An Efficient Single Copy Cache Coherence Protocol for Multiprocessors with Multistage Interconnection Network.

Ragab Omran and **Mokhtar Aboelaze**
Department of Computer Science
York University, North York, Ontario,
CANADA, M3J 1P3

ABSTRACT

Multistage interconnection networks offer an efficient, scalable, and reasonable cost solution for the problem of connecting processors to memory in a multiprocessor system. In this paper, we present an efficient single copy cache coherence protocol for multiprocessors with multistage interconnection network. Our protocol depends on incorporating the cache memory into the switches of the multistage interconnection network. In our proposed protocol, data blocks move between the switches in order to minimize the memory access time. We also develop a migration policy for data blocks not only to minimize the average memory response time but also to minimize the overhead in locating a specific memory block in case of a cache miss. We use two variations of our proposed protocol, the first allows caches to be in the caching switches in the first stage of the interconnection network only, while the second allows the switches to be in any switch in the interconnection network. We also use simulation to evaluate the performance of our protocol and compare it with one of the existing cache coherence protocol for systems employing a multistage interconnection networks. Our result indicate that our proposed protocol outperforms the write propagate protocol for a wide range of memory access patterns.

Key Words: cache coherence protocols, multistage interconnection networks, single copy protocol, multiprocessor systems, cache memory.

1 INTRODUCTION

Shared memory multiprocessor systems have been a very successful, cost effective way to meet the ever-increasing computing needs. However, due to the uneven growth of memory and processor speeds, memory access latency has been the most limiting factor in the speed of computation in high-performance multiprocessors. The use of private caches has proven to be highly effective in reducing memory latency and network traffic contention, however, they create yet another major hurdle, namely the well-known *cache coherence* problem [ArB86], [LeL90],[LeL92], [NaB93], [AnB93]. The problem arises when multiple copies of a shared block reside simultaneously in several caches and a processor modifies one of these copies. The problem has drawn considerable attention and numerous solutions have been proposed in the literature. Snoopy cache schemes are well-understood and known to offer a convenient solution to the cache coherence problem in bus-based multiprocessors.

In single bus architecture, snoopy bus protocols have shown to be an effective way of controlling the cache coherence problem [EgK89], [KaE85] [ArB86]. The snoopy bus protocols take advantage of the broadcast capability of the single bus. Any memory request by any processor is heard by all other processors, according to the type of the memory request, and the state of the requested block in the processor cache, an action is taken by all the caches which have copies of the requested block to assure the consistency of the memory (these actions may be invalidating a block, modifying a block, or supplying a copy of the block to the requesting processor). Snoopy bus protocols are effective for only a small number of processors, since by increasing the number of processors, the bus becomes the bottleneck in the system and the performance degrades.

The other extreme is a crossbar switch, in which every processor is connected to every memory module. The crossbar switch is very fast network. If the destination is free, a connection could be established in few switching devices time. The only drawback is that network complexity and cost become prohibitive for moderate to large number of processors.

As a compromise between these two extremes, multistage interconnection networks are considered a fast, reliable, and efficient means of communication for moderate to large number of processors. The multistage interconnection networks include Omega network, the Generalized Cube Network, and SW-Banyan Network, among many others [Sie90].

The memory coherence problem is usually handled in multistage interconnection networks using invalidation, or write-propagate techniques. In invalidation, if a memory block is about to be modified, all other copies of this block should be invalidated before the modification (write) is performed. In write-propagate protocols, all modifications to any memory block are propagated to all copies of this block thus guaranteeing a consistent memory.

In order to know the locations of the blocks either to be invalidated or modified, we have to have a directory to keep the information about the location of each data block. In directory based coherence protocols, [AgS88], [CeF78], and [ChK91], a centralized (or distributed) directory is kept in the main memory in order to keep track of all the caches that have a copy of any memory block. This directory is used in order to invalidate, or to modify, memory blocks. All references to shared memory blocks must go through this directory. Unfortunately, with increasing the number of processors, consequently increasing the memory size, the size of the directory increases and so does the directory storage overhead. Since the probability of simultaneously sharing a single memory block between a large number of processors is very small, and in order to decrease the storage overhead, Censier and Feautrier in [CeF78] proposed using a limited number of pointers to indicate the processors that have a copy of a particular memory block. Although this technique decreases the storage overhead, it has two important drawbacks. The first is we have to limit the number of processors that are allowed to share a single block (since we have a limited number of pointers), the second is operations such as block invalidation become inherently sequential requiring more time and slowing down the system.

A number of software techniques are used to maintain cache coherence, they depend on sophisticated compilers to detect data sharing and generate the appropriate control instructions. These protocols are very complicated and usually lead to a too conservative policies [AdA91].

Mizrahi et al in [MiB89] proposed a memory hierarchy network (MHN) and a single copy protocol in which the cache coherence is guaranteed by disallowing multiple copies of the same memory block. They also proposed that the MHN is used for handling shared data, while a separate network is used for handling private data and requests. The MHN idea is to consider the main memory as the root of a tree, while processors and their private caches are considered to be the leaves of that tree. Each internal tree node (excluding the root and the leaves) consists of a cache memory to hold data, a directory to locate the requested data block, and switching mechanism to forward the requested block. The authors also developed a data migration policy to control the movement of the memory blocks in order to minimize the average memory access time.

In a previous work [OmA93], we have extended this idea to multistage interconnection networks. In [OmA93] we assumed a generalized cube network in which caches are allowed in the first stage (close to the processors), the rest of the stages contain directories with the appropriate information in order to locate any requested memory blocks.

In this paper, we assume that every switch in every level in the multistage interconnection network is capable of holding caches. We propose a single copy protocol for such architecture, and we study the tradeoff of restricting the caches to the first stages only vs. allowing the caches to be in every switch in the network

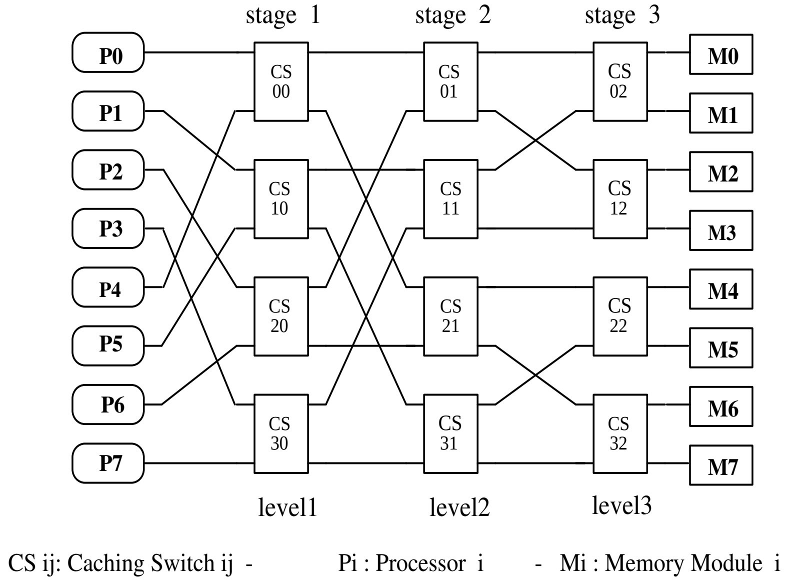


Figure 1: *The generalized cube interconnection network (*8x8*)*

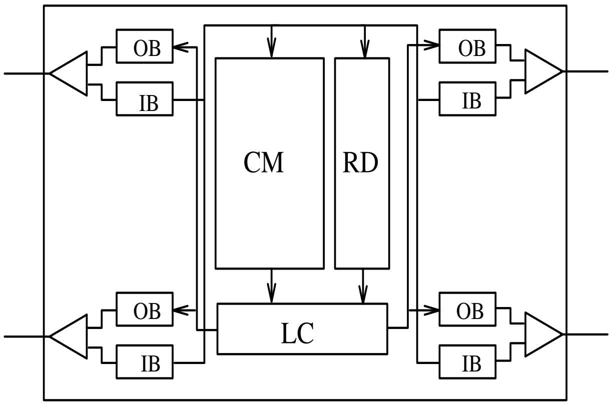
on the network performance. We also investigate the effect of the cache size on the network performance.

This paper is organized as follows. In section 2, we describe our proposed network architecture. In section 3, we describe our proposed coherence protocol. In section 4, we discuss the system model and the workload model we use in our simulation. In section 5, we present the simulation results and discuss them. Section 6 is a conclusion.

2 SWITCH ARCHITECTURE

Our proposed network architecture is the generalized cube network [Sie90] as shown in Figure 1. Although we assume this particular network in our analysis and simulation, our protocol could be implemented using any multistage interconnection network (Omega, SW-Banyan, among others).
As shown in Figure 1, the network consists of N processors connected to N memory modules using k stages, where $N = 2^k$. Each stage consists of $N/2$ switches, each switch is a 2-by-2 crossbar switch. We also assume an asynchronous, packet-switched network with worm-hole routing capability.

Each 2-by-2 switch, called *caching switch*, has a cache memory to store memory blocks and a routing directory to help finding the location of the different memory blocks. We assume that our network follows *partially* the *inclusion property* [BaW88] in routing directory information but not in memory blocks (since the protocol is a single copy protocol). What do we mean by following partially the inclusion property is as follows: If a block exists in any caching switch, this switch's routing directory, and *some* the routing directories in higher levels switches that



CM : Cache Memory - IB : Input Buffer - OB : Output Buffer
RD : Routing Directory - LC : Link Controller

Figure 2: *The organization of the memory hierarchy switch with a cache and a routing directory*

are ancestors to that switch, hold information about the location of this memory block, *i.e.* any switch routing directory have a directory entry indicating all the memory blocks in this switch, as well as, the memory blocks in some of the switches that are descendant from this switch. The remaining question is which subset of the blocks that are descendant from a particular caching switch is included in this caching switch routing directory ? The answer to this question is explained in the next section that describes our proposed protocol. Partial inclusion does not mean that the higher the switch in the hierarchy, the more memory it needs, since switches in higher stages need more memory for the routing directory but less memory for the cache, as it will become clear when discussing the simulation results in section 4.

Each caching switch, as shown in Figure 2, consists of:

I. Cache: The cache is used to store the unique copies of cached global memory blocks. In the first stage, the caching switch is connected to two processors, the two processors connected to this switch share the cache memory of this switch.

II. Routing Directory: The function of the directory is to maintain sufficient information to locate the memory blocks in its cache and in some of the caching switches in the subtree rooted at this caching switch. This means that the directory contain information not only about its memory blocks but also about some of the cached blocks in the subtree rooted at this switch. The information required can be encoded in two bits per memory block. If the directory does not contain an entry for this block, this means that the block resides in a caching switch outside this subtree, in this case, the request moves up in the network. Every node in the network is connected to 2 nodes in the upward direction, which node we have to forward the request to ? The answer to that question is a part of our proposed

protocol and it will be explained in section 3.
According to the location bits, the requested block could be in any one of three states.

LSS : The requested data exists in the Left Sibling Subtree.
RSS : The requested data exists in the Right Sibling Subtree.
VLD : The requested block is in this caching switch.

When the migrating block passes through the switch downwards, the directory adds the block tag to the directory table and sets the location bits of this block accordingly. If the block moves through the switch upwards, the directory entry for this block is deleted. The directories in the switches in stage 1 are much simpler than the directories in higher level switches, since in stage 1 caching switches, the only function for the directory is to indicate whether the memory block exists in this cache or not. If the block is not in that cache, it must be in another caching switch or in a memory module.

III. Link Controller: This module arbitrates between the upper and lower switch ports to route data and requests to their destinations. Shared data and shared requests are routed according to the directory entries (LSS, RSS) associated with each data block (or routed up if the entry does not exist).

IV. Input/Output Buffers and Ports: Each I/O port is a bidirectional link capable of transmitting request/data packets into and out of the switch. There are two buffers attached to each port implemented as FIFO queues. The buffers are used to queue packets to avoid conflicts and contention problems that occur when more than one packet arrive at the port to be routed on the same link. When a buffer saturates, data transfer to that buffer is blocked.

To implement the dynamic placement of data, switches communicate by the use of the different types of network packets; read-request, write-request, read-reply, migrate-request, and migrate-reply. When a processor issues a memory request (read/write-request), the first level caching switch, that is directly connected to it, will attempt to serve the request; if a miss occurs, then depending on the directory entry value, the request propagates through the hierarchy till the requested data is obtained. The requested block will be fetched from main memory, which is at the root of the hierarchy, if the block is not found in any of the caching switches. In the next section, we show how our proposed protocol utilizes this switch architecture in order to efficiently move the data blocks around the network to minimize the memory response time.

3 PROPOSED PROTOCOL

Our proposed protocol depends on the existence of a single copy of each memory block. When a processor request a certain memory address, if the requested block is in the processor cache (stage 1 caches), the data is sent to the processor (or modified if it is a write request). If the requested data is not in the processor cache, then the request is forwarded up in the interconnection network in order to locate the requested data block.

Each processor cache maintain a list of the α last referenced data blocks (in our simulation we assume that $\alpha = 4$). We call the Last Recently Used list of processor i (LRU_i). If processor i issued a memory request and could not be found in its local cache, the request propagates up in the network till the block containing the requested data is located, the requested block could be found in one of four locations, the action taken by the protocol depends on the location of the block. As we said before, the block could be in one of four locations.

1 The block is found in the main memory: In this case, the block is transferred to processor i cache.

2 The block is found in processor j cache, but not in LRU_j: In this case, the block is transferred to processor i cache.

3 The block is found in processor j cache and is in LRU_j: then, according to the principal of locality, processor j will reference this block in the near future with a high probability. In this case, the block is transferred to the caching switch which is the root of the subtree that contain both processors i and j as leaves.

4 The block is found in a caching switch in a stage other than stage 1: In this case the block is transferred to processor i cache in the second consecutive reference by the same processor (i).

The actions taken if the block is in locations 1 or 2 are self explanatory, and simply state that the requested block is brought to the requesting processor cache. If the block is in location 3 (*i.e.* in LRU_j), then processor j will probably request this block in the near future, moving it to cache i will result in ping-pong effect, we want this block to be on an equal distance from both processor i and processor j, that is why we move it to the root of the subtree containing both processor i and processor j.
If the block is in location 4, there are two possible reasons for any block to be in this location; either there are two processors actively referencing this block, or because it was brought to this location some time ago and no processor is actively referencing it. That is why we prefer to send the requested memory word and not the entire block on the first reference, but if the same processor (i) accesses the same block twice in a row, the block is moved to processor i's cache.

Allowing any memory block to reside in any caching switch, although simple to implement, creates many problems. The most important problem is how to locate a specific memory block. If a block was not found in a caching switch in the first stage, the request should be forwarded to the caching switch that have this specific block in its cache. Since each caching switch in stage 1 is connected to two caching switches at stage 2, we have to decide where to send the request to locate the block. Without any restrictions on where a memory block may reside, locating any memory block could be implemented in one of two ways.

(a) Either each caching switch contain a directory entry for every memory block, thus requiring a huge memory in each switch.

(b) Or, by forwarding the request to both switches in stage 2 connected to this switch, thus searching the entire network and creating unnecessary traffic.

In order to minimize the directory size, and to facilitate locating any block, we add some restriction on the location of any data block, this restriction could be explained as follows. Since we are assuming a shared memory model, the $logN$ most significant bits of any memory location determine the memory module which owns this block *"owner module"* A block could be in any caching switch in the tree rooted at the owner module of this block. Such a restriction facilitates locating the block. If a caching switch receives a request for a block, it checks to see if the block is in its routing directory. If the block in its routing directory, it send the request either to its left or right subtree according to the directory entry. If the block is not in the switch's routing directory, then, the request is sent up towards the owner module of the requested block, thus, limiting the search into the tree with the owner module being the root. This also clarifies the *partial inclusion property*, which in this case means: any caching switch CS_{ij}, contain an entry in its directory for each block in any caching switch descendant from CS_{ij} if and only if the caching switch CS_{ij} is a part of the tree formed with the owner module of that block as a root.

4 SYSTEM AND WORKLOAD MODELS

In this paper, we assume the multiprocessor system to be asynchronous system with N processors connected to N memory module via a generalized cube network as the one shown in Figure 1.

We assume that a processor generates a new access to a shared memory block such that the time between requests is exponentially distributed. We don't simulate access to private blocks since the purpose of this paper is to evaluate the cache consistency protocol. Memory coherence is not affected by accesses to private blocks.

The system model parameters can be summarized as follows.

N	:	Number of processors = number of memory modules.
p_w	:	The probability of a reference being a write.
B	:	The number of shared memory blocks in the system.
b	:	The size of the memory block.
τ	:	The average processor thinking time (time between memory accesses).
t_b	:	The time for a block to go through one switch.
t_r	:	The time for a request packet to go through one switch.
t_{mem}	:	Main memory access time.
t_{ch}	:	Cache access time

Some of the above parameters are fixed throughout this paper. We assume that $B = 2^{12} = 4096$, $b = 2^4 = 16$, $\tau = 2$, $t_b = 2$, $t_r = 1$, $t_{mem} = 4$, and $t_{ch} = 1$ time units, where the basic time unit is the processor cycle time.

Modeling the workload is more difficult than modeling the system. Many researchers have used real address traces collected from multiprocessors running benchmark programs. These simulation studies are limited to a relatively small number of processors, mainly because for large numbers of processors, a significant amount of storage is required to hold the traces. Furthermore, synchronization and ordering of these reference streams can be heavily dependent on the particular machine on which traces were collected, making these methods cumbersome to use.

In this paper, rather than using real reference traces, we adopt a flexible abstract model of reference patterns that can synthetically emulate the reference pattern of a variety of applications. We base our workload model on a variation on the model proposed in [DuB82]. These variations are mainly to allow independent referencing by N processors in a multiprocessor system.

Usually, in multiprocessor systems, memory references by any processor are not uniformly distributed over all memory space (modules). Processors tend to favor a subgroup of memory modules, either because of the clustering property of large programs, or because of the way processors communicate in shared multiprocessor systems (processor in a shared memory system communicate by writing a message in a prespecified place and setting some global variables or semaphores, then each processor checks its mail box periodically for messages, thus accessing this specific memory module more frequently than the other modules). This is known as the favorite memory concept [NaB93]. In this paper, we assume that memory module i is the favorite memory module for processor i. Our address generator implements this idea in generating the address trace. The address trace generator, built as part of the multiprocessor simulator, generates for every processor (i) an address that belongs to the following groups with the following probabilities.

p_{blk}	:	The probability that the reference is to the same memory block as the previous reference.
p_{grp}	:	The probability that the reference by processor i is directed to a block in LRU_i.
p_{mem}	:	The probability that the reference is directed to any block outside LRU_i.

When a processor references a block outside its LRU, we assume that this reference is to this processor favorite memory module with a probability β, and to other memory modules with a probability $(1-\beta)$. Thus, the probability for any processor to access its favorite memory module is βp_{mem}, and to access any other memory module with a probability $(1-\beta)p_{mem}/(N-1)$.

By changing the values of p_{blk}, p_{grp}, p_{mem}, and β we can simulate, and measure the performance of our protocol, for a wide range of memory access patterns without having to collect new traces.

5 SIMULATION RESULTS

In this section, we present and analyze our simulation results. We have simulated a 16x16 multiprocessor system with a generalized binary cube network using our proposed protocol. For comparison, we also simulated the same network with the same workload model using the protocol proposed in [Ste89].

In our simulation, we used CSIM [Sch90], a process oriented discrete-event simulation package for use with C language. A processor executes for a variable number of cycles, called thinking time, and then it issues a memory request. Processor computation periods between memory requests are assumed to be exponentially distributed with mean τ cycles. Once the request is made, the processor remains idle until its request is satisfied. If the request can be serviced locally by the processor's caching switch, then the processor resumes computation after one cycle. In case of a cache miss, the interconnection network is required to handle the miss by locating the requested block either in one of the caching switches or in the main memory.

We have simulated 4 different protocols.

1 MINCS: A single copy protocol using a Multistage Interconnection Network with Caching Switches. In this protocol, all the caching switches in the network contain caches. We assumed an infinite cache size.

2 MINCS-Level1: A single copy protocol using a Multistage Interconnection Network with Caching Switches. In this protocol, only the caching switches in the first stage have caches, the rest of the caching switches contain just the directory without caches. In this protocol, we also assumed an infinite cache size.

3 MCopy-500b: A Multiple Copy protocol proposed in [Ste89] where we set the maximum cache size to 500 blocks.

4 MCopy-Inf: A Multiple Copy protocol proposed in [Ste89] where the size of the caches is infinite.

Although we have assumed an infinite cache size for MINCS, because in a single copy protocols the data blocks are continuously moving between the different switches. We have found that the average cache size is approximately 520 blocks for MINCS-Level1, and approximately 490 blocks for MINCS. In MCopy-Inf, the average cache size is (2900 blocks), that is why we have decided to run the same protocol with a maximum cache of 500 blocks (the same as the average cache size for MINCS). We have also set β to 40% in our simulation. We also assumed that $\tau = 2$, the switch service time for a memory block to be 2 cycles, and the caching switch service time for a request packet (request to locate a memory block) to be 1 cycle.

Figure 3 shows the average request response time vs. data block locality (p_{blk}) for the four different protocols with $p_w = 20\%$. In this graph, p_{grp} is fixed to be 60% except the case of $p_{blk} = 0.45$, where we set p_{blk} to be 50%. In this graph, we notice that MCopy-Inf is the worst possible protocol. This is mainly because

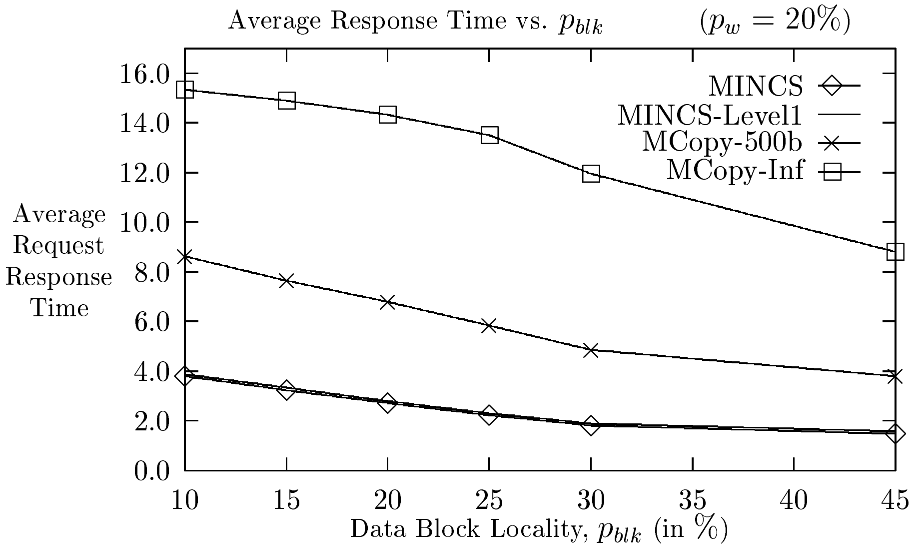


Figure 3: Average Request Response Time for 16 processors

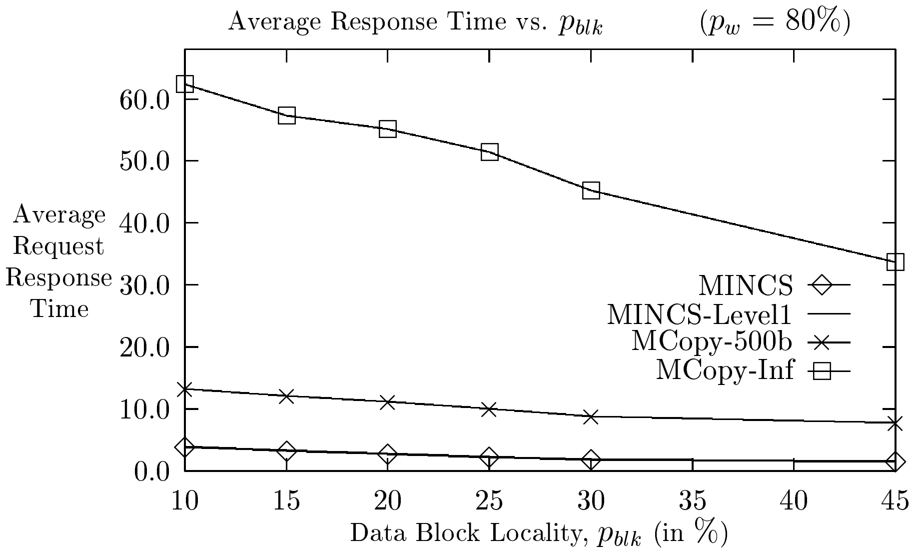


Figure 4: Average Request Response Time for 16 processors

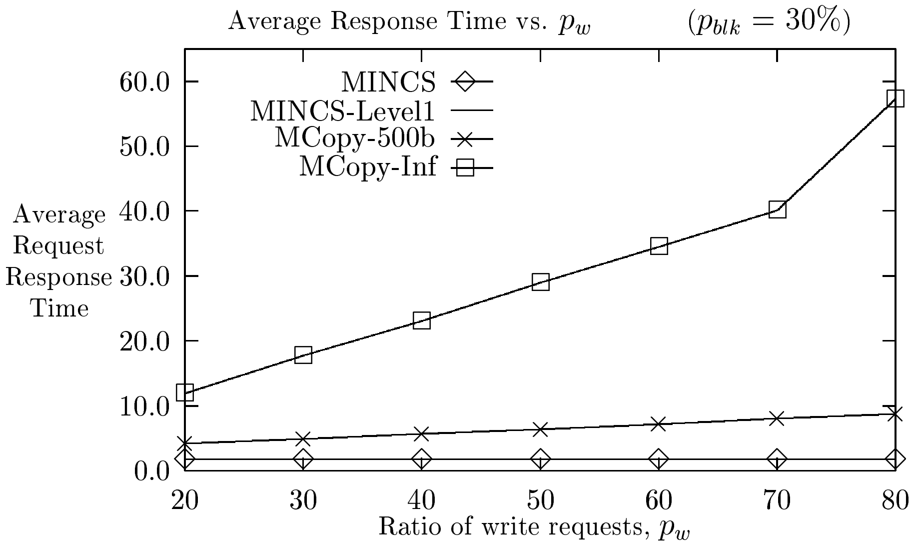


Figure 5: Average Request Response Time for 16 processors

of the infinite cache size with distributed write. Most of the distributed writes are to blocks that are not active anymore. However, since we have no mechanism to invalidate them, they are still being updated. MCopy-500b has a much better performance than MCopy-Inf. However, even at $p_{blk} = 0.45$, and $p_{grp} = 0.5$, which results in a very high hit ration (more than 95%), the average response time is the same as the main memory response time (4 cycles). Although the performance of MINCS and MINCS-Level1 is approximately the same, MINCS needs slightly less total cache size as it is clear from Table 1 for $p_{blk} = 0.45$ and $p_{grp} = 0.5$. Figure 4 shows the same performance measure with $p_w = 0.8$, which results in a very bad performance for distributed write (more writes means more updates). While MINCS and MINCS-Level1 has a much better performance.

protocol	MINCS-Level1	MNCS Stage 1	MINCS stage 2	MINCS stage 3	MINCS stage 4
Cache Size	524	495	2.2	3.8	7.4

Table 1: The average cache size for MINCS and MINCS-Level1

Figure 5 shows the average memory response time vs p_w for the 4 mentioned protocols for $p_{blk} = 0.3$ and $p_{grp} = 0.6$. We notice that the previous pattern continue with MCopy-Inf as being the worst possible protocol, MCopy-500b as a better protocol, and MINCS and MINCS-Level1 outperforms both of them.

6 CONCLUSION

In this paper, we have presented a new single copy protocol that can be implemented on any multiprocessor with a multistage interconnection network. The basic idea of this protocol is to distribute the cache memory among the network switches. In every switch, there is a directory to locate different memory blocks. By limiting the caching switches any memory block can reside in, we can minimize the routing directory size. Finally, we have presented simulation results to compare our proposed protocol with a previously known multiple copies protocol. The simulation results indicate that our protocol is superior to the more traditional write propagate directory-based protocols for a wide range of memory access patterns.

REFERENCES

[AdA91] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, *Comparison of Hardware and Software Cache Coherence Schemes* "Proceedings of the 18th Annual Int. Symp. on Computer Architecture," 1991, pp. 298-308..

[AgS88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, *An Evaluation of Directory Schemes for Cache Coherence* "Proceedings of the 15th Annual International Symposium on Computer Architecture," 1988, pp. 280-289.

[AnB93] C. Anderson and J.-L. Baer, *A Multi-Level Hierarchical Cache Coherence Protocol for Multiprocessors* "Proceedings of the 7th International Symposium on Parallel Processing," Newport Beach, CA May 1993, pp. 142-148.

[ArB86] J. Archibald and J.-L. Baer, *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model* "ACM Transactions on Computer Systems," Vol. 4, No. 4, November 1986, pp. 273-298.

[BaW88] J.-L. Baer and W.-H. Wang, *On the Inclusion Properties for Multi-Level Cache Hierarchies* "Proceedings of the 15th Annual Int. Symp. on Computer Architecture," 1988, pp. 73-80.

[CeF78] L. M. Censier and P. Feautrier, *A New Solution to Coherence Problems in Multicache Systems* "IEEE Transactions on Computers," Vol. C-27, No. 12, December 1978, pp. 1112-1118.

[ChK91] D. Chaiken and K. Kubiatowicz, *LimitLESS Directories: A Scalable Cache Coherence Scheme* "Proceedings of the 4th International Conference on ASPLOS," 1991, pp. 224-234.

[DuB82] M. Dubois and F. A. Briggs, *Effects of Cache Coherency in Multiprocessors* "IEEE Transactions on Computers," Vol. C-31, No. 11, November 1982, pp. 1083-1099.

[EgK89] S. Eggers and R. Katz, *Evaluating the Performance of Four Snooping Cache Coherency Protocols* "Proceedings of the 16th Annual International Symposium on Computer Architecture," 1989, pp. 2-15.

[KaE85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, *Implementing a Cache Consistency Protocol* "Proceedings of the 12th

Annual International Symposium on Computer Architecture," 1985, pp. 276-283.

[LeL90] D. Lenoski, J. Lauden, K. Gharachorloo, A. Gupta, and J. Hennessy, *The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor* "Proceedings of the International Symposium on Computer Architecture," 1990, pp. 148-159.

[LeL92] D. Lenoski, J. Lauden, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, *The Stanford Dash multiprocessor* "IEEE Computer," March 1992, pp. 63-79.

[MiB89] H. E. Mizrahi, J.-L. Baer, E. D. Lazowska, and J. Zahorjan, *Extending the Memory Hierarchy into Multiprocessor Interconnection Network: A Performance Analysis* "Proceedings of the International Conference on Parallel Processing," 1989, pp. I-41-I50.

[NaB93] A. K. Nanda and L. N. Bhuyan, *Design and Analysis of Cache Coherent Multistage Interconnection Networks* "IEEE Transactions on Computers," Vol. 42, No. 4, April 1993, pp. 458-470.

[OmA93] R. Omran and M. A. Aboelaze, *Introducing Caching Switches into MIN-based Shared Memory Multiprocessors* "Proceedings of the Eleventh IASTED International Conference on Applied Informatics," Annecy, France May 1993.

[Sch90] H. Schwetman, *CSIM User's Guide,* MCC Technical Report Number:ACT-126-90, MCC Austin Texas, 1990.

[Sie90] H. J. Siegel, *Introduction to Interconnection Networks for Large-scale Parallel Processing,* McGraw-Hill, 1990.

[Ste89] P. Stenstrom, *A Cache Consistency Protocol for Multiprocessors with Multistage Networks* "Proc. of 16th Annual International Symposium on Computer Architecture ," 1989, pp. 407-405.