Design and Evaluation of a Switch Cache Architecture for CC-NUMA Multiprocessors

Ravishankar R. Iyer, *Member*, *IEEE*, and Laxmi N. Bhuyan, *Fellow*, *IEEE*

Abstract—Cache coherent nonuniform memory access (CC-NUMA) multiprocessors provide a scalable design for shared memory. But, they continue to suffer from large remote memory access latencies due to comparatively slow memory technology and large data transfer latencies in the interconnection network. In this paper, we propose a novel hardware caching technique, called *switch cache*, to improve the remote memory access performance of CC-NUMA multiprocessors. The main idea is to implement small fast caches in crossbar switches of the interconnect medium to capture and store shared data as they flow from the memory module to the requesting processor. This stored data acts as a cache for subsequent requests, thus reducing the need for remote memory accesses tremendously. The implementation of a cache in a crossbar switch needs to be efficient and robust, yet flexible for changes in the caching protocol. The design and implementation details of a **CA**che **E**mbedded **S**witch **AR**chitecture, *CAESAR*, using wormhole routing with virtual channels is presented. We explore the design space of switch caches by modeling CAESAR in a detailed execution driven simulator and analyze the performance benefits. Our results show that the CAESAR switch cache is capable of improving the performance of CC-NUMA multiprocessors by up to 45 percent reduction in remote memory accesses for some applications. By serving remote read requests at various stages in the interconnect, we observe improvements in execution time as high as 20 percent for these applications. We conclude that switch caches provide a cost-effective solution for designing high performance CC-NUMA multiprocessors.

Index Terms—Crossbar switches, cache architectures, scalable interconnects, wormhole routing, shared memory multiprocessors, execution-driven simulation.

1 Introduction

To alleviate the problem of high memory access latencies, shared memory multiprocessors employ processors with small fast on-chip caches and additionally larger off-chip caches. Symmetric multiprocessor (SMP) systems are usually built using a shared global bus. However, the contention on the bus and memory heavily constrains the number of processors that can be connected to the bus. To build high performance systems that are scalable, several current systems [1], [12], [14], [15] employ the cache coherent nonuniform memory access (CC-NUMA) architecture. In such a system, the shared memory is distributed among all the nodes in the system to provide a closer local memory and several remote memories. While local memory access latencies can be tolerated, the remote memory accesses generated during the execution can bring down the performance of applications drastically.

To reduce the impact of remote memory access latencies, researchers have proposed improved caching strategies [16], [20], [29] within each cluster of the multiprocessor. These caching techniques are primarily based on data sharing among multiple processors within the same cluster. Nayfeh et al. [20] explore the use of shared L2 caches to benefit from the shared working set between the processors within a cluster. Another alternative is the use of network caches or remote data caches [16], [29]. Network caches can be considered to be shared L3 caches to processors within a cluster. They take advantage of shared working set effects and reduce the remote access penalty by serving capacity misses of L2 caches. The HP Exemplar [1] implements the network cache as a configurable partition of the local memory. Sequent's NUMA-Q [15] dedicates a 32MB DRAM memory for the network cache. The DASH multiprocessor [14] has provision for a network cache called the remote access cache. A recent proposal by Moga and Dubois [16] explores the use of small SRAM (instead of DRAM) network caches integrated with a page cache. The use of 32KB SRAM chips reduces the access latency of network caches tremendously.

Our goal is to reduce remote memory access latencies by implementing a global shared cache abstraction central to all processors in the CC-NUMA system. We observe that network caches provide such an abstraction limited only to the processors within a cluster. We explore the implementation issues and performance benefits of a multilevel caching scheme that can be incorporated into current CC-NUMA systems. By embedding a small fast SRAM cache within each switch in the interconnection network, called *switch cache*, we capture shared data as it flows through the interconnect and provide it to future accesses from processors that reuse this data. Such a scheme can be considered as a multilevel caching scheme, but without inclusion property. Our studies on application behavior indicate that there is enough spatial and temporal locality between requests from processors to benefit from small switch caches. Previous studies [17]

- *R.R. Iyer is with Intel Corporation, 15220 NW Greenbrier Pkwy., Beaverton, OR 97006. E-mail: ravishankar.iyer@intel.com.*
- *L.N. Bhuyan is with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. E-mail: bhuyan@cs.tamu.edu.*

Manuscript received 26 Apr. 1999; accepted 17 Dec. 1999.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109690.

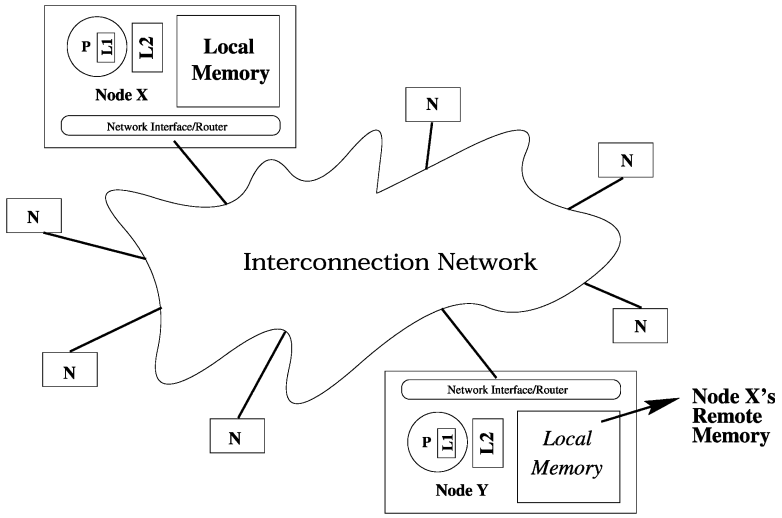


Fig. 1. CC-NUMA system and memory hierarchy.

used synthetic workloads and showed that increasing the buffer size in a crossbar switch beyond a certain value does not have much impact on network performance. Our application-based study [5] confirms that this observation holds true for the CC-NUMA environment running several scientific applications. Thus, we think that the large amount of buffers in current switches, such as SPIDER [8], is overkill. A better utilization of these buffers can be accomplished by organizing them as a switch cache.

There are several issues to be considered while designing such a caching technique. These include cache design issues such as technology and organization, cache coherence issues, switch design issues such as arbitration, and message flow control issues such as appropriate routing strategy, message layout, etc. The first issue is to design and analyze a cache organization that is large enough to hold the reusable data, yet fast enough to operate during the time a request passes through the switch. The second issue involves modifying the directory-based cache protocol to handle an additional caching layer at the switching elements so that all the cache blocks in the system are properly invalidated on a write. The third issue is to design buffers and arbitration in the switch which will guarantee certain cache actions within the switch delay period. For example, when a read request travels through a switch cache, it must not incur additional delays. Even in the case of a switch cache hit, the request must be passed on to the home node to update the directory, but not generate a memory read operation. The final issue deals with message header design to enable request encoding, network routing, etc.

The contribution of this paper is the detailed design and performance evaluation of a switch cache interconnect employing a **CA**che **E**mbedded **S**witch **AR**chitecture (CAESAR). The CAESAR switch cache is made up of a small SRAM cache operating at the same speed as a wormhole routed crossbar switch with virtual channels. The switch design is optimized to maintain crossbar bandwidth and throughput, while at the same time providing sufficient switch cache throughput and improved remote access performance. The performance evaluation of the switch cache interconnect is conducted using six scientific applications. We present several sensitivity studies to cover the entire design space and to identify the important parameters. Our experiments show that switch caches offer a great potential for use in future CC-NUMA interconnects for many of these applications.

The rest of the paper is organized as follows: Section 2 provides a background on the remote access characteristics of several applications in a CC-NUMA environment and builds the motivation behind our proposed global caching approach. The switch cache framework and the caching protocol are presented in Section 3. Section 4 covers the design and implementation of CAESAR. Performance benefits of the switch cache framework are evaluated and analyzed in great detail in Section 5. Sensitivity studies over various design parameters are also presented in Section 5. Finally, Section 6 summarizes and concludes the paper.

2 Application Characteristics and Motivation

Several current distributed shared memory multiprocessors have adopted the CC-NUMA architecture since it provides transparent access to data. Fig. 1 shows the disparities in proximity and access time in the CC-NUMA memory hierarchy of such systems. A load or store issued by processor X can be served in a few cycles upon L1 or L2 cache hits, in less than a hundred cycles for local memory access, or incurs a few hundred cycles due to a remote memory access. While the latency for stores to the memory (write transactions) can be hidden by the use of weak consistency models, the stall time due to loads (read transactions) to memory can severely degrade application performance. Data prefetching and speculation techniques based on per-processor request streams have been successful in reducing the load latency to a considerable extent. In this paper, we take a different approach that targets application's sharing patterns and evaluates benefits of a global, yet decentralized, caching mechanism.

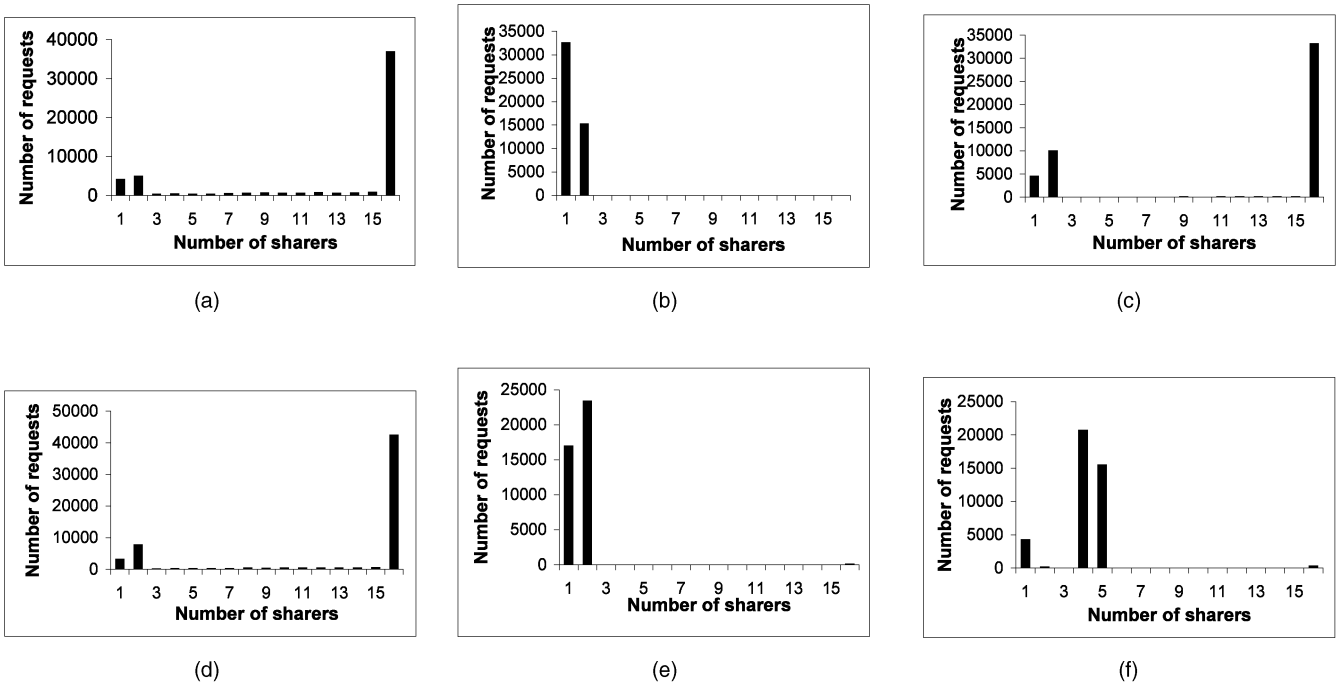


Fig. 2. Application read sharing characteristics. (a) FWA, (b) GS, (c) GAUSS, (d) SOR, (e) FFT, (f) MM.

2.1 Global Cache Benefits: A Trace Analysis

To reduce the impact of remote read transactions, we would like to exploit the locality in sharing patterns between the processors. Fig. 2 plots the read sharing pattern for six applications with 16 processors using a cache line size of 32 bytes. The six applications used in this evaluation are the Floyd-Warshall's Algorithm (FWA), Gaussian Elimination (GAUSS), Gram-Schmidt (GS), Matrix Multiplication (MM), Successive Over Relaxation (SOR), and the Fast Fourier Transform (FFT). The x-axis represents the number of sharing processors (X) while the y-axis denotes the number of accesses to blocks shared by X number of processors. From the figure, we observe that, for four out of the six applications (FWA, GAUSS, GS, MM), multiple processors read the same block between two consecutive writes to that block. These shared reads form a major portion (35 to 80 percent) of the application's read misses. To take advantage of such read-sharing patterns across processors, we introduce the concept of an ideal global cache that is centrally accessible to all processors. When the first request is served at the memory, the data sent back in the reply message is stored in a global cache. Since the cache is accessible by all processors, subsequent requests to the data item can be satisfied by the global cache at low latencies. There are two questions that arise here:

- *What is the time lag between two accesses from different processors to the same block?* We define this as *temporal read sharing locality* between the processors, somewhat equivalent to temporal locality in a uniprocessor system. The question raised here particularly relates to the size and organization of the global cache. In general, it would translate to: the longer the time lag, the bigger the size of the required global cache.
- *Given that a block can be held in a central location, how many requests can be satisfied by this cached block?* We call this term *attainable read sharing* to estimate the performance improvement by employing a global cache. Again, this metric will give an indication of the required size for the global cache.

To answer these questions, we instrumented a simulator to generate an execution trace with information regarding each cache miss. We designed a trace analysis tool, *SILA*—Sharing Identifier and Locality Analyzer—and fed the traces to it. In order to evaluate the potential of a global cache, SILA generates two different sets of data: temporal read shared locality (Fig. 3), and attainable sharing (Fig. 4). The data sets can be interpreted as follows:

Temporal Read Sharing Locality: Fig. 3 depicts the temporal read sharing locality among consecutive read transactions to the same block. A point $\{X, Y\}$ from this data set indicates that Y is the probability that two read transactions (from different processors) to the same block occur within a time distance of X or lower (i.e., $Y = P(x \leq X)$, where x is the average interarrival time between two consecutive read requests to the same block). As seen in the figure, most applications have an inherent temporal reuse of the cached block by other processors. The interarrival time between two consecutive shared read transactions from different processors to the same block is found to be less than 500 processor cycles (pcycles) for 60-80 percent of the shared read transactions for all applications, except SOR. Ideally, this indicates a potential for at least one extra request to be satisfied per globally cached block.

Attainable Read Sharing: Fig. 4 explores the probability of multiple requests satisfied by the global caching technique termed as attainable sharing degree. A point $\{X, Y\}$ in this data set indicates that if each block can be

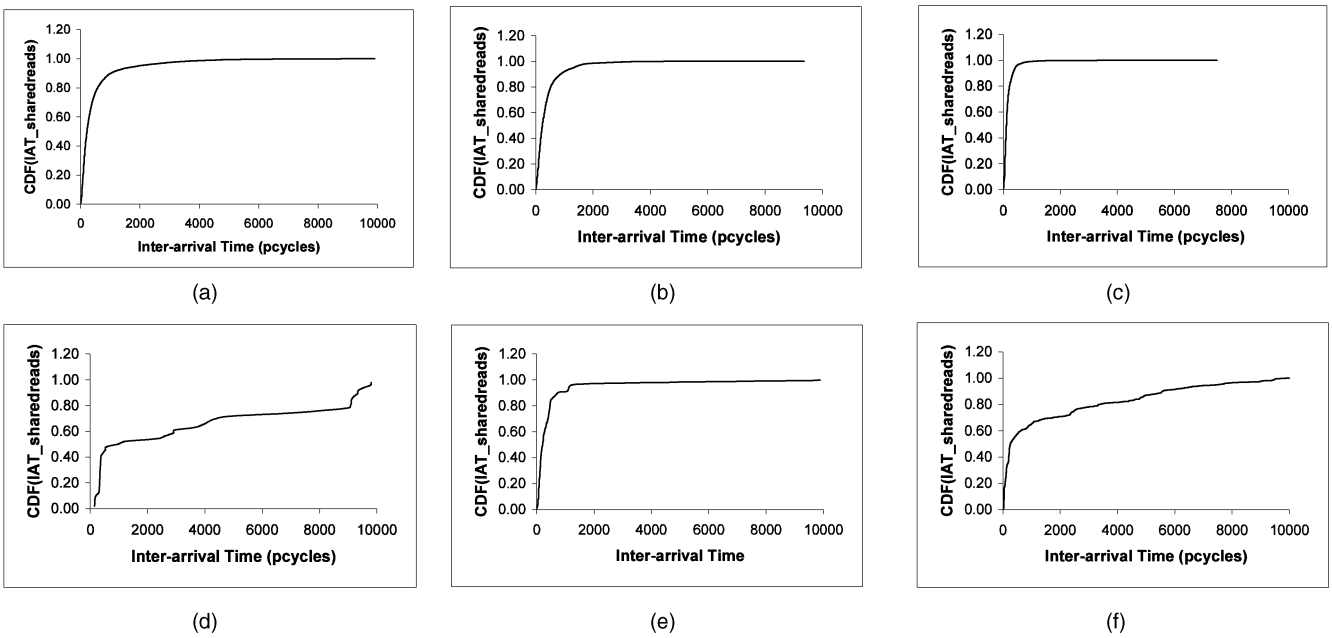


Fig. 3. The temporal locality of shared accesses. (a) FWA, (b) GS, (c) GAUSS, (d) SOR, (e) FFT, (f) MM.

held for X cycles in the global cache, the average number of subsequent requests per block that can be served is Y. The figure depicts the attainable read sharing degree for each application based on the residence time for the block in the global cache. The residence time of a cache block is defined as the amount of time the block is held in the cache before it is replaced or invalidated. While invalidations cannot be avoided, note that the residence time directly relates to several cache design issues such as cache size, block size and associativity. From Fig. 2, we observed that FWA, GS, and GAUSS have high read sharing degrees close to the number of processors in the system (in this case, 16 processors). However, it is found that the attainable sharing degree varies according to the temporal locality of each application. While GAUSS can attain a sharing degree of 10 in the global cache with a residence time of $2,000$ processor cycles, GS requires that the residence time be $5,000$ and FWA requires that this time be $7,000$. The MM application has a sharing degree of approximately four to five, whereas the attainable sharing degree is much lower. SOR and FFT are not of much interest since they have a very low percentage (1-2 percent) of accesses to blocks shared by more than two processors.

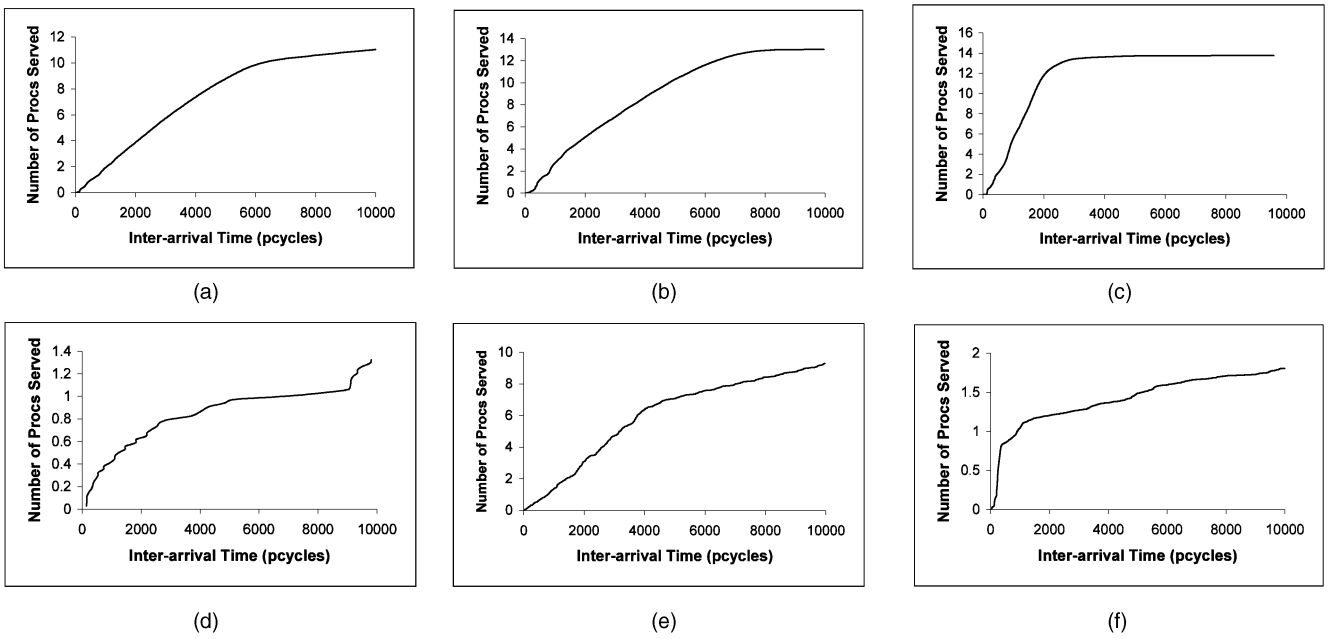


Fig. 4. The attainable sharing degree. (a) FWA, (b) GS, (c) GAUSS, (d) SOR, (e) FFT, (f) MM.

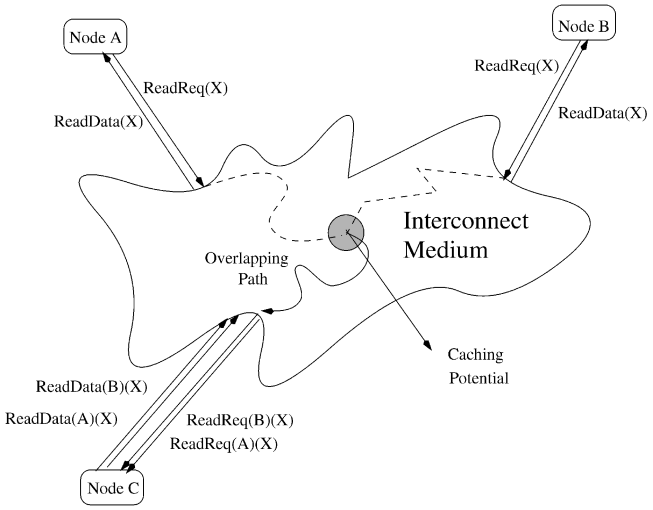


Fig. 5. The caching potential of the interconnect medium.

2.2 The Interconnect as a Global Cache

In Section 2.1, we identified the need for global caching to improve the remote access performance of CC-NUMA systems. In this section, we explore the possible use of the interconnect medium as a central caching location. We attempt to answer the following two important questions:

- *What makes the interconnect medium a suitable candidate for central caching?*
- *Which interconnect topology is beneficial for incorporating a central caching scheme?*

Communication in a shared memory system occurs in transactions that consist of requests and replies. The requests and replies traverse from one node to another through the interconnection network. The network is the only medium in the entire system that can keep track of all the transactions between the nodes. The potential for such a network caching strategy is illustrated in Fig. 5. The path of two read transactions to the same block X in node C that emerge from processors A and B overlap at some point in the network. The common elements in the network can act as small caches for replies from memory. A later request to the recently accessed block X can potentially find the block cached in the common routing element (illustrated by a shaded circle). The benefit of such a scheme is two-fold. From the processor point of view, the read request gets serviced at a latency much lower than the remote memory access latency. Second, the memory is relieved of servicing the requests that hit in the global interconnect cache, thus improving the access times of other requests that are sent to it.

From the example, we observe that incorporating such schemes in the interconnect requires a significant amount of path overlap between processor to memory requests. Also, replies must follow the same path as requests in order to provide an intersection. The routing/flow of requests and replies depends upon the topology of the interconnect. The ideal topology for such a system is the global bus. However, the global bus is not a scalable medium and bus contention severely degrades performance when the number of processors increases beyond a certain threshold. Consequently, multiple bus hierarchies and fat-tree structures have been considered effective solutions to the scalability issue. The tree structure provides the next best alternative to hierarchical caching schemes.

3 The Switch Cache Framework

In this section, we present a new hardware realization of the ideal global caching solution for improving the remote access performance of shared memory multiprocessors.

3.1 The Switch Cache Interconnect

Network topology plays an important role in determining the paths from a source to a destination in the system. Tree-based networks like the fat tree [13], the heirarchical bus network [25], [4], and the multistage interconnection network (MIN) [19] provide hierarchical topologies suitable for global caching. In addition, the MIN is highly scalable and it provides a bisection bandwidth that scales linearly with the number of nodes in the system. These features of the MIN make it very attractive as scalable high performance interconnects for commercial systems. Existing systems such as Butterfly [2], CM-5 [13], and IBM SP2 [23] employ a bidirectional MIN. The Illinois Cedar multiprocessor [24] employs two separate unidirectional MINs (one for requests and one for replies). In this paper, the switch cache interconnect is a bidirectional MIN to take advantage of the inherent tree structure. Note, however, that logical trees can also be embedded on other popular direct networks like the mesh and the hypercube [12], [14].

The baseline topology of the 16-node bidirectional MIN (BMIN) is shown in Fig. 6a. In general, an N-node BMIN system using $k \times k$ switches is comprised of N/k switching elements (a $2k \times 2k$ crossbar) in each of the $log_k N$ stages. We chose wormhole routing with virtual channels as the switching technique because it is prevalent in current systems such as the SGI Origin [13].

In a shared memory system, communication between nodes is accomplished via read/write transactions and coherence requests/acknowledgments. The read/write requests and coherence replies from the processor to the memory use forward links to traverse through the switches. Similarly, read/write replies with data and coherence requests from memory to the processor traverse the backward path, as shown in bold in Fig. 6a. Separating the paths enables separate resources and reduces the possibility of deadlocks in the network. At the same time, routing them through the same switches provides identical paths for requests and replies for a processor-memory pair that is essential to developing a switch cache hierarchy. The BMIN tree structure that enables this hierarchy is shown in Fig. 6b.

The basic idea of our caching strategy is to utilize the tree structure in the BMIN and the path overlap of requests, replies, and coherence messages to provide coherent shared data in the interconnect. By incorporating small fast caches in the switching elements of the BMIN, we can serve the requests that pass through these switching elements. We use the term *switch cache* to differentiate these caches from the processor cache. An example of a BMIN employing switch caches that can serve multiple processors is shown in Fig. 6c. An initial shared read request from processor P_i to a

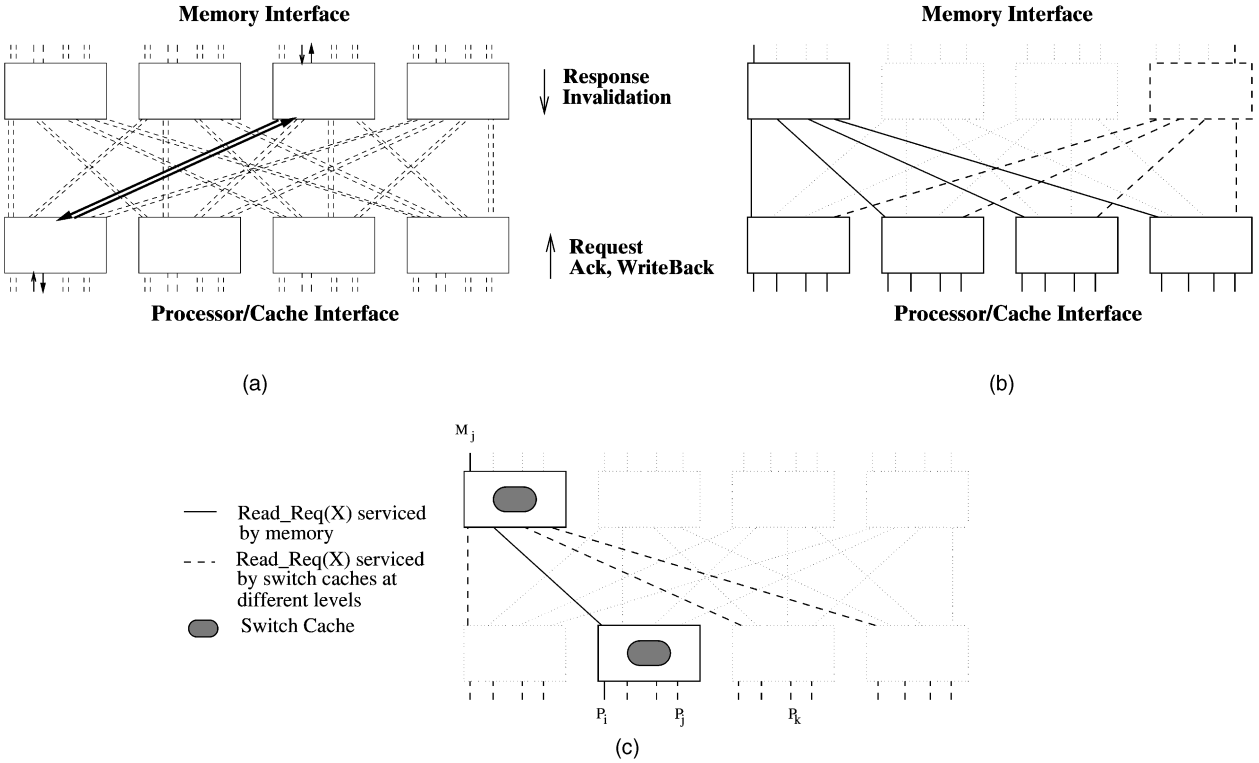


Fig. 6. The switch cache interconnect. (a) Bidirectional MIN—structure and routing. (b) Multiple tree structures in the BMIN. (c) Switch caching in the BMIN.

block is served at the remote memory M_j. When the reply data flows through the interconnection network, the block is captured and saved in the switch cache at each switching element along the path. Subsequent requests to the same block from sharing processors, such as P_j and P_k, take advantage of the data blocks in the switch cache at different stages, thus incurring reduced read latencies.

3.2 The Caching Protocol

The incorporation of processor caches in a multiprocessor introduces the well-known cache coherence problem. Many hardware cache-coherent systems employ a full-map directory scheme [7], In this scheme, each *home node* maintains a bit vector to keep track of all the sharers of each block in its local shared memory space. On every write, an ownership request is sent to the home node, invalidations are sent to all the sharers of the block, and the ownership is granted only when all the corresponding acknowledgments are received. At the processing node, each cache employs a three-state (MSI) protocol to keep track of the state of each cache line. Incorporating switch caches comes with the requirement that all caches remain coherent and data access remain consistent with the system consistency model.

Our basic caching scheme can be represented by the state diagram in Fig. 7 and explained as follows: The switch cache stores only blocks in a shared state in the system. *When a block is read to the processor cache in a dirty state, it is not cached in the switch.* Effectively, the switch cache needs to employ only a 2-state protocol where the state of a block can be either SHARED or NOT_VALID. The transitions of blocks from one state to another is shown in Fig. 7a. When a block is read into a processor cache in shared state ($ReadReply$), the switch caches along the path also capture the block in SHARED state. A SHARED block in the switch cache can be evicted in two possible ways. First, an invalidation to the processor cache ($InvType$) will also invalidate the block in the switch cache, if present. Second, when cache line conflicts occur, data entered by a reply message ($ReadReply^*$) can replace an existing cache block in SHARED state. To illustrate the difference between block invalidations ($InvType$) and block replacements ($ReadReply^*$), the figure shows the NOT_VALID state conceptually separated into two states INVALID and NOT_PRESENT, respectively. An obvious enhancement to this scheme is to incorporate a TRANSIENT state when a request is initiated to a block that does not exist in the switch cache. Thus, subsequent requests to the TRANSIENT block can be held at the switch cache until a reply arrives to the first request. This enhancement increases the complexity of the crossbar switch cache design and is not discussed in this paper. A detailed description of our caching protocol based on request, reply, and coherence messages is as follows:

Read Requests: Each read request message that enters the interconnect checks the switch caches along its path. In the event of a *switch cache hit*, the switch cache is responsible for providing the data and sending the reply to the requestor. The original message is marked as switch hit and it continues to the destination memory (ignoring subsequent switch caches along its path) with the sole purpose of informing the home node that another sharer just read the block. Such a message is called a $marked$ read request. This request is necessary to maintain the full-map directory protocol. Note that memory access is not needed

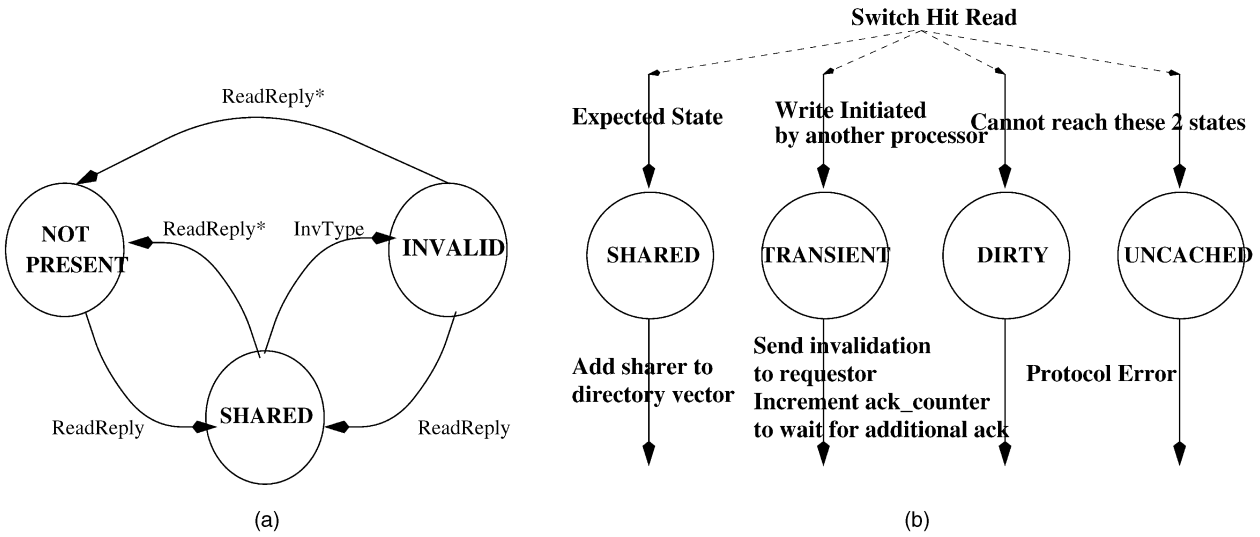


Fig. 7. Switch cache protocol execution. (a) Switch cache state diagram. (b) Change in directory protocol.

for these *marked* requests and no reply is generated. At the destination memory, a *marked* read request can find the block in only two states, SHARED, or TRANSIENT (see Fig. 7b). If the directory state is SHARED, then this request only updates the sharing vector in the directory. However, it is possible that a write has been initiated to the same cache line by a different processor and the invalidation for this write has not yet propagated to the switch cache. This can only be present due to false sharing or in an application that allows data race conditions to exist. If this occurs, then the *marked* read request observes a TRANSIENT state at the directory. In such an event, the directory sends an invalidation to the processor that requested the read and waits for this additional acknowledgment before committing the write.

Read Replies: Read replies originate from the memory node and enter the interconnect following the backward path. The read reply should check the switch cache along its path. If the line is not present in a switch cache, the data carried by the read reply is written into the cache. The state of the cache line is marked SHARED.

Writes, Write-backs, and Coherence Messages: These requests flow through the switches, check the switch cache, and invalidate the cache line if present in the cache. By doing so, the switch cache coherence is maintained somewhat transparently.

Message Format: The messages are formatted at the network interface where requests, replies, and coherence messages are prepared to be sent over the network. In a wormhole-routed network, messages are made up of flow control digits or flits. Each flit is 8 bytes as in Cavallino [6]. The message header contains the routing information, while data flits follow the path created by the header. The format of the message header is shown in Fig. 8. To implement the caching technique, we require that the header consists of three additional bits of information. Two bits ($Reqtype - R_1R_2$) are encoded to denote the switch cache access type as follows:

- *00 - sc_read*—Read a cache line from the switch cache. If present, mark read header and generate reply message.
- *01 - sc_write*—Write cache line into the switch cache.
- *10 - sc_inv*—Invalidate the cache line, if present in the cache.
- *11 - sc_ignore*—Ignore switch cache, no processing required.

Note from the above description and the caching protocol that read requests are encoded as sc_read requests. Read replies whose home node and requestor id are different are encoded as sc_write. Coherence messages, write ownership requests, and write-back requests are encoded as sc_inv requests. All other requests can be encoded as sc_ignore requests. An additional bit is required to mark sc_read requests as earlier switch cache hit. Such a request is called a $marked$ read request. This is used to avoid multiple caches servicing the same request. As discussed, such a marked request only updates the directory and avoids a memory access.

4 Switch Cache Design and Implementation

Crossbar switches provide an excellent building block for scalable high performance interconnects. Crossbar switches mainly differ in two design issues: *switching technique* and *buffer management*. We use wormhole routing as the switching technique and input buffering with virtual channels since these are prevalent in current commercial crossbar switches [6], [8]. We begin by presenting the organization and design alternatives for a 4×4 crossbar

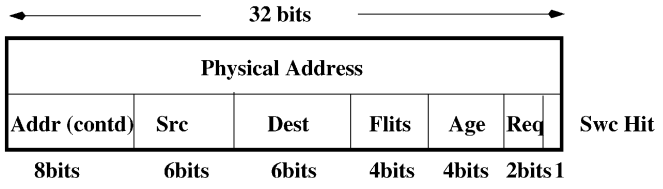


Fig. 8. Message header format.

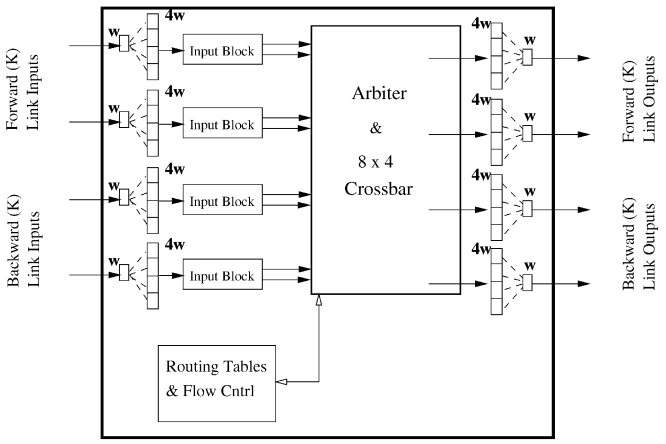


Fig. 9. A conventional crossbar switch.

switch cache. In a later subsection, the extensions required for incorporating a switch cache module into a larger (8×8) crossbar switch are presented.

4.1 Switch Cache Organization

Our base bidirectional crossbar switch has four inputs and four outputs, as shown in Fig. 9. Each input link in the crossbar switch has two virtual channels, thus providing eight possible input candidates for arbitration. The arbitration process is the *age* technique, similar to that employed in the SGI Spider Switch [8]. At each arbitration cycle, a maximum of four highest age flits are selected from eight possible arbitration candidates. The flit size is chosen to be 8 bytes ($4w$) with 16 bit links, similar to the Intel Cavallino [6]. Thus, it takes four link cycles to transmit a flit from output of one switch to the input of the next. Buffering in the crossbar switch is provided at the *input block* at each link. The input block is organized as a fixed size FIFO buffer for each virtual channel that stores flits belonging to a single message at a time. The virtual channels are also partitioned based on the destination node. This avoids out-of-order arrival of messages originating from the same source to the same destination. We also provide a bypass path for the incoming flits that can be directly transmitted to the output if the input buffer is empty.

While organizing the switch cache, we are particularly interested in maximizing performance by serving flits within the cycles required for the operation of the base crossbar switch. Thus, the organization depends highly on the delay experienced for link transmission and crossbar switch processing. Here, we present two different alternatives for organizing the switch cache module within conventional crossbar switches. The *arbitration-independent organization* is based on a crossbar switch operation similar to the SGI Spider [8]. The *arbitration-dependent organization* is based on a crossbar switch operation similar to the Intel Cavallino [6].

Arbitration-Independent Organization: This switch cache organization is based on link and switch processing delays similar to those experienced in the SGI Spider. The internal switch core runs at 100 MHz, while the link transmission operates at 400 MHz. The switch takes four 100 MHz clocks to move flits from the input to the link transmitter at the output. The link, on the other hand, can transmit an 8 byte flit in a single 100 MHz clock (four 400 MHz clocks). Fig. 10a shows the arbitration-independent organization of the switch cache. The organization is arbitration independent because the switch cache performs the critical operations in parallel with the arbitration operation. At the beginning of each arbitration cycle, a maximum of four input flits stored in the input link registers are transmitted to the switch cache module. In order to maintain flow control, all required switch cache processing should be performed in parallel with the arbitration and transmission delay of the switch (four cycles).

Arbitration-Dependent Organization: This switch cache organization is based on link and switch processing delays similar to those experienced in the Intel Cavallino [6]. The internal switch core and link transmission operate at 200MHz. It takes four cycles for the crossbar switch to pass the flit from its input to the output transmitter and four cycles for the link to transmit one flit from one switch to another. Fig. 10b shows the arbitration-dependent organization of the switch cache. The organization is arbitration dependent because it performs the critical operations at the end of the arbitration cycle and in parallel with the output link transmission. At the end of every arbitration cycle, a

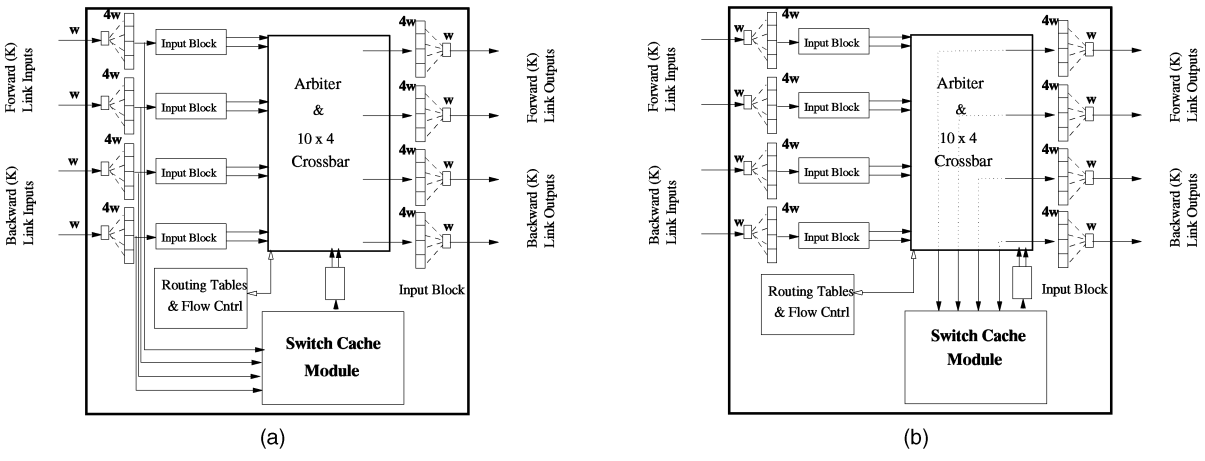


Fig. 10. Crossbar switch cache organization. (a) Arbitration-independent. (b) Arbitration-dependent.

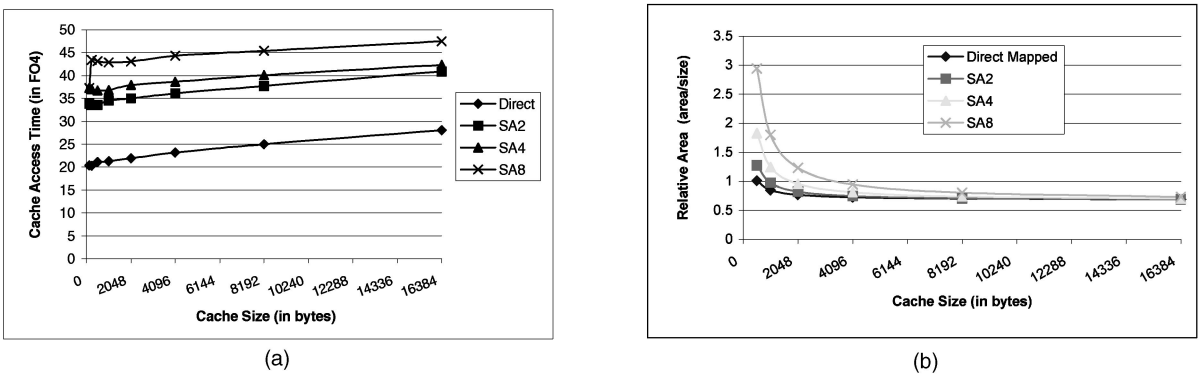


Fig. 11. Cache area and access time issues. (a) Access time (in FO4). (b) Relative area.

maximum of four flits passed through the crossbar from input buffers to the output link transmitters are also transmitted to the switch cache. Since the output transmission takes four 200 MHz cycles, the switch cache needs to process a maximum of four flits within four cycles.

Each organization has its advantages/disadvantages. In the arbitration-independent organization, the cache operates at the switch core frequency and remains independent of the link speed. On the other hand, this organization lacks arbitration information which could be useful for completing operations in an orderly manner. While this issue does not affect 4×4 switches, the drawback will be evident when we study the design of larger crossbar switches. The arbitration-dependent organization benefits from the completion of the arbitration phase and can use the resultant information for timely completion of switch cache processing. However, in this organization, the cache is required to run at link transmission frequency in order to complete critical switch cache operations. As in the Intel Cavallino, it is possible to run the switch core, the switch cache, and the link transmission at the same speed.

Finally, note that, in both cases, the reply messages from the switch cache module are stored in another input block, as shown in Fig. 10. With two virtual channels per input block, the crossbar switch size expands to 10×4. Also, in both cases, the maximum number of switch cache inputs are four requests and processing time is limited to four switch cache cycles.

4.2 Cache Design: Area and Access Time Issues

The access time and area occupied by an SRAM cache depends on several factors such as associativity, cache size, number of wordlines, and number of bitlines [27], [18]. In this section, we study the impact of cache size and associativity on access time and area constraints. Our aim is to find the appropriate design parameters for our crossbar switch cache.

Cache Access Time: The CACTI model [27] quantifies the relationship between cache size, associativity, and cache access time. We ran the CACTI model to measure the switch cache access time for different cache sizes and set associativity values. In order to use the measurements in a technology independent manner, we present the results using the delay measurement technique known as the *fan-out-of-four* (FO4) [9]. One FO4 is the delay of a signal passing through an inverter driving a load capacitance that is four times larger than its input. It is known that an 8 Kbyte data cache has a cycle time of 25 FO4 [10].

Fig. 11a shows the results obtained in FO4 units. In Fig. 11a, the x-axis denotes the size of the cache and each curve represents the access time for a particular set associativity. We find that direct mapped caches have the lowest access time since a direct indexing method is used to locate the line. However, a direct mapped cache may exhibit poor hit ratios due to mapping conflicts in the switch cache. Set-associative caches can provide improved cache hit ratios, but have a longer cycle time due to a higher data array decode delay. Most current processors employ multi-ported set associative L1 caches operating within a single processor cycle. We have chosen a 2-way set associative design for our base crossbar switch cache to maintain a balance between access time and hit ratio. However, we also study the effect of varied set associativity on switch cache performance.

Cache Relative Area: In order to determine the impact of cache size and set associativity on the area occupied by an on-chip cache, we use the area model proposed by Mulder et al. [18]. The area model incorporates overhead area such as drivers, sense amplifiers, tags, and control logic to compare data buffers of different organizations in a technology independent manner using register-bit equivalent or rbe. One rbe equals the area of a bit storage cell. We used this area model and obtained measurements for different cache sizes and associativities.

Fig. 11b shows the results obtained in relative area. The x-axis denotes the size of the cache and each curve represents different set associativity values. For small cache sizes ranging from 512 bytes to 4KB, we find that the amount of area occupied by the direct mapped cache is much lower than that for an 8-way set associative cache. From the figure, we find that an increase in associativity from 1 to 2 has a lower impact on cache area than an increase from 2 to 4. From this observation, we think that a 2-way set associative cache design would be the most suitable organization in terms of cache area and cache access time, as measured earlier.

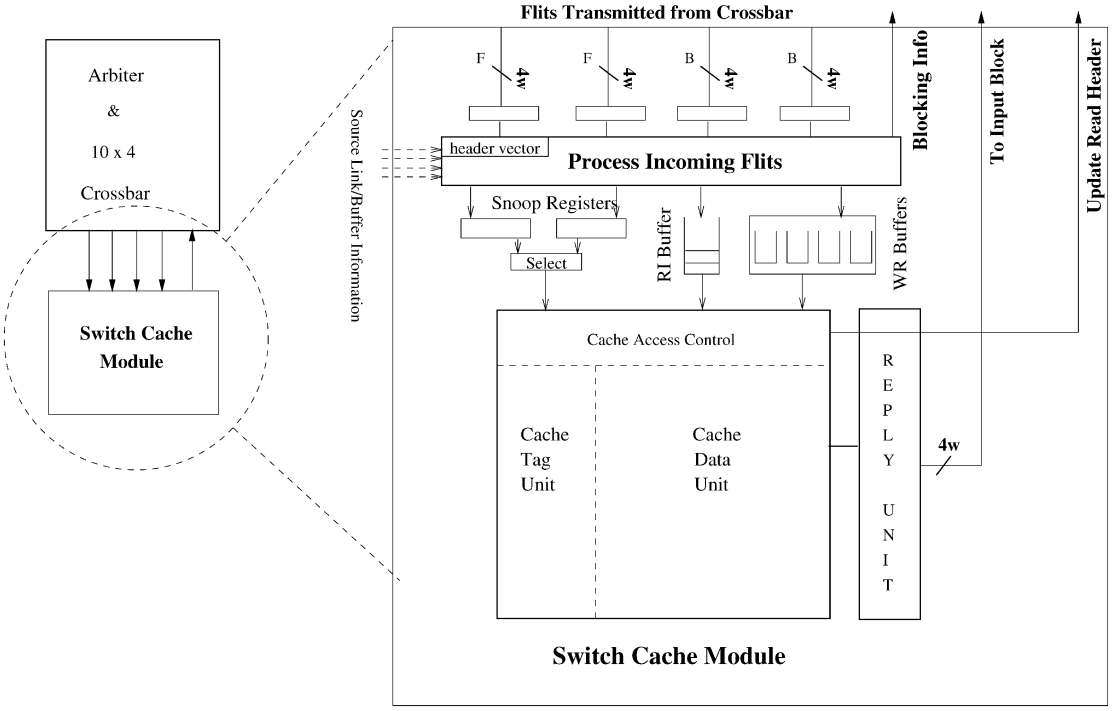


Fig. 12. Implementation of CAESAR.

4.3 CAche Embedded Switch ARchitecture (CAESAR)

In this section, we present a hardware design for our crossbar switch cache called $CAESAR$, (**CA**che **E**mbedded **S**witch **AR**chitecture). A block diagram of the $CAESAR$ implementation is shown in Fig. 12. For a 4×4 switch, a maximum of 4 flits are latched into switch cache registers at each arbitration cycle. The operation of the $CAESAR$ switch cache can be divided into 1) *process incoming flits*, 2) *switch cache access*, 3) *switch cache reply generation*, and 4) *switch cache feedback*. In this section, we cover the design and implementation issues for each of these operations in detail.

Process Incoming Flits: Incoming flits stored into the registers can belong to different request types. The request type of the flit is identified based on the 2 bits (R_1R_0) stored in the header. Header flits of each request contain the relevant information, including memory address required for processing reads and invalidations. Subsequent flits belonging to these messages carry additional information not essential for the switch cache processing. Write requests to the switch cache require both the header flit for address information and the data flits to be written into the cache line. Finally, $ignore$ requests need to be discarded since they do not require any switch cache processing. An additional type of request that does not require processing is the $marked$ read request. This read request has the swc_hit bit set in the header to inform switch caches that it has been served at a previous switch cache. Having classified the types of flits entering the cache, the switch cache processing can be broken into two basic operations.

The first operation performed by the flit processing unit is that of propagating the appropriate flits to the switch cache. As mentioned earlier, the flits that need to enter the cache are read headers, invalidation headers, and all write flits. Thus, the processing unit masks out $ignore$ flits, $marked$ read flits, and the data flits of invalidation and read requests. This is done by reading the R_1R_0 bits from the header vector and the swc_hit bit. To utilize this header information for the subsequent data flits of the message, the switch cache maintains a register that stores these bits.

Flits requiring cache processing are passed to the request queue one in every cycle. The request queue is organized as two buffers, the *RI* buffer and the set of *WR* buffers, shown in Fig. 12. The RI buffer holds the header flits of read and invalidation requests. The WR buffers store all the reply data blocks, to be written into the cache. Since there are a maximum of four replies that can flow through a switch at a time, we organize the WR buffer into four separate units to store them without mixing the packets. When all data flits of a write request have accumulated into a buffer, the request is ready to initiate the cache line fill operation.

The second operation to complete the processing of incoming flits is as follows: All unmarked read header flits need to snoop the cache to gather $hit/miss$ information. This information is needed within the four cycles of switch delay or link transmission to be able mark the header by setting the last bit (swc_hit). To perform this snooping operation on the cache tag, the read headers are also copied to the snoop registers (shown in Fig. 12). We require two snoop registers because a maximum of two read requests can enter the cache in a single arbitration cycle.

Switch Cache Access: Fig. 13 illustrates the design of the cache subsystem. The cache module shown in the figure is that of a 2-way set associative SRAM cache The cache operates at the same frequency as the internal switch core. The set associative cache is organized using two subarrays

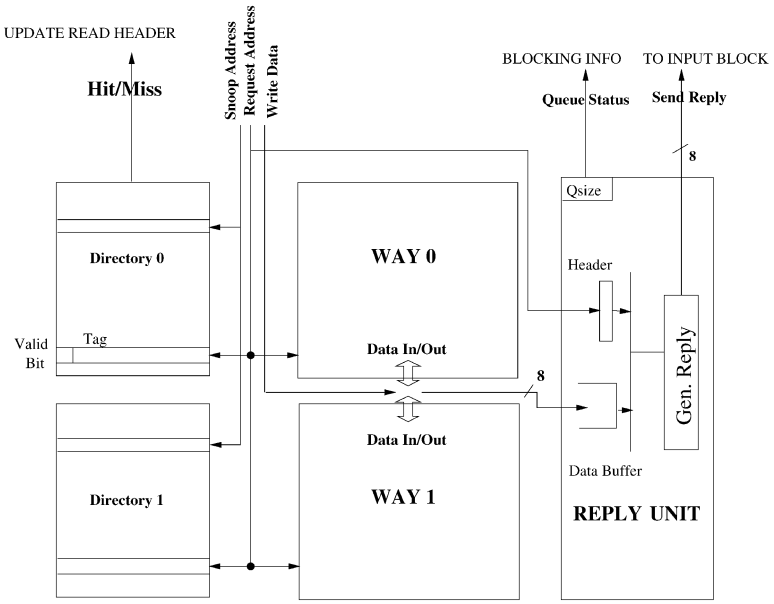


Fig. 13. Design of the $CAESAR$ cache module.

for tag and data. The cache output width is 64 bits, thus requiring four cycles of data transfer for reading a 32 byte cache line. The tag array is dual ported to allow two independent requests to access the tag at the same time. We now describe the switch cache access operations and their associated access delays. Requests to the switch cache can be broken into two types of requests: *snoop requests* and *regular requests*.

Snoop Requests: Read requests are required to snoop the cache to determine hit or miss before the outgoing flit is transmitted to the next switch. For the arbitration independent switch cache organization (Fig. 10a), it takes a minimum of four cycles for moving the flit from the switch input to the output. Thus, we need the snoop operation within the last cycle to mark the message before link transmission. Similarly, for the arbitration dependent organization (Fig. 10b), it takes four cycles to transmit a 64-bit header on a 16-bit output link after the header is loaded into the 64-bit (4w) output register. From the message format in Fig. 8, the phit containing the swc_hit bit to be marked is transmitted in the fourth cycle. Thus, it is required that the cache access be completed within a maximum of three cycles. From Fig. 12, copying the first read to the snoop registers is performed by the flit processing unit and is completed in one cycle. By dedicating one of the ports of the tag array primarily for snoop requests, each snoop in the cache takes only an additional cycle to complete. Since a maximum of two read headers can arrive to the switch cache in a single arbitration cycle, we can complete the snoop operation in the cache within three cycles. Note from Fig. 13 that the snoop operation is done in parallel with the pending requests in the RI buffer and the WR buffers. When the snoop operation completes, the hit/miss information is propagated to the output transmitter to update the read header in the output register. If the snoop operation results in a switch cache miss, the request is also dequeued from the RI buffer.

Regular Requests: A regular request is a request chosen from either the RI buffer or the WR buffers. Such a request is processed in a maximum of four cycles in the absence of contention. Requests from the RI buffer are handled on an $FCFS$ basis. This avoids any dependency violation between read and invalidation requests in that buffer. However, we can have a candidate for cache operation from the RI buffer as well as from one or more of the WR buffers. In the absence of address dependencies, the requests from these buffers can progress in any order to the switch cache. When a dependency exists between two requests, we need to make sure that cache state correctness is preserved. We identify two types of dependencies between a request from the RI buffer and a request from the WR buffer:

- An invalidation (from the RI buffer) to a cache line X and a write (from the the WR buffer) to the same cache line X can occur simultaneously. To preserve consistency, the simplest method is to discard the write to the cache line, thus avoiding incorrectness in the cache state. Thus, when invalidations enter the switch cache, write addresses of pending write requests in the WR buffer are compared and invalidated in parallel with the cache line invalidation.
- A read (from the RI buffer) to a cache line X and a write (from the WR buffer) to a cache line Y that map on to the same cache entry can also occur. If the write occurs first, then cache line X will be replaced. In such an event, the read request cannot be served. Since such an occurrence is rare, the remedy is to send the read request back to the home node destined to be satisfied as a typical remote memory read request.

Switch Cache Reply Generation: While invalidations and writes to the cache do not generate any replies from the switch cache, read requests need to be serviced by reading

the cache line from the cache and sending the reply message to the requesting processor. The read header contains all the information required to send out the reply to the requester. The read header and cache line data are directly fed into the reply unit shown in Fig. 13. The reply unit gathers the header at the beginning of the cache access and modifies the source/destination and request/reply information in the header in parallel with the cache access. When the entire cache line has been read, the reply packet is generated and sent to switch cache output block. The reply message from the switch cache now acts as any other message entering a switch in the form of flits and gets arbitrated to the appropriate output link and progresses using the backward path to the requesting processor.

Switch Cache Feedback: Another design issue for the CAESAR switch is the selection of queue sizes. In this section, we identify methods to preserve crossbar switch throughput by blocking only those requests that violate correctness. As shown in Figs. 12 and 13, finite size queues exist at the input of the switch cache (RI buffer and WR buffer) and at the reply unit (virtual channel queues in the switch cache output block). When any limited size buffer gets full, we have two options for the processing of read/write requests. The first option is to block the requests until a space is available in the buffer. The second option, probably the wiser one, is to allow the request to continue on its path to memory. The performance of the switch cache will be dependent on the chosen scheme only when buffer sizes are extremely limited. Finally, invalidate messages have to be processed through the switch cache since they are required to maintain coherence. These messages need to be blocked only when the RI buffer gets full. The modification to the arbiter required to make this possible is quite simple. To implement the blocking of flits at the input, the switch cache needs to inform the arbiter of the status of all its queues. At the end of each cycle, the switch cache informs the crossbar about the status of its queues in the form of $free\_space$ available in each queue. The modification to the arbiter to perform the required blocking is minor. Depending on the $free\_space$ of each queue, appropriate requests (based on R_1R_0) will be blocked while others will traverse through the switch in a normal fashion.

4.4 Design of a 8×8 Crossbar Switch Cache

In the previous sections, we presented the design and implementation of a 4×4 cache embedded crossbar switch. Current commercial switches, such as SGI Spider and Intel Cavallino, have six bidirectional inputs, while the IBM SP2 switch has eight inputs and eight outputs. In this section, we present extensions to the CAESAR design to incorporate a switch cache module in a 8×8 switch. We maintain the same base parameters depending on switch type (see Fig. 10) for switch core frequency (100 MHz, 200 MHz), core delay (four cycles), link speed (400 MHz, 200 MHz), link width (2 bytes), and flit size (8 bytes).

The main issue when expanding the switch is that the number of inputs to the switch cache module doubles from four to eight, requiring a 16×8 crossbar. Thus, in each arbitration cycle, a maximum of four read requests can enter the switch. These requests require snoop operation on the switch cache within four cycles of switch core delay or link transmission, depending on the switch cache organization shown in Fig. 10. As shown in Fig. 12, it takes one cycle to move the flits to the snoop registers. Thus, it is required that the snoop operation for four requests completes within two cycles to mark the header flit in the last cycle, depending on the snoop result.

In order to perform four snoops in two cycles, we propose using a multiported CAESAR cache module. Multiporting can be implemented either by duplicating the caches or interleaving it into two independent banks. Since duplicating the cache consumes a tremendous amount of on-chip area, we propose using a 2-way interleaved switch cache called $CAESAR^+$. The cache organization of $CAESAR^+$ is shown in Fig. 14. Interleaving splits the cache organization into two independent banks. Current processors such as MIPS R10000 [27] use even and odd addressed banked caches. However, the problem remains that four simultaneous even addressed or four simultaneous odd addressed requests will still require four cycles for snooping due to bank conflicts. We propose interleaving the banks based on the destination memory by using the outgoing link *ids*. In an 8×8 switch there are four outgoing links that transmit flits from the switch towards the destination memory and vice versa. Each cache bank will serve requests flowing through two of these links, thus partitioning the requests based on the destination memory. In an arbitration-independent organization (Fig. 10a), it is possible that four incoming read requests are directed to the same memory module and thus result in link conflicts and bank conflicts. However, in the arbitration-dependent organization (Fig. 10b), link conflicts get resolved during the arbitration phase. This guarantees that the arbitrated flits flow through different links. Since each bank serves only flits flowing through two links, a maximum of only two requests can target a single cache bank in an arbitration period of four cycles. Thus, for 8×8 switches, it would be more advantageous to use an arbitration dependent organization where the snoop operation of four requests can be completed in the required two cycles. Finally, note that only a few bits from the routing tag are needed to identify the bank in the cache.

Such an interleaved organization changes the aspect ratio of the cache [27] and may affect the cycle time of the cache. Wilson and Olukotun [26] showed that the increase in cycle time measured using the fan-out-of-four (FO4) [10] for banked or interleaved caches over single ported caches was minimal. Since two requests can simultaneously access the switch cache, the reply unit needs to provide twice the buffer space for storing the data from the cache. Similarly, the header flit of the two read requests also needs to be stored. As shown in Fig. 14, the buffers are connected to outputs from different banks to gather the cache line data.

5 Performance Evaluation

In this section, we present a detailed performance evaluation of 8×8 switch caches employed in a CC-NUMA multiprocessor. A preliminary evaluation of 4×4 switch caches was presented in [11]. In this paper, we present base comparisons as well as detailed sensitivity studies of cache size, line size, associativity,

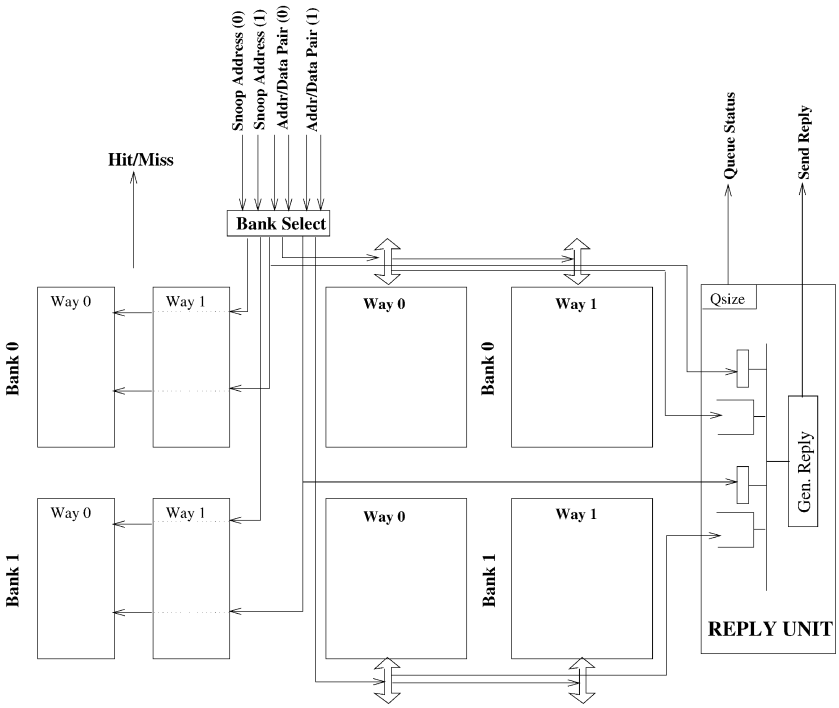


Fig. 14. Design of the $CAESAR^{+}$ cache module.

and application size. Our results are based on extensive execution-driven simulation experiments.

5.1 Simulation Methodology

To evaluate the performance impact of switch caches on the application performance of CC-NUMA multiprocessors, we use a modified version of the Rice Simulator for ILP Multiprocessors (RSIM) [21]. RSIM is an execution driven simulator for shared memory multiprocessors with accurate models of current processors that exploit instruction-level parallelism. In this section, we present the various system configurations and the corresponding modifications to RSIM for conducting simulation runs.

The base system configuration consists of 16 nodes. Each node consists of a 200MHz processor capable of issuing four instructions per cycle, a 16KB L1 cache, a 128KB L2 cache, a portion of the local memory, directory storage, and a local bus interconnecting these components. The processor microarchitecture in RSIM is based on the MIPS R10000. Latency hiding techniques such as hardware prefetching are not enabled. The L1 cache is 2-way set associative with an access time of a single cycle. The L2 cache is 4-way set associative and has an access time of eight cycles. The raw memory access time is 40 cycles, but it takes more than 50 cycles to submit the request to the memory subsystem and read the data over the memory bus. The system employs the full-map three-state directory protocol [7] and the MSI cache protocol to maintain cache coherence. The system uses a release consistency model. We modified RSIM to employ a wormhole routed bidirectional MIN using 8×8 switches organized in two stages, as shown earlier in Fig. 6. Virtual channels were also added to the switching elements to simulate the behavior of commercial switches like Cavallino and Spider. Each input link to the switch is provided with two virtual channel buffers capable of storing a maximum of four flits from a single message. The crossbar switch operation is similar to the description in Section 4.1. A detailed list of simulation parameters is also shown in Table 1.

To evaluate switch caches, we further modified the simulator to incorporate switch caches into each switching element in the IN. The switch cache system improves on the base system in the following respects. Each switching element of the bidirectional MIN employs a variable size cache that models the functionality of the CAESAR switch

TABLE 1
Simulation Parameters

Multiprocessor System - 16 processors			
Processor		Memory	
Speed	200MHz	Access time	40
Issue	4-way	Interleaving	4
Cache		Network	
L1 Cache	16KB	Switch Size	8x8
line size	32bytes	Core delay	4
set size	2	Core Freq	200MHz
access time	1	Link width	16 bits
L2 Cache	128KB	Xfer Freq	200MHz
line	32bytes	Flit length	8bytes
set size	4	Virtual Chs.	2
access time	8	Buf. Length	4 flits
Switch/Network Caches			
Switch Cache	128bytes-8KB	Network Cache	4KB
Application Workload			
FWA	128x128	GE	128x128
GS	96x128	MM	128x128
SOR	512x512	FFT	$16K$ pts

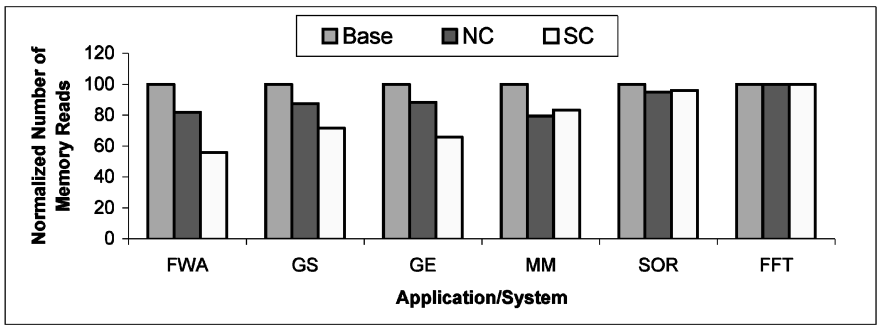


Fig. 15. Percentage reduction in memory reads.

cache presented in Section 4. Several parameters such as cache size and set associativity are varied for evaluating the design space of the switch cache.

We have selected some numerical applications to investigate the potential performance benefits of the switch cache interconnect. These applications are Floyd-Warshall's all-pair-shortest-path algorithm, Gaussian elimination (GE), QR factorization using the Gram-Schmidt Algorithm (GS), and the multiplication of 2D matrices (MM), successive over-relaxation of a grid (SOR), and the six-step 1D fast Fourier transform (FFT) from SPLASH [22]. The input data sizes are shown in Table 1 and the sharing characteristics were discussed in Section 2.1. Four of the six applications were chosen since they exhibit wide-shring characteristics. The remaining two applications, FFT and SOR, were chosen to show that switch caches do not degrade the performance of applications with different sharing patterns.

5.2 Base Simulation Results

In this subsection, we present and analyze the results obtained through extensive simulation runs to compare three systems: the base system ($Base$), network cache (NC), and switch cache (SC). The $Base$ system does not employ any caching technique beyond the L1 and L2 caches. We simulate a system with NC by enabling $4KB$ switch caches in all the switching elements of stage 0 in the MIN. Note that stage 0 is the stage close to the processor, while stage 1 is the stage close to the remote memory, as shown in Fig. 6. The SC system employs switch caches in all the switching elements of the MIN.

The main purpose of switch caches in the interconnect is to serve read requests as they traverse to memory. This enhances the performance by reducing the number of read misses served at the remote memory. Fig. 15 presents the reduction in the number of read misses to memory by employing network caches (NC) and switch caches (SC) over the base system (Base). In order to perform a fair comparison, here we compare an SC system with 2KB switch caches at both stages (overall 4KB cache space) to an NC system with 4KB network caches. Fig. 15 shows that network caches reduce remote read misses by 6-20 percent for all the applications, except FFT. The multiple layers of switch caches are capable of reducing the number of memory read requests by up to 45 percent for FWA, GS, and GE applications.

Table 2 shows the distribution of switch cache hits across the two stages (St0 and St1) of the network. From the table, we note that a high percentage of requests get satisfied in the switch caches present at the first stage (St0) in the interconnect. Note, however, that for three of the six applications, roughly 30-40 percent of the requests are switch cache hits in the stage close to the memory (St1). It is also interesting to note the number of requests satisfied by storing each block in the switch cache. Table 2 presents this data as $sharing$, which is defined as the number of different processor requests served for a block, encached in the switch cache. We find that this sharing degree ranges from 1.0 to 2.7 across all applications. For applications with high overall read sharing degrees (FWA, GS, and GE), the degree of sharing is approximately 1.7 in the stage closer to the processor. With only four of 16 processors connected to each switch, many read requests do not find the block in the first stage, but get satisfied in the second stage. Thus, we find a higher (approximately 2.5) read sharing degree at St1 for these applications. The MM application has an overall sharing degree of approximately 4 (see Fig. 2). The data is typically shared by four processors physically connected to the same switch in the first stage of the network. Thus, most of the requests (88.2 percent) get satisfied in the first stage and attain a read sharing degree of 1.8. Finally, the SOR and FFT applications have very few read shared requests, most of which are satisfied in the first stage of the network.

Fig. 16 shows the improvement in average memory access latency for reads for each application by using switch caching in the interconnect. For each application, the figure consists of three bars corresponding to the Base, NC, and SC systems. The average read latency is comprised of processor cache delay, bus delay, network data transfer delay, memory service time, and queuing delays at the network and memory module. As shown in the figure, by employing

TABLE 2
Distribution of Switch Cache Accesses

Appl	Hit Distribution		Sharing	
	St0	St1	St0	St1
FWA	67.21	32.79	1.79	2.49
GS	69.02	30.98	1.94	2.37
GE	59.55	40.45	1.58	2.66
MM	88.20	11.80	1.80	1.08
SOR	94.47	5.53	1.32	1.0
FFT	60.59	39.41	1.95	1.96

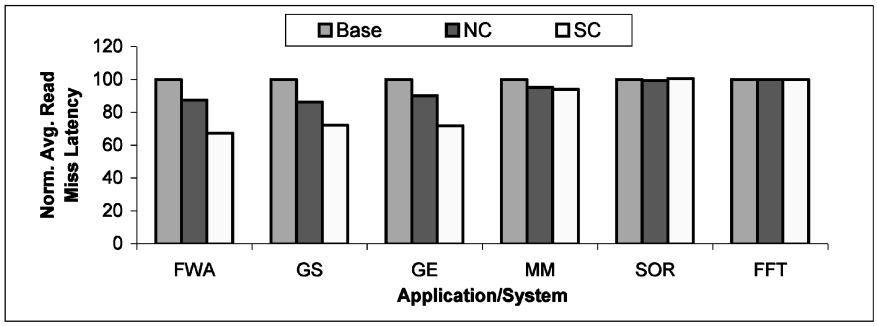


Fig. 16. Impact on average read latency.

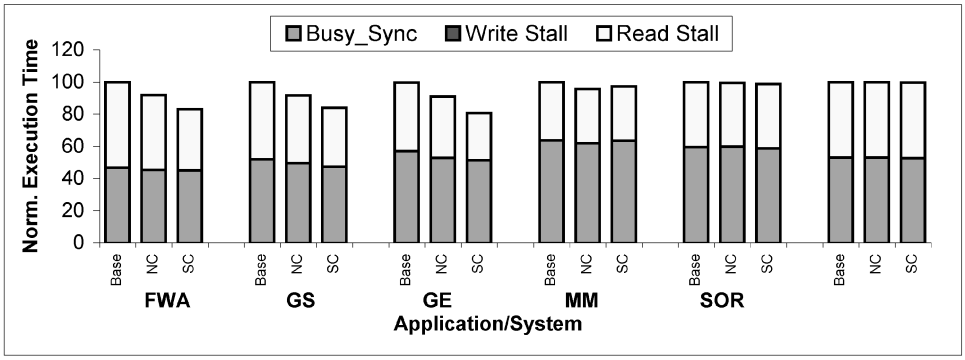


Fig. 17. Application execution time improvements.

network caches, we can improve the average read latency by atmost 15 percent for most of the applications. With switch caches in multiple stages of the interconnect, we find that the average read latency can be improved by as high as 35 percent for FWA, GS, and GE applications. The read latency reduces by about 7 percent for the MM application. Again, SOR and FFT are unaffected by network caches or switch caches due to negligible read sharing.

The ultimate parameter for performance is execution time. Fig. 17 shows the execution time improvement. Each bar in the figure is divided into computation and synchronization time, read stall time, and write stall time. In a release consistent system, we find that the write stall time is negligible. However, the read stall time in the base system comprises as high as 50 percent of the overall execution time. Using network caches, the read stall time reduces by a maximum of 20 percent (for the FWA, GS, and GE applications) and thus translates to an improvement in execution time by up to 10 percent. Using switch caches, we observe execution time improvements as high as 20 percent in the same three applications. The execution time of the MM application is comparable to that with network caches. SOR and FFT are unaffected by either network or switch caches.

5.3 Sensitivity Studies

5.3.1 Sensitivity to Cache Size

In order to determine the effect of cache size on performance, we varied the switch cache size from a mere 128 bytes to a large $8KB$. Figs. 18 and 19 show the impact of switch cache size on the number of memory reads and the overall execution time. As the cache size is increased, we see that a switch cache size of 512 bytes provides the maximum performance improvement (up to 45 percent reads and 20 percent execution time) for three of the six applications. The MM and SOR applications require larger caches for additional improvement. The MM application attains a performance improvement of 7 percent in execution time at a switch cache size of 2KB. Increasing the cache size further has negligible impact on performance. For SOR, we found some reduction in the number of memory reads, contrary to the negligible amount of sharing in the application (shown in Fig. 2). Upon investigation, we found that the switch cache hits come from replacements in the L2 caches. In other words, blocks in the switch cache are accessed highly by the same processor whose initial request entered the block into the switch cache. The switch cache acts as a victim cache for this application. The use of switch caches does not affect the performance of the FFT application.

Fig. 20 investigates the impact of cache size on the eviction rate and type in the switch cache for the FWA application. The x-axis in the figure represents the size of the cache in bytes. A block in the switch cache can be evicted either due to replacement or due to invalidation. Each bar in the figure is divided into two portions to represent the amount of replacements versus invalidations in the switch cache. The figures are normalized to the number of evictions for a system with 128 byte switch caches. The first observation from the figure is the reduction in the number of evictions as the cache size increases. Note that the number of evictions remains constant beyond a cache size of 1KB. With small caches, we also observe that roughly 10-20 percent of the blocks in the switch cache are invalidated, while all others are replaced. In other words, for most blocks, invalidations are not processed through the switch cache since they have already been evicted through

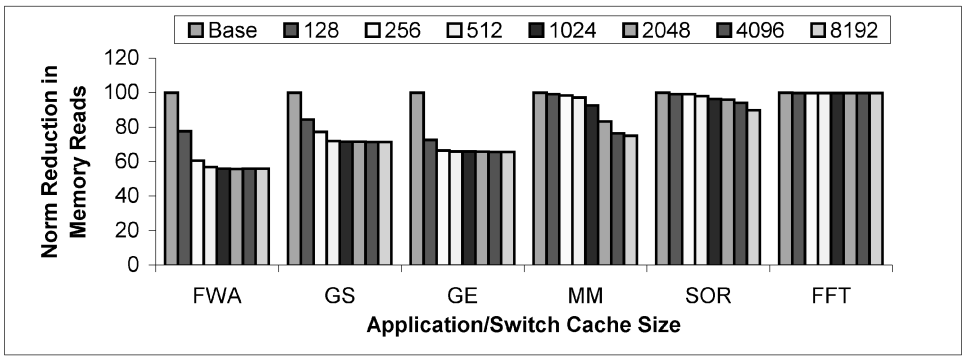


Fig. 18. Impact of cache size on the number of memory reads.

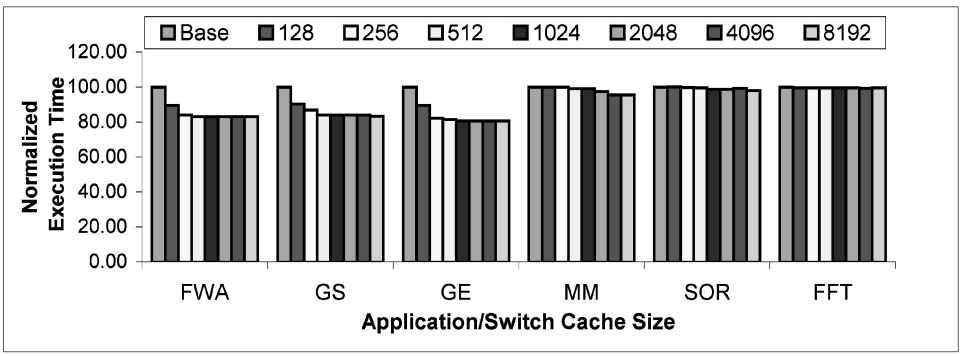


Fig. 19. Impact of cache size on execution time.

replacements due to small capacity. As the cache size increases, we find the fraction of invalidations increases since fewer replacements occur in larger caches. For the 8KB switch cache, we find that roughly 50 percent of the blocks are invalidated from the cache.

We next look at the impact of cache size on the amount of sharing across stages. Fig. 21 shows the amount of hits obtained in each stage of the network for the FWA application. Each bar is divided into two segments, representing each stage of switch caches, denoted by the stage number. Note that $Stage0$ is the stage closest to the processor interface. From the figure, it is interesting to note that, for small caches, an equal amount of hits are obtained from each stage in the network. On the other hand, as the cache size increases, we find that a higher fraction of the hits are due to switch caches closer to the processor interface (60-70 percent from $St0$). This is beneficial because fewer hops are required in the network to access the data, thereby reducing the read latency considerably.

5.3.2 Sensitivity to Cache Line Size

In the earlier sections, we analyzed data with 32 byte cache lines. In this section, we vary the cache line size to study its impact on switch cache performance. Figs. 22 and 23 show the impact of a larger cache line (64 bytes) on the switch cache performance for three applications (FWA, GS, and GE). We vary the cache size from 256 bytes to 16KB and compare the performance to the base system with 32 byte cache lines and 64 byte cache lines. Note that the results are normalized to the base system with 64 byte cache lines. We found that the number of memory reads were reduced by 37 to 45 percent when we increase the cache line size in the base system. However, the use of switch caches still has significant impact on application performance. With 1KB switch caches, we can reduce the number of read requests served at the remote memory by as much as 45 percent and the execution time by as much as 20 percent. In summary, the switch cache performance does not depend highly on cache line size for highly read shared applications with good spatial locality.

5.3.3 Sensitivity to Set Associativity

In this section, we study the impact of cache set associativity on application performance. Fig. 24 shows the percentage of switch cache hits as the cache size and associativity are varied. We find that set associativity has no impact on switch cache performance. Frequently accessed blocks need to reside in the switch cache only for a short amount of time, as we observed earlier from our trace analysis. A higher degree of associativity tries to prolong the residence time by reducing cache conflicts. Since we do not require a higher residence time in the switch cache, the performance is neither improved nor hindered.

5.3.4 Sensitivity to Application Size

Another concern for the performance of switch caches is the relatively small data set that we have used for faster simulation. In order to verify that the switch cache performance does not change drastically for larger data sets, we used the FWA application and increased the number of vertices from 128 to 192 and 256. The results obtained from these simulations are shown in Fig. 25. Note that the data set size increases by a square of the number of vertices. The base system execution time increases by a factor of 2.3 and 4.6,

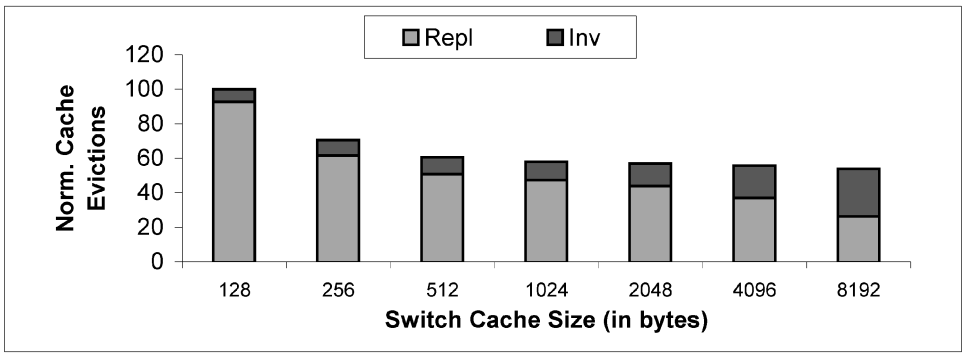


Fig. 20. Effect of cache size on the eviction rate.

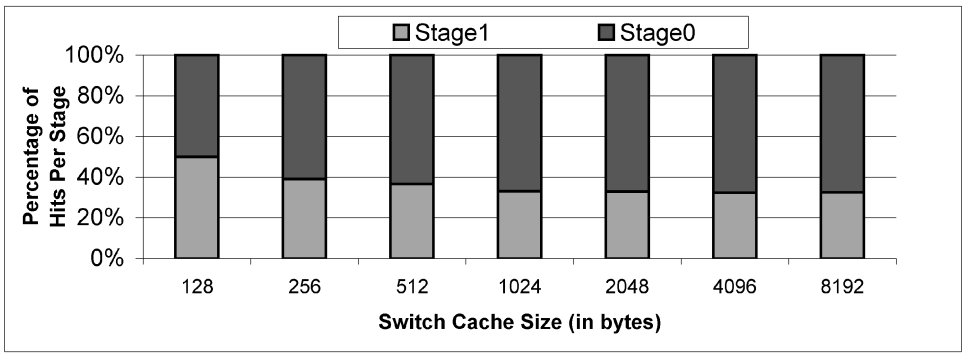


Fig. 21. Effect of cache size on switch cache hits across stages.

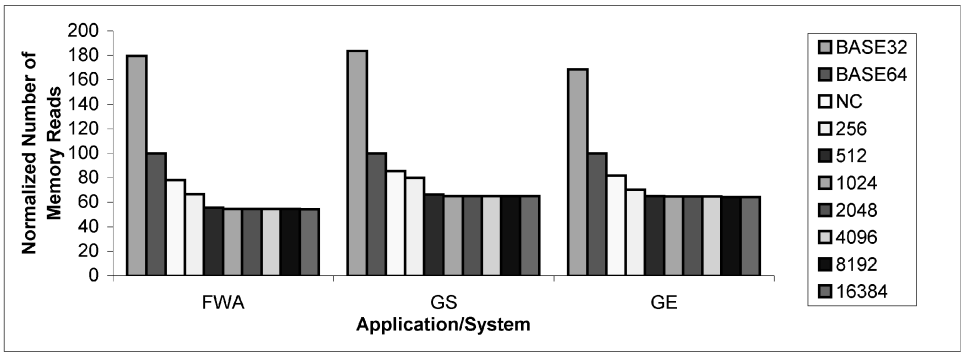


Fig. 22. Effect of line size on the number of memory reads.

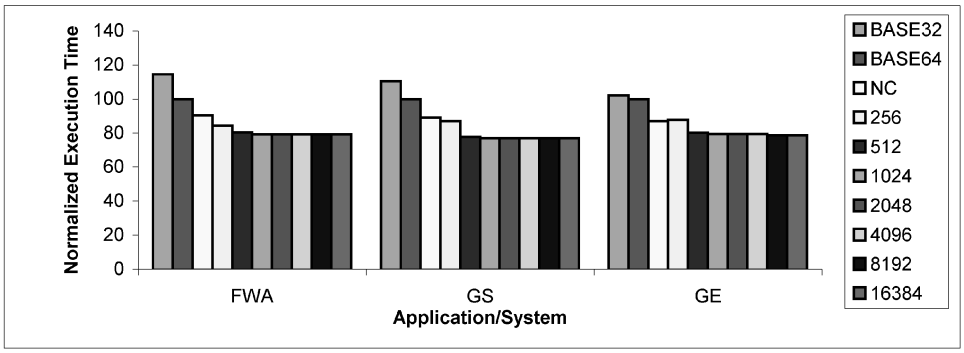


Fig. 23. Effect of line size on execution time.

respectively. With 512 byte switch caches, the execution time reduces by 17 percent for 128 vertices, 13 percent for 192 vertices, and 10 percent for 256 vertices. In summary, we believe that switch caches require small cache capacity and can provide sufficient performance improvements for large applications with frequently accessed read shared data.

6 Conclusions

In this paper, we presented a novel hardware caching technique, called switch cache, to improve the remote memory access performance of CC-NUMA multiprocessors. A detailed trace analysis of several applications

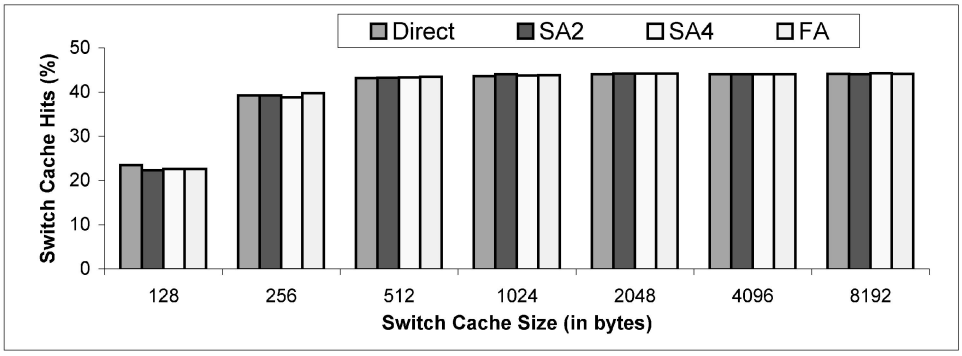


Fig. 24. Effect of associativity on switch cache hits.

showed that accesses to shared blocks have a great deal of temporal locality. Thus, remote memory access performance can be greatly improved by caching shared data in a global cache. To make the global cache accessible to all the processors in the system, the interconnect seems to be the best location since it has the ability to monitor all internode transactions in the system in an efficient, yet distributed fashion.

By incorporating small caches within each switching element of the MIN, shared data was captured as they flowed from the memory to the processor. In designing such a switch caching framework, several issues were dealt with. The main hindrance to global caching techniques is that of maintaining cache coherence. We organized the caching technique in a hierarchical fashion by utilizing the inherent tree structure of the BMIN and sending invalidations from the home node along the tree. The read requests that hit in the switch cache were marked and allowed to continue on their path to the memory for the sole purpose of updating the directory. The caching technique was also kept noninclusive and, thus, devoid of the size problem in a multilevel inclusion property.

The most important issue while designing switch caches was that of incorporating a cache within typical crossbar switches (such as SPIDER and CAVALLINO) in a manner such that requests are not delayed at the switching elements. A detailed design of a cache embedded switch architecture (CAESAR) was presented and analyzed. The size and organization of the cache depends heavily on the switch transmission latency. We presented a dual-ported 2-way set associative SRAM cache organization for a 4×4 crossbar switch cache. We also proposed a link-based interleaved cache organization to scale the size of the CAESAR module for 8×8 crossbar switches. Our simulation results indicate that a small cache of size 1 KB bytes is sufficient to provide up to 45 percent reduction in memory service and, thus, a 20 percent improvement in execution time for some applications. This relates to the fact that applications have a lot of temporal locality in their shared accesses. Current switches such as SPIDER maintain large buffers that are underutilized in shared memory multiprocessors. It seems that, by organizing these buffers as a switch cache, more improvement in performance can be realized.

In this paper, we studied the use of switch caches to store recently accessed data in the shared state to be reused by subsequent requests from any processor in the system. In addition to these requests, applications also have a significant amount of accesses to blocks in the dirty state. To improve the performance of such requests, directories have to be embedded within the switching elements. By providing shared data through switch caches and ownership information through switch directories, the performance of the CC-NUMA multiprocessor can be improved. Latency hiding techniques such as data prefetching or forwarding can also utilize the switch cache and reduce the risk of processor cache pollution. The use of switch caches along with the above latency hiding techniques can further improve the application performance on CC-NUMA multiprocessors. A recent paper [3] on wide area network (WAN) caches suggests that a variant of switch caches can also help in reducing the response time to frequently accessed world wide web objects.

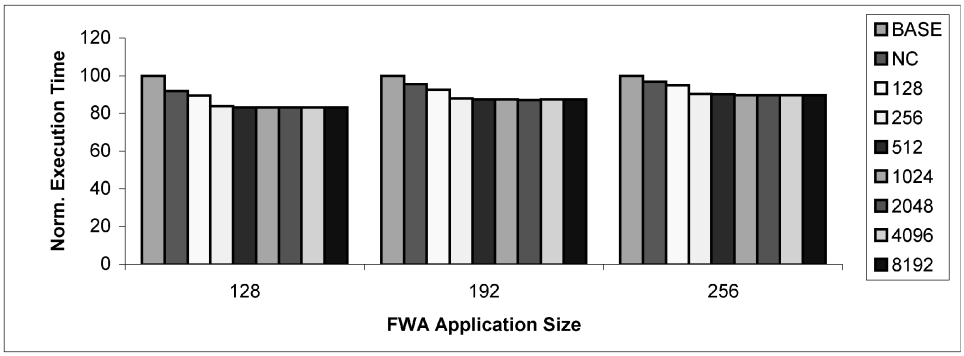


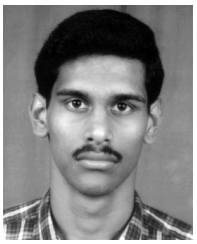
Fig. 25. Effect of application size on execution time.

ACKNOWLEDGMENTS

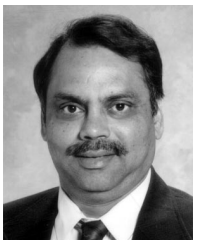
This research has been supported by US National Science Foundation grants MIP-9622740 and CCR-9810205.

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, accessed in Dec. 1997.
[2] BBN Laboratories Inc., "Butterfly Parallel Processor Overview, version 1,"Cambridge, Mass., Dec. 1985.
[3] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura, "Self-Organizing Wide-Area Network Caches," *Proc. IEEE Infocom '98,* Mar. 1998.
[4] L. Bhuyan, R. Iyer, T. Askar, A. Nanda, and M. Kumar, "Performance of the Multistage Bus Networks for a Distributed Shared Memory Multiprocessor," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 1, pp. 82-95, Jan. 1997.
[5] L. Bhuyan, H. Wang, R. Iyer, and A. Kumar, "The Impact of Switch Design on the Application Performance of Shared Memory Multiprocessors," *Proc. Int'l Parallel Processing Symp.,* pp. 466-474, Mar. 1998.
[6] J. Carbonaro and F. Verhoorn, "Cavallino: The Teraflops Router and NIC," *Proc. Symp. High Performance Interconnects (Hot Interconnects 4),* Aug. 1996.
[7] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers,* vol. 27, no. 12, pp. 1,112-1,118, Dec. 1978.
[8] M. Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip," *Proc. Symp. High Performance Interconnects (Hot Interconnects 4),* Aug. 1996.
[9] M. Horowitz, S. Przybylski, and M. Smith, "Tutorial on Recent Trends in Processor Design: Reclimbing the Complexity Curve," Stanford, Calif.: Western Inst. of Computer Science, Stanford Univ., 1992.
[10] M. Horowitz, "High Frequency Clock Distribution," *Proc. 1996 Symp. VLSI Circuits,* June 1996.
[11] R. Iyer and L.N. Bhuyan, "Switch Cache: A Framework for Improving the Remote Memory Access Latency of CC-NUMA Multiprocessors," *Proc. Fifth Int'l Conf. High Performance Computer Architecture (HPCA-5),* pp. 152-160, Jan. 1999.
[12] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 241-251, 1997.
[13] C.E. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi et al., "The Network Architecture of the Connection Machine," *J. Parallel and Distributed Computing,* vol. 33, no. 2, pp. 145-158, Mar. 1996.
[14] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta et al., "The Stanford DASH Multiprocessor," *Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.
[15] 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.
[16] 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.
[17] P. Mohapatra and C.R. Das, "A Performance Model for Finite-Buffered Multistage Interconnection Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 1, pp. 18-25, Jan. 1996.
[18] J. Mulder, N. Quach, and M. Flynn, "An Area Model for On-Chip Memories and Its Application," *IEEE J. Solid State Circuits,* vol. 26, no. 2, pp. 98-106, Feb. 1991.
[19] A. Nanda and L. Bhuyan, "Design and Analysis of Cache Coherent Multistage Interconnection Networks," *IEEE Trans. Computers,* vol. 42, no. 4, Apr. 1993.
[20] B. Nayfeh, K. Olukotun, and J.P. Singh, "The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors," *Proc. Second Int'l Symp. High Performance Computer Architecture,* pp. 74-84, Feb. 1996.
[21] V. Pai, P. Ranganathan, and S.V. Adve, "RSIM Reference Manual, Version 1.0," Technical Report 9705, Dept. of Electrical and Computer Eng., Rice Univ., July 1997.
[22] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *ACM SIGARCH Computer Architecture News,* vol. 20, no. 1, pp. 5-44, Mar. 1992.
[23] C.B. Stunkel, D. Shea, B. Abali, M. Atkins, C. Bender et al., "The SP2 High Performance Switch," *IBM Systems J.,* vol. 34, no. 2, pp. 185-204, 1995.
[24] J. Torrellas and Z. Zhang, "The Performance of the Cedar Multistage Switching Network," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 4, pp. 321-336, Apr. 1994.
[25] A.W. Wilson, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. 14th Ann. Int'l. Symp. Computer Architecture,* pp. 244-252, 1987.
[26] K. Wilson and K. Olukotun, "Designing High Bandwidth On-Chip Caches," *Proc. 23rd Int'l Symp. Computer Achitecture,* pp. 121-132, 1997.
[27] S. Wilton and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," Technical Report no. 93/5, DEC-Western Research Lab, 1994.
[28] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.
[29] Z. Zhang and J. Torellas, "Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA," *Proc. Third Int'l Symp. High Performance Computer Architecture,* pp. 272-281, Jan. 1997.



Ravishankar R. Iyer received his PhD in computer science, MS in computer science, and BS in electrical engineering in August 1999, August 1996, and December 1994, respectively, from Texas A&M University, College Station, Texas. He is currently working in the Server Architecture Laboratory at Intel Corporation. He previously worked in a chipset architecture group at Intel Corporation and in a memory technology group at Hewlett-Packard Laboratories. His research interests include computer architecture, parallel computing and performance evaluation. He is a member of the IEEE.



Laxmi N. Bhuyan (S'81-M'82-SM'87-F'98) 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. Currently, he is a professor of computer science at Texas A&M University, College Station, Teaxas. 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 the *Computer* magazine, the *Journal of Parallel and Distributed Computing* (JPDC), *IEEE Transactions on Parallel and Distributed Systems*, and *Parallel Computing* journal. 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 ACM and the IEEE.