Impact of Switch Design on the Application Performance of Cache-Coherent Multiprocessors *

L. Bhuyan, H. Wang, and R.Iyer
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112, USA
{bhuyan, hwang, ravi}@cs.tamu.edu

A. Kumar
Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95052-8119, USA
akumar2@mipos2.intel.com

Abstract

In this paper, the effect of switch design on the application performance of cache-coherent non-uniform memory access (CC-NUMA) multiprocessors is studied in detail. Wormhole routing and cut-through switching are evaluated for these shared-memory multiprocessors that employ multistage interconnection network (MIN) and full map directory-based cache coherence protocol. The switch design also considers virtual channels and varying number of input buffers per switch. Based on this, four different switch architectures are presented and compared. The evaluation is based on execution-driven simulation using five different applications to capture the random bursty nature of the network traffic arrival. The round-robin memory management policy is implemented. We show that the use of cut-through switching with buffers and virtual channels improves the average message latency tremendously. The waiting times of messages at various stages of switches are also presented. Finally, we show the variation of stall times and execution times for these applications by varying the switch delay and wire width.

1. Introduction

Shared-memory multiprocessors (SMPs) provide a unified view of the memory for easy programming. With the rapid increase in processor speed, parallel computation in shared memory systems benefits tremendously both for scientific and commercial applications. As a result, almost all the computer manufacturers have started marketing SMP servers. While the small SMP servers are based on shared-bus technology, medium scale multiprocessors with more than 16 processors employ sophisticated communication switches in a cache coherent non-uniform memory access (CC-NUMA) environment. Examples include SGI Origin, Convex Exemplar, and Sequent NUMA-Q machines. Design issues for faster communication time in shared memory multiprocessors include the efficient use of caches, interconnection networks and memory management.

Performance evaluation of interconnection networks (INs) have been an active area of research for a long time. Most studies consider the network in isolation and are based on analytical models that assume simple traffic situations. These models do not consider different types of messages, bulky requests and invalidation traffic associated with cache coherence and synchronization in a shared memory environment. It is difficult to capture all the details of a network into a simple analytical or simulation model. On the other hand, numerous cache coherence studies of CC-NUMA architectures based on execution-driven simulation do not model the IN switches explicitly, rather they assume a fixed delay in the IN even ignoring the possible interference among various messages. The IN has become quite advanced with the introduction of techniques such as virtual channel [5] and adaptive routing [7]. The effect of all these advances in the INs has to be judged from the dynamic changes during the execution of applications. Execution-based evaluations for mesh interconnection networks have been reported recently [1, 8, 14] to test the effectiveness of virtual channels and adaptive routing. These studies evaluate the multiprocessors at the network level and do not reflect the effect of design details of switch hardware. Also, the measurements are done at the execution time level instead of a detailed study at the switch or IN level.

We consider a multistage interconnection network (MIN) in this paper, similar to the one employed in Butterfly and Cedar multiprocessors. Torrellas and Zhang [13] recently reported detailed performance evaluation of the Cedar multiprocessor network. Besides being a simulation-based evaluation, our work differs from theirs in (a) switching: we adopt wormhole routing, virtual cut-through

*This research has been supported by NSF grant MIP 9622740.

switching and virtual channels, as employed in commercial multiprocessor switches [3, 6] instead of packet switching in Cedar, (b) cache coherence: we employ a directory-based cache coherence whereas Cedar has no hardware support for cache coherence and (c) organization: Cedar is a UMA system where memory modules are centrally located and employ a forward network from processors to memories and a backward network from memories to processors. We employ a NUMA organization with one network, like Butterfly, that is used for both forward and backward (reply) messages. NUMA architectures offer good scalability for shared memory processing and a number of such machines are commercially available now.

Cache memories are an integral part of shared memory systems to help avoid the large latency of memory accesses. The time to service a cache miss especially from a remote memory in a large system could be several orders of magnitude higher than the cache access time. Even with low miss rates, the bottleneck in performance of the NUMA systems remains the remote memory access times on cache misses. The presence of caches requires support for maintaining coherence among copies of data present in the several caches and memory. Cache coherence protocols can be divided into invalidation-based or update-based. In this paper, we use an invalidation-based directory cache coherence protocol and try to analyze the overhead it introduces with respect to different switch designs.

Our aim in this paper is to re-evaluate the impact of the interconnection network switch design in a CC-NUMA environment. The paper makes the following contributions.

- We present four different switch design alternatives for the multistage interconnection network in cache-coherent shared memory systems, called *simple wormhole (SWH), buffered cut-through (BCT), simple virtual channel (SVC)* and *buffered virtual channel (BVC)* switches. They differ in their buffer sizes and presence of virtual channels.
- We incorporate exact models of the above switches into a detailed application-driven simulation to analyze their performance impact on cache coherent NUMA architectures. Our work enables us to vary system parameters to simulate and analyze designs of realistic systems with five different applications.
- We measure and analyze the execution with varied parameters at three different levels.
  - *IN level:* request/data transfer time and waiting delays, a stage to stage comparison of delays and average message latencies
  - *System or Protocol level:* read/write network latency, invalidation overhead, memory service time

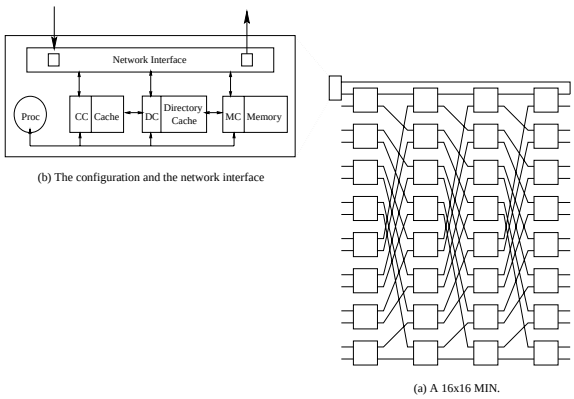

Figure 1. A MIN-connected multiprocessor

  - *Application level:* communication stall time and overall execution time

The rest of the paper is organized as follows. Section 2 presents the MIN architecture and various switch designs used in the evaluation. Section 3 describes the simulation environment, hardware parameters and application characteristics. Section 4 analyzes the waiting delay at various stages and studies the effects of the switch designs. Section 5 presents various stall and execution times as a function of switch delay and link width. Finally, Section 6 presents the conclusion and direction for further work in this area.

2. Switch design alternatives

The architectures under consideration can be modeled as processor, cache, memory, network interface and the network, as shown in Fig. 1 for a 16 processor system. This is a non-uniform memory access (NUMA) architecture where the memory is 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 the message into packets or flits, initializing header flit of every packet with necessary routing information, etc. It contains a coherence controller, memory controller, directory controller and a reply controller, as described in [9].

The schematic of the network is shown in Fig. 1. It is a multistage interconnection network (MIN) employing 2x2 switches. In general, an NxN MIN consists of $\log_2 N$ stages of 2x2 switches with N/2 such switches per stage. The interconnection between the stages is perfect shuffle. We use wormhole routing and virtual cut-through switching techniques. In wormhole routing, the messages are divided into small flits, typically 8-32 bits long which is the same as the link width between two adjacent switches. In case of blocking, the worm stands on the path until the block

is removed. In virtual cut-through, buffers are provided in the switch so that a message can be buffered when it is blocked. We use these switching techniques because they are used in the design of current multiprocessor switches [3, 6]. There are two types of messages that are transmitted over the MIN. One is a *control* message that consists of read, write and invalidation signals that are short and 4 flits long. The *rdata* and *wdata* messages supply the block to the requesting cache and are 20 flits long for a cache line size of 32 bytes. Wormhole routing or virtual cut-through will be better than packet switching because it will save 20 cycles per switch without having to buffer the large data packets.

A simple wormhole (SWH) 2x2 crossbar switch is shown in Fig. 2.a. The amount of buffer needed per input and output for wormhole routing is only one flit that is essential for transmission over the links. Once the flit arrives, the arbitration starts and the connection is made. Such a simple crossbar arbiter takes one cycle arbitration time [12], however, the grant signals are modified to ensure continuity of transmission due to wormhole routing. The input synchronization takes one to two cycles, and it takes one more cycle for transmission. As a result, the switch delay is about 4 cycles, which is similar to the time taken in SGI Spider and Intel Cavallino switches [3, 6]. Many times the requests arrive in bulk. For example, in case of a write miss on a block, the interface/coherence controller sends invalidation signals to processors having a copy of the block one after another. In such situations, providing some buffer in the switches will be helpful. We study the effect of providing a buffer size of 8 flits and 32 flits so that a control message or a data message can be held entirely in the switch if required. However, the arriving message is bypassed without being stored in the buffer if the output is free. Note that the output buffer is still one flit. We call such a switch a buffered cut-through (BCT) design and is shown in Fig.2.b.

Virtual channels (VC's) [5] are used in wormhole networks to avoid deadlocks and to improve link utilization and network throughput. Fig. 2.c shows such a simple virtual channel (SVC) design with two VC's. Whenever there is blocking of a message, another message can be routed through the extra VC. In this situation, the arbiter consists of two simple 4 to 1 crossbar arbiters. When the number of inputs is more (6 inputs with 4 VC's or buffers per input which is equivalent to 24 possible inputs per output in Cavallino and Spider [3, 6]), two-stage arbiters are needed to keep arbitration time within limit. However, we have no such problem with a simple 2x2 switch. Each of the virtual channels can again have space for multiple flits. Flits from only one message can occupy the buffers at one virtual channel at a time. There are no buffers on the output side so that an unnecessary delay can be avoided. We call such a design a buffered virtual channel (BVC) switch, as shown in Fig. 2.d. As denoted in Fig. 2, the switch designs can be represented in terms of two parameters, buffer length (B) and number of virtual channels (VC). For SWH, B=1 and VC=1; for BCT, B=8 or 32 and VC=1; for SVC, B=1 and VC=2; and for BVC, B=8 or 32 and VC=2.

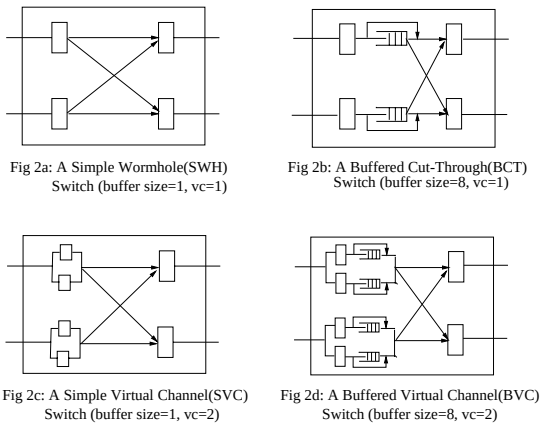

Figure 2. Four designs for a 2x2 crossbar switch

The BVC switch is quite complicated because there can be four different messages per input competing for an output port at a time. We designed a modified version of round-robin arbitration policy to reduce the message latency. Here the virtual channel that used the link in the previous cycle gets priority if it is still transmitting the flits from the same message, otherwise the round-robin policy is used. When the last flit of a message is sent, the round-robin policy avoids starvation on other channels. The crossbar output selection and conflict resolution is a two-step process. In the first step, the input channel that has a message to send, generates a request for the output channel. Multiple input channels can request the same output channel in this phase. This conflict is resolved in the second step. A similar arbitration policy is used in the Intel Cavallino switch [3].

3. Simulator design

Our simulator is based on PROTEUS [2] which implemented MIN using an analytical model. We have modified the simulator extensively to exactly model the MIN with wormhole routing. We have also incorporated the switch architectures with virtual channels and multi-flit buffers, as explained in section 2. The system parameters used in the simulation are listed in Table 1. We simulated a 16 node system using a 4 stage MIN with 512KB of memory per node which is enough to hold the shared data of the chosen applications. We chose a 16-node system to limit the simulation time, a larger system would need an appropriately larger data structure to get realistic results. Moreover,

Parameter	Value
Number of processors	16
Shared memory size per node	512 Kbytes
Cache size	32 Kbytes
Cache line size	32 bytes
Set size	2
Cache access time	2
Directory access time	2
Memory access time	16
Switch delay	1, 2, or 4
Link width	16 bits
Flit length	16 bits
Message length	8 or 40 bytes
Page size	1 Kbytes
Interface delay	20

Table 1. Simulation parameters

since our study focuses on the switch instead of scalability, a detailed simulation of a 16 processor system is more appropriate than a high level simulation of a bigger system. The cache parameters used in the simulation are similar to the HP PA-8000. There is one level cache of size 32 KB per processor with a line size of 32 bytes, 2-way set associative organization and access time of 2 cycles. The switch parameters are set as explained in section 2. A flit size of 16 bits was considered which is same as the width of the link. The switch latency or delay is considered to be 1, 2 or 4 processor cycles. The SGI Spider and Cavallino [3, 6] both have 4 cycles of latency. Additionally, we consider 1 or 2 cycle delays for comparison since our aim is to design high-speed switches. The interface delay for protocol handling, etc. is 20 cycles, similar to DASH [9]. As explained in section 2, the messages are of two different lengths in a cache coherent multiprocessor. We assumed message lengths of 8 and 40 bytes for control and data messages, respectively, similar to the SGI switch.

3.1. Cache coherence and synchronization

We implemented the full-map directory-based cache coherence protocol [4] with some modifications for evaluation in this paper. In this scheme, each shared memory block is assigned to a node, called *home node*, which maintains the directory entries for that block. An invalidation protocol has been implemented in which all the cached copies of a block are invalidated on a write operation. We have modified the coherence protocol where the cache controllers detect if a message has arrived out-of-order and holds it to be serviced later. The synchronization method used in our simulations is based on spin-locks using test-and-set operation with exponential backoff [10]. Barriers used in many of the applications were implemented using a shared counter. We also experimented with other types of barriers to reduce contention, however, our experience suggests that the main overhead in synchronization is the contention for the lock itself, not for the shared variable in the barrier. The overhead due to contention on synchronization variables is significant for some applications, such as FFT and MP3D [11]. The network latency also plays an important role on the synchronization overhead. In this paper, we do not discuss the synchronization overhead associated with any of the simulations. We concentrate only on the read/write requests to data items and the coherence requests.

3.2. Benchmark applications

We have selected some numerical applications as the workload for evaluating the network performance in a cache-coherent shared-memory environment. These applications are multiplication of two 2-D matrices (MATMUL), Floyd-Warshall's all-pair-shortest-path algorithm (FWA), blocked LU factorization of a dense 2-D matrix (LU), 1-D fast Fourier transform (FFT), and simulation of rarefied flows over objects in a wind tunnel (MP3D).

The matrix multiplication was done between two 128×128 double precision matrices. The principal data structures are four shared two-dimensional arrays of real numbers: two input matrices, a transpose matrix, and one output matrix. The shared data size is about 512 Kbytes. The problem was partitioned into square blocks of the output matrix, and all the elements in a block are computed by one processor. There is no real sharing of writable data, but there may be some false sharing of cache blocks at the boundaries of partitions. This type of partition minimizes the amount of shared data accessed by each processor. One of the input matrices is transposed to avoid large number of conflict misses.

For Floyd-Warshall's algorithm, we used 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 another predecessor matrix. The distance matrix is initialized with the weights of the edges in the graph. The problem is partitioned as per the rows in the distance matrix, so a set of vertices is statically scheduled to a processor to compute the shortest paths from them. Therefore, the elements in the distance and predecessor matrices are modified by only one processor during the whole execution. The program goes through as many iterations as the number of vertices, and during an iteration a particular row of distance and predecessor matrix is read by all the processors. Each iteration is followed by a barrier.

The blocked LU decomposition application was done on a 256×256 matrix using 8×8 blocks. The principal data

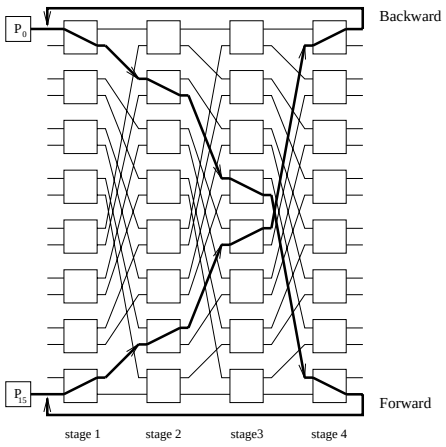

Figure 3. Path for forward and reply messages

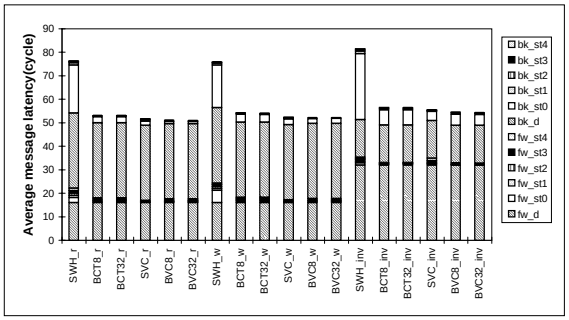

Figure 4. Avg. message latency for LU

structure is a two-dimensional array in which the first dimension is the block, and the second contains all data points in that block. In this manner, all data points in a block (which are operated on by the same processor) are allocated contiguously, and false sharing and line interference are eliminated.

We implemented Cooley-Tukey 1-D FFT algorithm. The simulations are done on an input of 2^{14} points. The principal data structures are two 1-D arrays of complex numbers. There is no data sharing during first $\log_2(N/P)$ stages, where N is the number of data points and P is the number of processors. In the remaining $\log_2 P$ stages, every data point is shared by two processors. During these stages, instead of using two separate input and output arrays, we interleave these arrays to avoid conflict misses.

MP3D is a three-dimensional particle simulator used in rarefied fluid flow simulation. We used 25000 molecules with the default geometry provided with SPLASH [11] which uses a $14 \times 24 \times 4$ (2646-cell) space containing a single flat sheet placed at an angle to the free stream. The simulation was done for 5 time steps. In this application, the active space is divided into cells, and the molecules interact only with the molecules in the same cell. There are two principal data structures: one stores the state information of each molecule, and the other stores the properties of each cell. The shared data set size is about 800 Kbytes. The work is partitioned by molecules, which are statically scheduled on processors. A clump size of 8 was used.

4. Switch/network level analysis

When a processor generates a read or write request, different actions are taken according to the directory protocol and depending on the status of the block in the cache, as explained in section 3.1. For example, if it is a read miss and the local memory is the home node, the request is satisfied locally. If the home is a remote node, the request is sent over the network to that node. At the remote node interface, the directory is checked and the data is supplied to the requester over the network again. The paths taken for the forward request and the reply message are different, as shown by bold lines in Fig.3 for processor P0 and memory M15. These messages encounter delays at various stages (1 through 4) of the network. In addition, there is a delay at the interface (called stage 0) to pump the message into the first stage of the network. The purpose of this section is to analyze the waiting delays at different stages for various applications and measure the effect of the improved switch designs, presented in section 2.

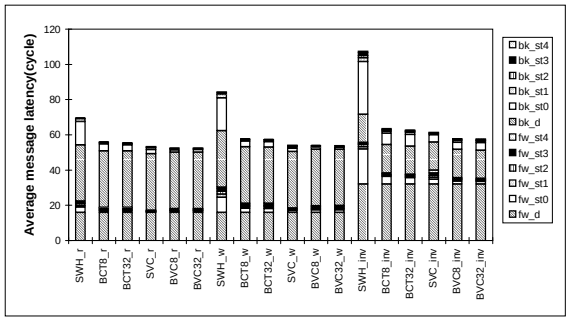

Figure 5. Avg. message latency for MATMUL

We present the results for different applications under two groups. The first group consists of MATMUL, LU and MP3D whose results are presented in Figs. 4, 5 and 6 respectively. The average message latencies for read, write and invalidation are plotted separately, as measured from the simulation. The average message latency is obtained by dividing the total time taken for these types of messages by the number of such messages throughout the execution of

the program. Consider the first bar for read request as an example. Starting from the bottom, we first show the transfer time (fw_d) over the network for the forward message which is the read request. This is the time without considering any waiting at switches. The next latencies are the waiting times that occur at stages 0 through 4, as denoted by fw_st0 through fw_st4. Stage 0 is the interface and stages 1 through 4 correspond to the 4 stages of the MIN from left to right, as marked in Fig.3. On the same bar, time taken for the reply (backward) message over the network is also plotted. Since a data block consists of 20 flits, so transfer time is much bigger, as shown by bk_d. The next field (bk_st0) indicates the time taken at stage 0 which is at the interface. This does not include times for memory access, directory access or protocol processing. Figs.7 and 8 present the average message latencies for the second group of applications, namely, FFT and FWA. In these two figures, one can notice a huge waiting delay at stage 0 (interface) for the backward messages. A similar situation was observed in Cedar network [13] where there was a large delay at the input of the backward network. In our case, the same network is used both for forward and backward requests and we use wormhole or cut-through switching instead of packet switching in Cedar.

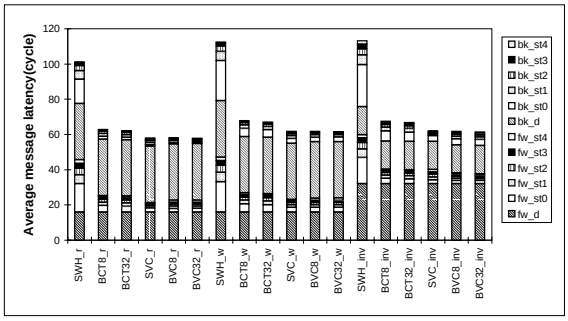

Figure 6. Avg. message latency for MP3D

Figs. 4 through 8 also show the waiting times at different stages for all requests. They appear to be negligible, meaning that there isn't much interference between various messages in the network. The write latency consists of both the forward write miss request and backward data block supply only. It does not include time taken for invalidation. The latencies due to invalidation requests and their acknowledgment or update (in case of a dirty block) are counted in the last set of bars in Figs. 4 through 8. The forward invalidation requests originate from the home memory and are sent to all nodes that have a copy of the block. From the transmission delay (fw_d), it is obvious that there are more than one copy of the block because the time taken is more than the corresponding delays for a read or write request. It may be noticed that the effect of the switch architectures is similar in these cases to the case of read miss. Also, there is a large delay at the interface for the backward message. Since acknowledge messages are short, the delay is due to the presence of dirty copies in remote nodes which have to be updated at the home memory. Also, like read data, there seems to be an accumulation of these dirty blocks at the same node at a time.

In order to understand why there is such a big delay at the input of the switch we need to understand the operation of the network interface in detail. The interface, shown in Fig.1, provides a large buffer space sufficient for holding all incoming and outgoing messages. Whenever there is space in the output transmit buffer, the large interface buffer is bypassed. The average queue length for the forward and backward requests at the interface buffer is shown in Table 3 for different applications. A big difference in queue length between the forward and backward messages may be noticed in case of FWA and FFT applications. More importantly, we observed that these backward messages occur in burst not being uniformly spread across the execution time. Although the overall requests to the memories are equally distributed over the entire execution period (as shown previously in Table 2), it is observed that memory requests from processors are directed to a particular memory (or page) at a certain time. Hence, at times, the memory or interface has to supply the data blocks to various processors one after another. Since the memory access rate (16 cycles) is smaller than the transmission time of about 30 cycles (without any interference in the network), these messages are queued up temporarily in the interface buffer.

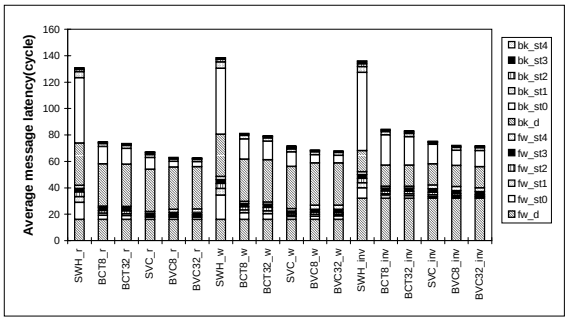

Figure 7. Avg. message latency for FFT

The above situation is caused due to the round-robin memory management policy of page allocation and due to the behavior of applications in that the data items are either read/written from/to one page at a time or different pages are accessed but wrap-around to the same memory mod-

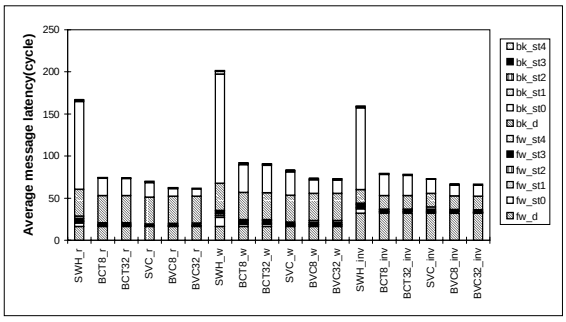

Figure 8. Avg. message latency for FWA

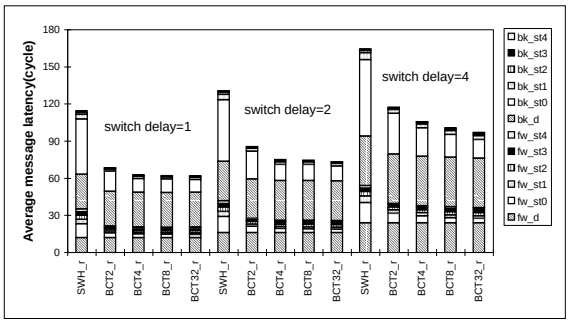

Figure 9. Avg. read message latency for FFT

Application	Message Type	Queue Length
FWA	forward	0.019304
	backward	0.146970
FFT	forward	0.066796
	backward	0.202756
MATMUL	forward	0.012961
	backward	0.022165
LU	forward	0.012619
	backward	0.032083
MP3D	forward	0.110048
	backward	0.120732

Table 2. Avg. queue length at network interface

ule. Designing further optimal memory management policies for different applications has been examined in [15] and is beyond the scope of this paper. Given the access pattern, our aim is to reduce this delay at the interface by designing a smoother pipeline in the switch architecture. The next bar in Figs. 4 through 8 show a drastic reduction in the delay in buffered wormhole(BCT) architecture (fig. 2b) where we provide a buffer of 8 flits at the input of the 2x2 switch. Increasing the buffer size to 32 flits, which can hold a complete data message (20 flits), does not improve the performance any further. The same gain can be obtained by providing two virtual channels per switch through a simple wormhole virtual cut-through (SVC) design, shown in Fig. 2c. With a buffered virtual cut-through (BVC) design of Fig. 2d, the latency can be cut down a little further, but there is almost no difference between the buffer lengths of 8 or 32 flits.

In order to understand why there is such a big improvement in latency between buffer sizes 1 and 8, but no improvement between 8 and 32, we performed another set of experiment by varying the buffer size. Fig. 9 shows the average read message latency for FFT application for SWH and BCT architectures with buffer size 1, 2, 4, 8 and 32. We also vary the switch delay to be 1, 2 and 4 cycles. Since a header is blocked for the switch delay number of cycles at a switch, providing a buffer size equal to the switch delay will ensure a continuous flow of the message from the interface and will avoid waiting there. Providing any more buffer than the switch delay will not increase the performance. This is true assuming that there is no interference between various messages in the network. It is noticed in Fig. 9 that most of the gain is achieved when buffer size equals the switch delay. Improvement after that is minimal and is due to a little interference that is still present in the network.

5. System level performance

The average message latencies, in the previous section, present the delay involved in a single message to the remote node. They accurately represent the interference between the messages in an application, but do not say anything about how many such remote messages are encountered during the program execution or the total stall time by a processor due to these remote messages. In this section, we present read, write and invalidation stall times for different applications as a function of switch delay and link width. Since section 4 shows that the BVC architecture performs the best, we fix the switch architecture with two virtual channels and a buffer size of 8 flits in this section.

The current switch delay in the commercial machines is about 5 switch core cycles (switch core operates slower than the CPU) for a 6x6 crossbar [3, 6]. With a 2x2 switch, we could obtain a faster switch because it takes much less time to arbitrate. However, since we assume switch core frequency to be the same as the CPU clock frequency, we derive the results with a switch delay of 1, 2 and 4 cycles in this section. The other parameter we vary is the flit size or

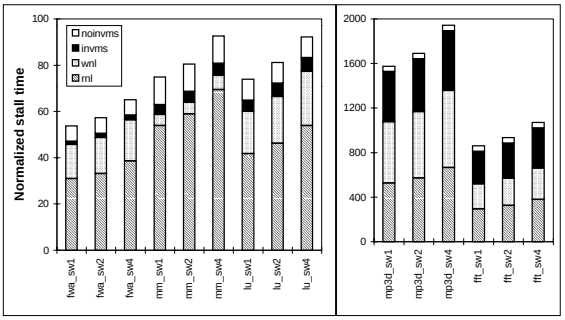

Figure 10. Normalized stall time results

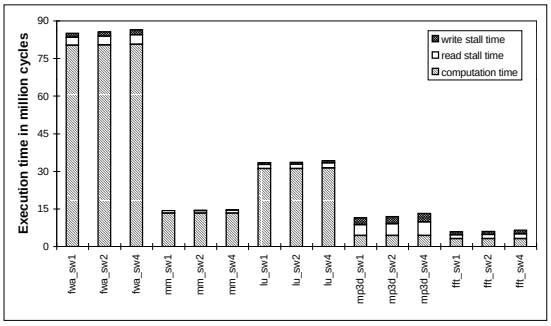

Figure 11. Execution time results

the width of the link. The flit size is kept the same as the link width throughout this simulation. With 2x2 switches, the link width can be increased to 32 bits and easily satisfy the pin constraints with the current technology. The increase in the link width reduces the transmission time over the network because a message takes fewer flits. As an example, with 32 bits width, a control message has two flits and a data block has only 10 flits which would cut down the transmission time (not the switch delay or waiting time), presented in the previous section, by half.

Stall time is the total amount of time a processor waits for read/write operation. Apart from the cache cold and capacity misses, it also partly represents the amount of interprocessor commuincation involved in an application. Stall time can be divided into several components based on the different system components. However, we only present the network stall time and memory stall time separately. The read network latency (RNL) is the total time spent in the network during execution of a program to transfer the read requests and their replies. Similarly, the write network latency (WNL) is the total time spent in the network during execution of a program to transfer the write requests and their replies. It does not include the invalidation time, if any. Memory Service Time is the time taken for both read and write requests to be serviced by the memory controller thus generating a reply. Further we would like to address the invalidation overhead. Thus we divide the requests that reach memory into those that require invalidations and those that do not. These are called memory service time with invalidations (INVMS) and memory service time without invalidations (NOINVMS), respectively.

According to the above types of stall time, we present the simulation results for a 16 processor MIN-based NUMA in Figs. 10, 11 and 12. Figure 10 shows the performance variation of RNL, WNL, INVMS, and NOINVMS stall time for various applications with switch delays (sw) of 1, 2 and 4 cycles. The variation is not very significant compared to the overall stall time. All the results have been normalized with their computation times, so that the y-axis represents the proportion of time spent on stalling for different applications. Normalized Stall Time is 1000 times the ratio between stall time and parallel computation time of the same application. For FWA, LU and MATMUL, we use the scale on the left side in the figure. The stall times of MP3D and FFT applications are high, so they are plotted on a different scale. The execution times of different applications are presented in Fig. 11 where each bar is divided into computation time, read and write stall times. The stall times of applications like FWA, LU and MATMUL are quite low because they are mostly shared read kind of applications. On the other hand, applications like MP3D and FFT spend as high as 50% of their execution times in waiting for read/write/invalidation messages. In Fig. 11, the computation time includes the time for synchronization overhead.

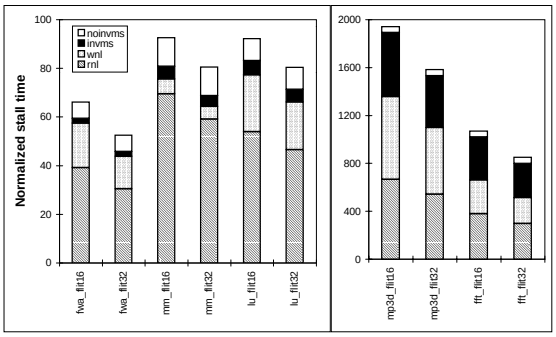

Figure 12. Impact of varying flit length

It may be noticed from Fig. 10 that the network stall time increases by about 10% with increase in the switch delay from 1 to 4 cycles. For FWA, MATMUL and LU, the read network stall times dominate because they involve

mostly shared read data. For MP3D and FFT, the stall times are distributed among read, write and invalidation messages because these applications have a higher percentage of write sharing. We also observe that the invalidation overhead is significant. We can reduce the invalidation and write times by one network latency by incorporating dirty forward where the processor having a dirty copy sends the block directly to the requester instead of through the directory update, like in DASH [9]. The invalidation overhead is also highly dependent on the network design. To reduce this overhead we need to employ techniques such as broadcasting and multicasting. We can thus perform the memory access and many invalidations simultaneously. The faster the network, the faster the invalidation requests and replies can be sent or received. In Figures 4 through 8, we showed that the invalidation delay reduces tremendously with increased virtual channels and buffer space.

The variation of normalized stall time with different link width or flit size is shown in Fig. 12. For all the applications, there is only about 15% improvement in the stall time. The figure considers a switch delay of 4 cycles which means there is a constant overhead of 16 cycles in one direction. Increasing the flit size to 32 bits will reduce the delay only by 2 cycles for control messages of 64 bits. It will reduce the data transmission time by about 10 cycles. Thus, the overall effect is not substantial. It must be kept in mind that we are already using a high-performance BVC switch with a buffer size of 8 bytes and 2 virtual channels that already cut down the waiting time tremendously. If we used a simple wormhole (SWH) switch architecture, there would be a substantial gain due to increase in the link width.

6. Conclusion

In this paper we presented detailed simulation results on four different switch architectures for a multistage interconnection network. The architectures are different in terms of their switching, buffer length and virtual channels. We presented the results in terms of network traffic at various stages, average message latencies and stall times. Improvements in average message latency due to increase in the number of virtual channels and buffer space were found to be as high as 70% for some applications. We measured the effect on read, write and invalidation messages separately and showed improvement in all cases. The effect of switch delay and link width on the stall time performance of these applications was also presented. Future work in this area includes relaxing the consistency model and studying the impact of the switch architectures while employing latency hiding techniques such as data prefetching and data forwarding. Also, prefetching of data should be incorporated to reduce the read latency from the execution point of view.

References

[1] L. Bhuyan and et al. Performance of multistage bus networks for a distributed shared memory multiprocessor. *IEEE Transactions on Prallel and Distributed Systems*, 8(1):82–95, January 1997.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. *Technical Report MIT/LCS/TR-516*, Massachusetts Institute of Technology, Cambridge, MA, September 1991.

[3] J. Carbonaro and F. Verhoorn. Cavallino: The teraflops router and nic. *Proc. Symp. High Performance Interconnects (Hot Interconnects 4)*, Augest 1996.

[4] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[5] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[6] M. Galles. Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip. *Proc. Symp. High Performance Interconnects (Hot Interconnects 4)*, Augest 1996.

[7] P. Gawghan and S. Yalamanchi. Adaptive routing protocols for hypercube interconnection networks. *IEEE Transactions on Computers*, 26(5):12–23, May 1993.

[8] A. Kumar and L. Bhuyan. Evaluating virtual channels for cache coherent shared memory multiprocessors. *ACM International Conference on Supercomuting*, Philadelphia, May 1996.

[9] D. Lenoski and et al. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[10] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *Proceedings of Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, pages 269–278, April 1991.

[11] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[12] Y. Tamir and H. Chi. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, January 1993.

[13] J. Torrellas and Z. Zheng. The performance of the cedar multistage switching network. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):321–336, April 1997.

[14] A. Vaidya, A. Sivasubramaniam, and C. Das. Performance benefits of virtual channels and adaptive routing: An application-driven study. *Proc. 11th International Conference on Supercomputing*, Vienna, July 1997.

[15] B. Vergese and et al. Operating system support for improving data locality on CC-NUMA computer servers. *Proc. ASPLOS-VII*, pages 279–289, October 1996.