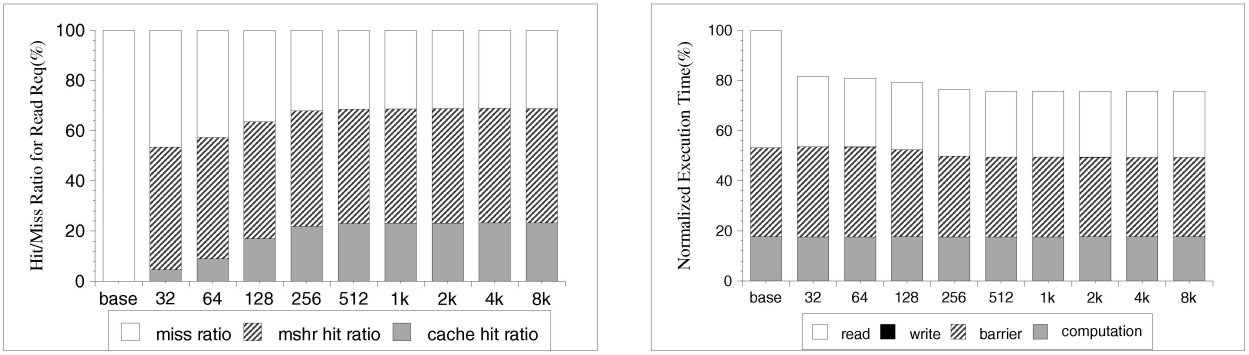


Fig. 17. Impact of cahce size on SMSHR+Cache for GAUSS application. (a) Hit/miss ratio. (b) Execution time.

we increase the number of MSHR entries to eight. For the GS application, the improvements for the hit ratio and execution time are as much as 55 percent and 36 percent with only eight registers, respectively. For the GAUSS application, only four registers are sufficient to improve the hit ratio and execution time by about 48 percent and 17 percent, respectively. The change from MSHR size 8 to 256 has negligible impact on the hit ratio and execution time, indicating that only a few MSHR entries in a switch are sufficient. Eight MSHR entries only occupy about a 48-byte space and, when compared to the switch cache [10], save a lot of buffer space.

In order to determine the impact of switch cache size on the performance, we varied the cache size from 32 bytes to 8 Kbytes for the SMSHR+cache architecture. Here, the number of MSHR entries is fixed at four. Figs. 15, 16, and 17 show the impact of changing cache size on the hit/miss ratio and execution time. The leftmost bar in each figure corresponds to *base*, which has no MSHR or cache. When we increase the cache size, the hit ratios for the FWA, GS, and GAUSS applications improve by as high as about 76 percent, 80 percent, and 68 percent, respectively. The execution times for the FWA, GS, and GAUSS applications improve by as much as 34 percent, 39 percent, and 24 percent,

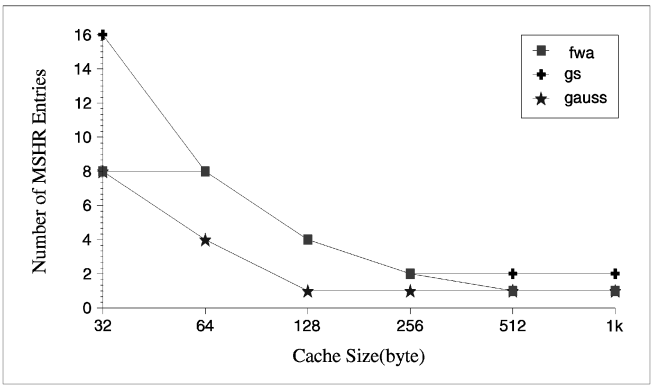


Fig. 18. Optimal MSHR size and cache size for SMSHR+Cache.

respectively. Note that, with a 32-byte cache, the relative hit ratio in the MSHRs is significant. But, as the cache size increases, more cache hits occur and the use of the MSHRs is reduced. The reverse would be true if we were varying the MSHR size and keeping the cache size fixed. It is seen that a small MSHR size and a small cache size of about 256 bytes are enough for a dramatic reduction in the execution time of these applications.

To provide further insight into how the number of MSHR entries should vary with different cache sizes for the SMSHR+cache scheme, we present the minimum number of MSHR entries needed for a given cache size in Fig. 18. A point (X, Y) in each curve is the optimal point which indicates that the execution time is lowest when the cache size is greater than or equal to X and the number of MSHR entries is at least Y. We also find that the common optimal point for the FWA, GS, and GAUSS applications is (256, 2), that is, the execution time for these three applications reaches the absolute lowest when the cache size is 256 bytes and the number of MSHR entries is two. This means that the MSHR size of 12 bytes (two registers) and the cache size of 256 bytes are enough to reach the optimal performance for the SMSHR+cache architecture.

7 Conclusions

In this paper, we presented details of a Switch MSHR (SMSHR) architecture and an SMSHR+cache architecture to reduce the remote memory access time in CC-NUMA multiprocessors. By exploiting the property of data sharing between processors, we were able to block secondary requests to the memory for the same data block. We presented detailed simulation results on these two different switch architectures and showed that the impact on the execution time is significant for applications that have a high degree of read sharing. Improvements in the execution time for some applications were found to be as high as 38 percent.

We observed that the performance of the SMSHR architecture is better than the switch cache architecture. Also, we need only about four to eight MSHR entries per switch, which is negligible buffer space when compared to a switch cache. We performed a sensitivity study of the MSHR and cache parameters for the FWA, GS, GAUSS applications. We found that a small number of MSHR entries and a small cache per switch offer an optimal combination for the performance improvement. We believe that CC-NUMA multiprocessor switches do not need large input buffers. Rather, they should be custom designed with these techniques which occupy significantly less buffer space per switch than what is currently being used. It would have been interesting to test these ideas with commercial applications, but finding source code for these applications is almost impossible in an academic environment.

Acknowledgments

This research has been supported by US National Science Foundation grant CCR-9810205.

References

[1] G. Astfalk and T. Brewer, "An Overview of the HP/Convex Exemplar Hardware," http://www.convex.com/tech_cache/ps/hw_ov.ps, 1997.

[2] J.K. Bennett, K.E. Bennett, and W.E. Speight, "The Performance Value of Shared Network Caches in Clustered Multiprocessor Workstations," *Proc. 16th Int'l Conf. Distributed Computing Systems,* pp. 64-74, May 1996.

[3] J. Carbonaro and F. Verhoorn, "Cavallino: The Teraflops Router and NIC," *Proc. Symp. High Performance Interconnects (Hot Interconnects 4),* pp. 157-160, Aug. 1996.

[4] W.J. Dally and H. Aoki, "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 4, pp. 466-475, Apr. 1993.

[5] K.I. Farkas and N.P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 211-222, Apr. 1994.

[6] K.E. Fletcher, W.E. Speight, and J.K. Bennett, "Techniques for Reducing the Impact of Inclusion in Shared Network Cache Multiprocessors," Technical Report 9413, Dept. of Electrical and Computer Eng., Rice Univ., Dec. 1994.

[7] M. Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip," *Proc. Symp. High Performance Interconnects (Hot Interconnects 4),* pp. 141-146, Aug. 1996.

[8] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers,* vol. 32, no. 2, pp. 175-189, Feb. 1983.

[9] IBM Corp., "Blue Gene Project Update," http://www.research.ibm.com/bluegene/, Jan. 2002.

[10] R. Iyer and L.N. Bhuyan, "Design and Evaluation of a Switch Cache Architecture for CC-NUMA Multiprocessors," *IEEE Trans. Computers,* vol. 49, no. 8, pp. 779-797, Aug. 2000.

[11] R. Iyer, L.N. Bhuyan, and A. Nanda, "Using Switch Directories to Speed Up Cache-to-Cache Transfers in CC-NUMA Multiprocessors," *Proc. 12th Int'l Parallel and Distributed Processing Symp.,* pp. 721-728, May 2000.

[12] S. Kaxiras, S. Gjessing, and J. Goodman, "A Study of Three Dynamic Approaches to Handle Widely Shared-Memory Multiprocessors," *Proc. Int'l Conf. Supercomputing,* pp. 457-464, July 1998.

[13] S. Kaxiras and J. Goodman, "Kiloprocessor Extensions to SCI," *Proc. Int'l Parallel Processing Symp.,* pp. 166-172, Apr. 1996.

[14] Kendall Square Research, "KSR1 Principles of Operations," Waltham, Mass., rev. 5.5 ed., Oct. 1991.

[15] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proc. Eighth Ann. Int'l Symp. Computer Architecture,* pp. 81-87, May 1981.

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

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford DASH Multiprocessor," *Computer,* vol. 25, vol. 3, pp. 63-79, Mar. 1992.

[18] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 308-317, May 1996.
[19] M. Michael and A. Nanda, "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors," *Proc. Fifth Int'l Symp. High Performance Computer Architecture,* pp. 142-151, Jan. 1999.
[20] A. Moga and M. Dubois, "The Effectiveness of SRAM Network Caches on Clustered DSMs," *Proc. Fourth Int'l Symp. High Performance Computer Architecture,* pp. 103-112, Feb. 1998.
[21] A. Nanda and L. Bhuyan, "Design and Analysis of Cache Coherent Multistage Interconnection Networks," *IEEE Trans. Computers,* vol. 42, no. 4, pp. 458-470, Apr. 1993.
[22] B.A. Nayfeh and K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 166-175, Apr. 1994.
[23] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Parkin, "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. Int'l Conf. Parallel Processing,* pp. 1-10, Aug. 1995.
[24] V.S. Pai, P. Ranganathan, and S.V. Adve, "RSIM Reference Manual," Technical Report 9705, Dept. of Electrical and Computer Eng., Rice Univ., Aug. 1997.
[25] S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Tempest and Typhoon: User-Level Shared Memory," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 325-337, Apr. 1994.
[26] A. Saulsbury, T. Wlkinson, J. Carder, and A. Landin, "An Argument for Simple COMA," *Proc. First Int'l Symp. High-Performance Computer Architecture,* pp. 276-285, Jan. 1995.
[27] H. Wang, "Efficient Memory Management and Interconnection Schemes for CC-NUMA Multiprocessors," PhD dissertation, Dept. of Computer Science, Texas A&M Univ., May 2002.
[28] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke, "The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 98-107, June 1997.

Laxmi Narayan Bhuyan received the MSc degree in electrical engineering from Sambalpur University, India, in 1979, and the PhD degree in computer engineering from Wayne State University, Detroit, Michigan, in 1982. At present, he is a professor of computer science and engineering at the University of California, Riverside,. His research interests are in the areas of computer architecture, parallel processing, interconnection networks, and performance evaluation. He has published more than 100 papers in these areas. Dr. Bhuyan has served on the editorial boards of *Computer*, the *Journal of Parallel and Distributed Computing* (JPDC), *IEEE Transactions on Parallel and Distributed Systems*, and *Parallel Computing*. He was the founding program committee chairman of the First International Symposium on High-Performance Computer Architecture (HPCA), January 1995, and later chairman of the IEEE Computer Society Technical Committee on Computer Architecture (TCCA) between 1996-1998. He is a fellow of the IEEE and ACM.

Hujun Wang received the MS degree in computer science from NanJing University of Science and Technology, China, in 1993, and the BS degree in computer science from the East China Shipbuilding Institute of Technology, China, in 1990. He is currently working toward the PhD degree in the Department of Computer Science at Texas A&M University. His current research interests include memory management, parallel applications, and interconnection networks.