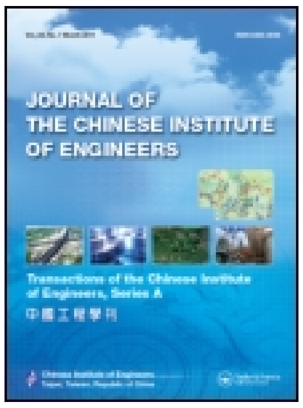

Journal of the Chinese Institute of Engineers
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tcie20

A performance study of cache coherence protocols and write caches for parallel-multithreaded shared-memory multiprocessors

Chao-Chin Wu [a] & Cheng Chen [a]
[a] Department of Computer Science and Information Engineering , National Chiao Tung University , Hsinchu, Taiwan 300, R.O.C.

Published online: 03 Mar 2011.

To cite this article: Chao-Chin Wu & Cheng Chen (1998) A performance study of cache coherence protocols and write caches for parallel-multithreaded shared-memory multiprocessors, Journal of the Chinese Institute of Engineers, 21:1, 33-46, DOI: 10.1080/02533839.1998.9670368

To link to this article: http://dx.doi.org/10.1080/02533839.1998.9670368

Journal of the Chinese Institute of Engineers, Vol. 21, No. 1, pp. 33-46 (1998)

A PERFORMANCE STUDY OF CACHE COHERENCE PROTOCOLS AND WRITE CACHES FOR PARALLEL-MULTITHREADED SHARED-MEMORY MULTIPROCESSORS

Chao-Chin Wu* and Cheng Chen
Department of Computer Science and Information Engineering
National Chiao Tung University,
Hsinchu, Taiwan 300, R.O.C.

ABSTRACT

According to published research results, no directory-based cache coherence protocol provides best performance for all application programs in conventional multiprocessor systems that use sequential consistency models. However, recently it has been claimed that competitive-update protocols are superior to other protocols under a relaxed consistency model. Moreover, incorporating write caches improves the system performance of clean and competitive-update protocols.

In this paper, we examine the different effects that occur when processing elements are replaced by parallel-multithreaded processors. According to our simulation results, the clean protocol provided the best performance for five out of six SPLASH programs. After augmentation with write caches, the clean protocol outperformed others for all applications. Though competitive-update protocols have been improved, their performance is not better than that of write-invalidate protocols for most programs.

Key Words: Write cache, Multithreaded processor, Shared-memory multiprocessor, Cache coherence protocol.

I. INTRODUCTION

According to Dahlgren and Stenstrom [4], cache coherence protocols can be divided into four types: write-invalidate, write-update, clean, and competitive-update. Write-invalidate protocols have lower write traffic and write penalties but have higher coherence miss rates and read penalties. Write-update protocols, on the other hand, have higher write traffic and write penalties but have lower coherence miss rates and read penalties. Clean and competitive-update are two hybrid update/invalidate protocols [4]. Clean protocols are similar to write-invalidate protocols except that they update the memory copy if there are copies in caches other than that from which the write request came. The advantage they offer is that latency periods for coherence misses can be reduced because the misses can be served by the memory controller instead of by a remote cache controller [4]. Competitive-update protocols are similar to write-update protocols except that a cache block is invalidated when it has been updated a fixed number of times, called the competitive threshold, without intervening local accesses. The advantage they provide is that the number of updates not contributing to reductions in the number of misses is reduced [4].

*Correspondence addressee

Unfortunately, none of them is always superior to the others for all application programs. According to previous evaluations [1, 6], determining which protocol is best depends on individual program characteristics. However, Grahn and Stenstrom [9] reported an exciting observation: competitive-update protocols outperform write-invalidation and write-update protocols in distributed shared-memory multiprocessor systems with centralized directories that use relaxed consistency models because (1) centralized directories provide higher network bandwidth for parallel updates, and (2) relaxed consistency models allow write request latencies to be hidden by overlapping them with each other and with local computation. In addition, compared with write-invalidate protocols, competitive-update protocols provide much lower read miss rates at a cost of some increase in memory traffic.

Dahlgren and Stenstrom [4] also studied a related topic: using write caches to improve the performance of cache coherence protocols in distributed shared-memory multiprocessors with centralized directories using relaxed consistency models. Write caches were first proposed by Bray and Flynn for uniprocessors. They used allocate-on-a-write-miss, write-back, and no-allocate-on-a-read-miss strategies, with a single combined dirty/valid bit per word [3]. For a write miss, a block frame in the write cache was allocated to buffer the data, and the corresponding dirty bit was set. Therefore, all write accesses belonging to the same block were merged into a single write miss request. Only dirty words needed to be transferred to the next level in the memory hierarchy when the block was replaced, or the whole write cache was flushed. Consequently, write traffic was reduced because of the temporal and spatial locality of write accesses.

However, write caches have a different effect on shared-memory multiprocessor systems. Unlike uniprocessor systems, memory access ordering requirements must be enforced by some underlying memory consistency model. Sequential consistency [12] and processor consistency [8] require that before any one processor can perform a write access, all previous write accesses in the program order must first be performed. Consequently, there is no use incorporating write caches in this kind of system because each write access must be made consistent with respect to other processors before the cache controller can issue the next memory request. However, in weak [5] or release [7] consistency systems, every ordinary access can be performed independently of other ordinary accesses, so write caches must only be flushed upon arrival of a synchronization access or a release access. Therefore, write caches can merge write access requests between any two adjacent synchronization points. Dahlgren and Stenstrom showed that using write caches can improve the performance of shared-memory multiprocessor systems that use relaxed consistency models [4].

In this paper we study the performance implications of cache coherence protocols and write caches in parallel-multithreaded shared-memory multiprocessor systems because of the importance of multithreaded architectures. According to the simulation results obtained from six SPLASH programs, with and without write caches, the clean protocol, rather than the competitive-update, had the best performance in this kind of architecture. We also found that merging too many write accesses delays the execution of release accesses and thus degrades system performance. Consequently, larger write cache sizes or complicated placement policies do not ensure better performance. We suggest write caches be direct-mapped with eight blocks.

The rest of the paper is organized as follows. Section 2 introduces parallel-multithreaded multiprocessor systems. Section 3 describes our simulation environment and benchmark programs. The effects of cache coherence protocols and adding write caches are evaluated in Sections 4 and 5. Then, the impacts of write cache parameters are discussed in Section 6. Finally, we give concluding remarks.

II. PARALLEL-MULTITHREADED MULTIPROCESSOR

Improvements in semiconductor technology allow much more complicated designs on single chips, stimulating the study of multithreaded processors [11]. In multithreaded architectures, whenever a long latency operation occurs in a running thread, the system immediately switches out the thread and selects another waiting thread for execution. Hiding long latencies by executing these other threads improves system resource utilization [10].

Parallel multithreaded processors (PMPs) [10] can execute more than one thread at the same time. Although, they usually require more expensive hardware and greater design complexity, they can hide latencies at the instruction level rather than at the thread level. When an instruction from a thread cannot be issued, because of either a control or data dependence within the thread, an independent instruction from another thread is executed instead. PMPs are stalled instead of performing context switching upon cache misses. Thus, the advantage of PMPs is that they enable greater hardware utilization because the processors' functional units are shared by all parallel running threads. We expect that this type of processor will become one of the most popular single-processor designs because of its superior resource

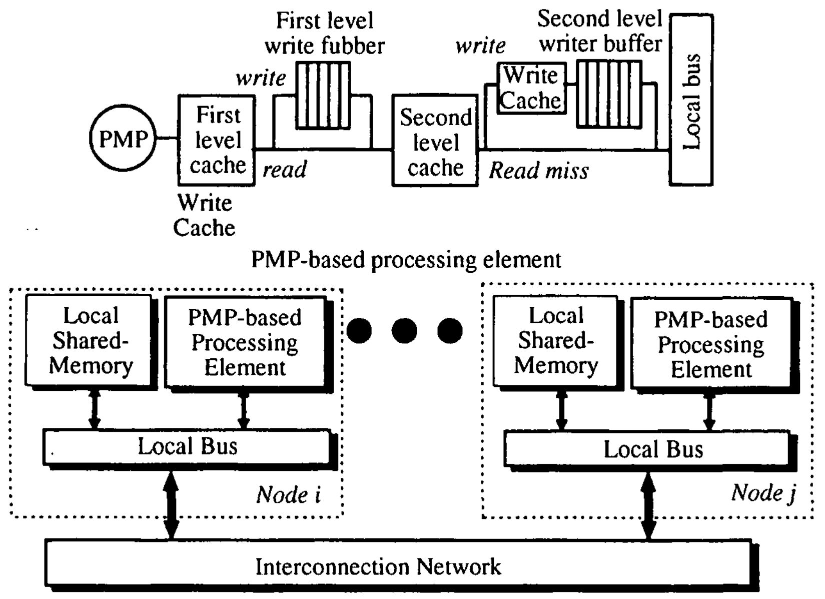

Fig. 1. Organization of PMP-based NUMA multiprocessor.

utilization.

PMP architectures are not powerful enough for huge computations, however, because a single chip cannot support execution of tens of parallel threads. One alternative architecture is the PMP-based multiprocessor system (PMP-MP) shown in Fig. 1. It is a cache-coherent, non-uniform memory architecture (CC-NUMA) and consists of a number of processing elements linked by an interconnection network. Each processing element contains a PMP, a private cache, and a part of the global shared memory. The shared address space is partitioned into equal-sized pages and these memory pages are distributed to the local memories of processing elements. Like conventional multiprocessor systems, the PMP-MP has to employ a protocol to maintain cache coherence because each processing element has its own cache. However, the most compelling feature of the PMP-MP is that several threads share only one cache in each processing element. Though memory accesses may replace useful data of previous executing threads in the cache, we do not handle this situation by a special method. In this paper, we focus on PMP-MP systems.

III. SIMULATION ENVIRONMENT AND BENCHMARK PROGRAMS

In order to evaluate the performance differences among various architectures, we constructed a simulation environment by extending the MINT package [15], which is a program-driven simulation framework that uses MIPS processors. The environment was constructed on a SUN SPARC workstation and written in C programming language. It consists of two parts: the memory reference generator and the memory subsystem simulator. The former interprets instructions and forwards memory references to the latter. The memory subsystem simulator is

Table 1. Architecture Parameters

Parameter	Value
Number of Processing nodes	16
Number of threads per PE	4
Size of FLC	16Kbytes
Size of SLC	256Kbytes
Block size of FLC and SLC	32bytes
Number of entries in FLWB	16
Number of entries in SLWB	32

composed of a two-level cache hierarchy, directory-based cache coherence protocols, memory consistency models and an interconnection network. All functions were modeled carefully and simulated cycle by cycle.

We assume that all memory accesses to code and private data always hit in the first level cache (FLC) and each takes a single processor clock cycle.

We summarize several important architecture parameters in Table 1 and others are as follows: (1) the processor is blocked on read misses but not on write misses [14]; (2) the processor clock rate is 100 MHz; (3) the FLC and second level cache (SLC) take 1 and 3 processor clock cycles, respectively; (4) write cache size is 8 blocks; (5) FLC, SLC, and write cache are all direct-mapped; (6) the size of the memory page is 4-Kbytes and the memory pages are distributed in a round-robin fashion; (7) the interconnection network is a 4-by-4 torus; (8) the linked width of the network is 64 bits; (9) release consistency is used as the memory consistency model; and (10) the cache coherence protocol is fully-mapped, directory-based.

Some parameters differ from those in Dahlgren and Stenstrom [4] and influence performance, so we must point them out. Because the total number of threads was 64 (16x4) in our study, the acquire-stall time was perhaps much larger than that for the 16 threads in [4]. In addition, because the SLC capacity was 256 Kbytes instead of the infinite size in [4], we have to consider the impact of replacement on the SLC. A smaller cache increases the number of dirty blocks that must be replaced and the number of coherence misses accessing to clean memory blocks under write-invalidate protocols, thus diminishing the advantage of the clean protocol.

We used the six SPLASH [13, 16] applications listed in Table 2 as benchmark programs. All applications were written in C using the PARMACS macros from Argonne National Laboratory [2] and were compiled using cc under IRIS version 3 at the optimization level 2 in an SGI workstation. Because the working set of these six programs was very small

Table 2. Benchmark Programs

Benchmark	Description	Data sets
MPED	3-D particle-based wind-tunnel simulator	5K particles, 10 time steps
Cholesky	Cholesky factorization of a sparse matrix	The matrix bcsstk 14
Pthor	Distributed time digital circuit simulator	RISC circuit, 1000 time step
Ocean	Ocean basin simulator	130×130 grid, tolerance 10^{-7}
Barnes	Hierarchical N-body gravitation simulator	1024 bodies, 3 steps
Water	N-body water molecular dynamics simulation	343 molecules, 3 time steps

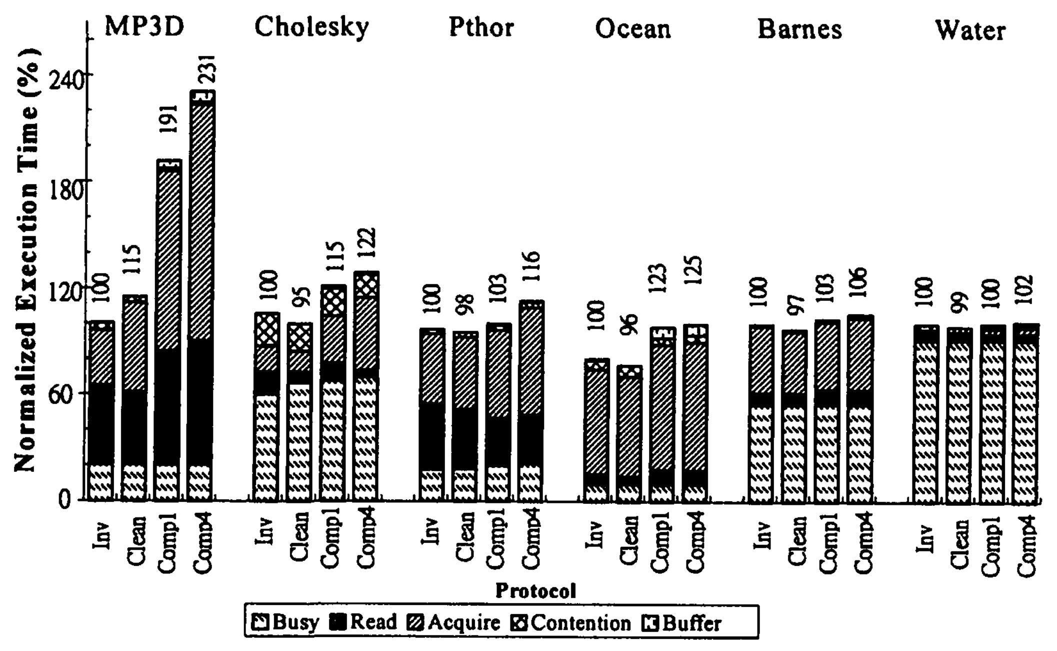

Fig. 2. Normalized execution time for various cache coherence protocols.

when compared to the size of the FLC and SLC used, the influence of replacements on the performance was little. The effect of coherence maintenance is the primary factor that influences performance. All statistics collected in the following were gathered in the parallel sections of the benchmarks.

IV. EFFECTS OF CACHE COHERENCE PROTOCOL

In this section we report on which cache coherence protocol proved more suitable for a PMP-MP without write caches. Fig. 2 shows comparisons of how write-invalidate protocols without write caches (Inv), clean protocols without write caches (Clean), and competitive-update protocol with threshold T and no write caches (CompT) performed. The execution times for each application and protocol were normalized relative to the execution times under Inv. In addition, we divided each execution-time bar into five sections: busy time (bottom section); the read-stall time (i.e., the time spent servicing cache misses); the acquire-stall time (i.e., the time spent waiting for a lock to be acquired); the contention time (i.e., the time spent waiting for access to the FLC); and on top, the buffer-stall time (i.e., the time the processor spnt stalled due to a full FLWB). Because Cholesky, Pthor, and Barnes are dynamically scheduled during the run time, the busy times may be not equal for different protocols. In addition, because Water has a long busy time, the potential performance improvement is very limited.

The clean protocol had the best performance on all application programs, as shown in Figure 2, except that the write-invalidate protocol performed better on the MP3D. The competitive-update protocol had longer execution times than the Inv for all programs. This result is different from the observations of Grann, Stenstrom, and Dubois [9], and we will explain the reason in the following.

For MPs with non-multithreaded PEs as evaluated in [9], the competitive-update protocol outperforms other protocols because: (1) the directory structure can update more than one cached copy simultaneously, and (2) only those cached copies recently accessed by their own PEs need to be updated, the others are invalidated and require no updating. The latter is the key difference between the competitive-update protocol and the write-update protocol. The write-update protocol updates all cached copies even though some of them not will be accessed again, resulting in huge network traffic. For MPs with PMP processors, because each PE executes more than

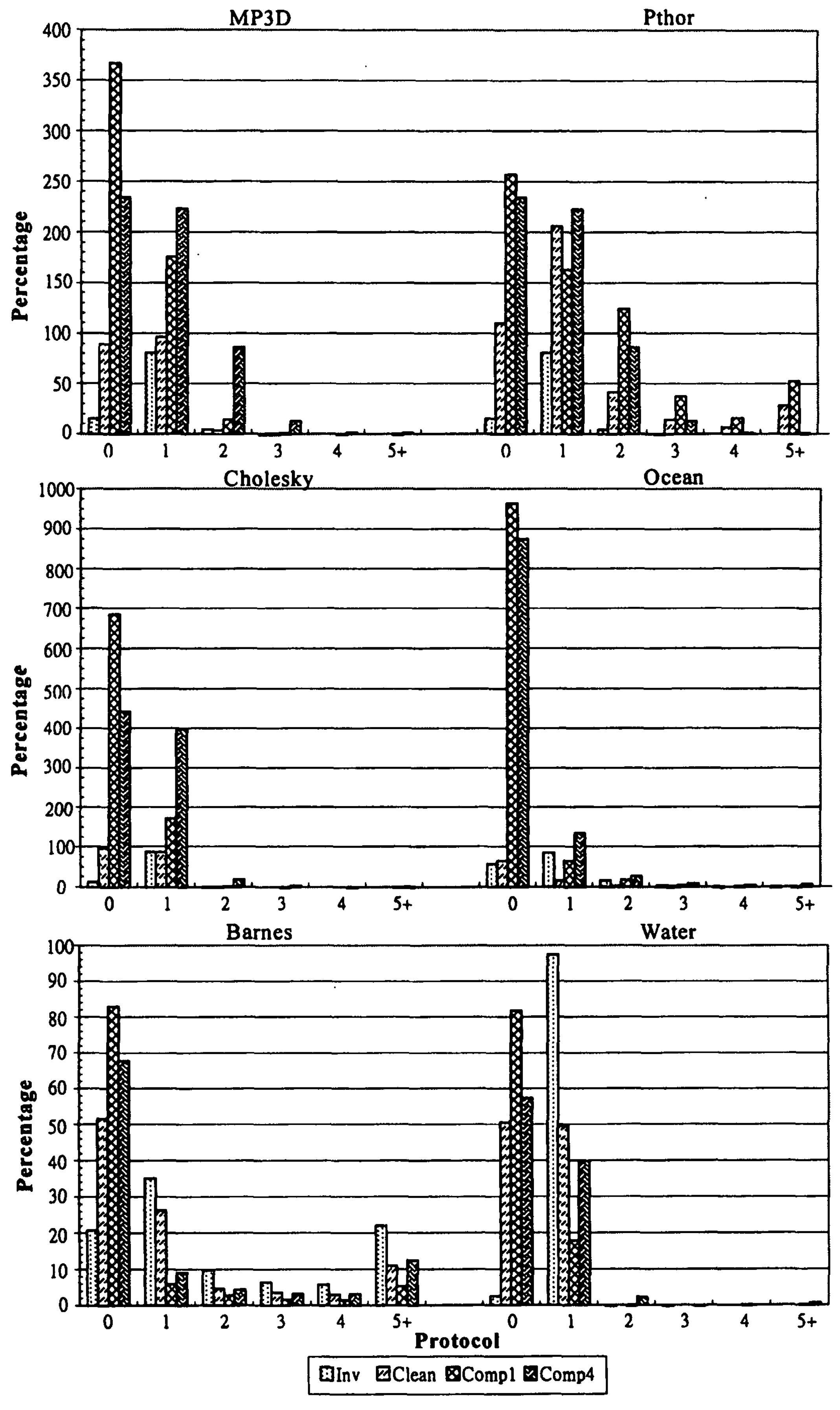

Fig. 3. Distribution of invalidation/update count.

one thread at any time and these threads share only one cache, there is a higher probability that several caches will have the same block at the same time. This increases write latency because the memory controller has to invalidate or update all other cached copies for each write request. This is the primary factor that degrades the performance of competitive-update protocols in PMP-MP systems.

We next compare the distributions of invalidation/update counts for different protocols, as shown in Fig. 3. The percentages were derived using the following formula:

$$\frac{\text{the number of write accesses issuing } \lambda \text{ invalidation/ update requests}}{\text{the total number of write accesses in the Inv protocol}}$$

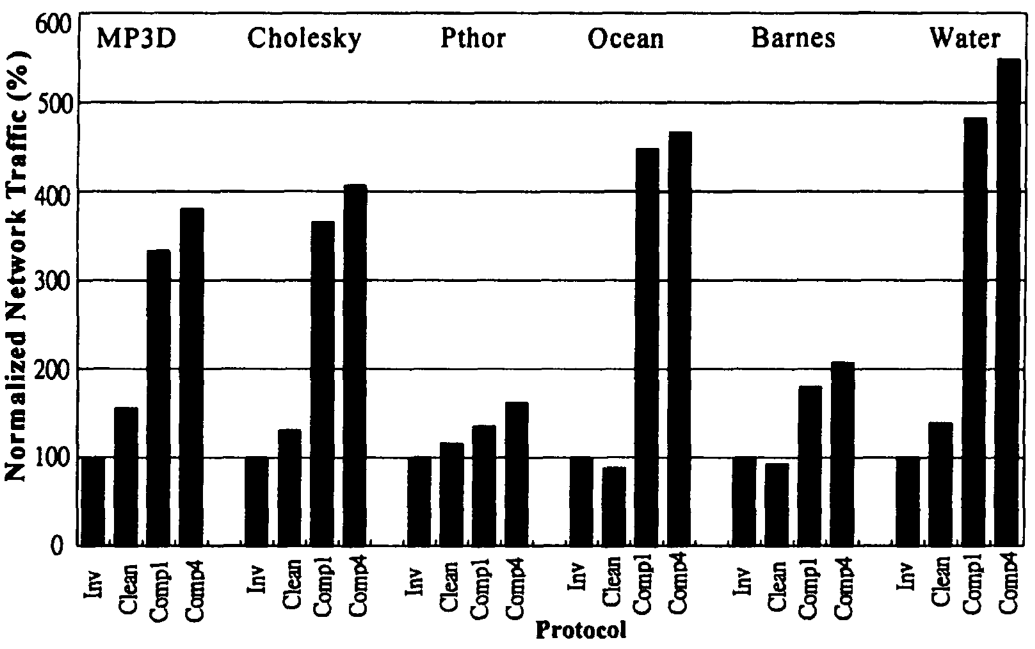

Fig. 4. Normalized network traffic for various cache coherence protocols.

The formula considers not only the invalidation/update count distribution for each protocol but also the amount of network traffic relative to the Inv. If λ equals zero, there are no other cached copies and we only have to update the memory copy. These write requests are completed as soon as the cache controller receives a reply from the memory controller. For the Inv protocol, these requests are issued to memory because of write misses or requests for ownership, thus they are necessary. However, other protocols have to send a write request to update the memory copy even though the requesting processor has the only cached copy. Consequently, a lot of these requests are wasted because no other cache will access them. Analysis shows that the competitive-update protocol has a much higher percentage of count-zeros than the write-invalidate protocol. Another interesting observation is that Compl had a higher percentage of count-zeros than the Comp4 because there is a probability that at Compl cached copies will be invalidated because the local processor did not access the updated copy when receiving a new update request, rather than four new update requests.

On the other hand, when λ is larger than zero, these write accesses have to issue λ requests to other caches. The memory controller has to wait for all the invalidation/update acknowledgments from cache controllers before it can send a reply message back to the requesting cache controller. As a result, not only is network traffic increased but write latency is also lengthened. The situation happens much more frequently with competitive-update protocols; Comp4 in particular always had longer update counts for all programs.

Figure 3 shows only the network traffic for write requests, therefore, we need another diagram to show the total network traffic including read and write requests. Competitive-update protocols have much higher traffic than Inv and Clean protocols, as shown in Fig. 4. This contradicts the philosophy of competitive-update protocols: reducing read penalties at the expense of some increase in network traffic. In other words, the performance of competitive-update protocols deteriorates to nearly that of write-update protocols in PMP-MP architectures.

Clean, on the other hand, only increases network traffic a little more than Inv for four applications (MP3D, Cholesky, Pthor, and Water). Surprisingly, it has less traffic than Inv for Ocean and Barnes. Like the write-invalidate protocol, Clean invalidates all cached copies. Unlike the write-invalidate protocol, the memory is kept clean until a write request arrives from the only node that has a cached copy. Clean has the disadvantage of having to send an additional message to invalidate the memory copy if the cached copy has become the only one after the last write. However, its advantage is that read requests can be served directly from memory. With the write-invalidate protocol, the memory has to redirect read requests to the dirty cache if the memory copy is invalidated. That is, in such a case, Inv sends an additional message for each read request. If this happens frequently, Clean will have less traffic because Clean often reduces read penalties at the expense of some increase in network traffic. It outperforms other protocols for all applications except MP3D.

In summary, the PMP-MP system has a higher probability that more than one cache will have the same block at the same time because several threads are executing simultaneously on each processing element. Therefore, competitive-update protocols have much higher network traffic in PMP-MP

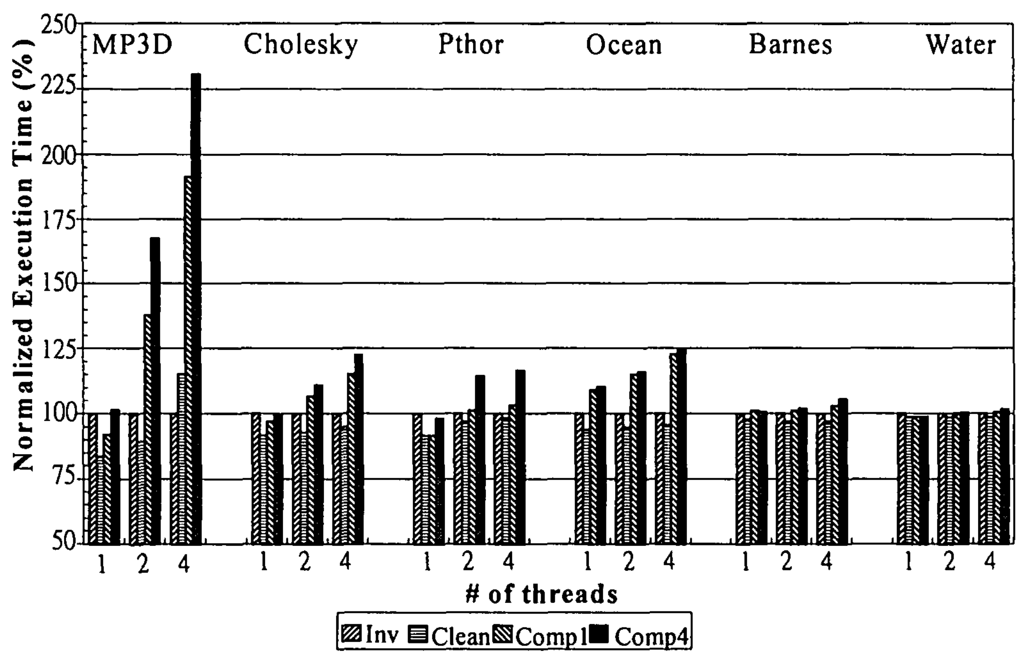

Fig. 5. Normalized execution time for various cache coherence protocols under PMP-MPs with various numbers of threads per processing element.

systems than in ordinary multiprocessor systems.

In the following, we examine the different effects on total performance of various numbers of threads for each processing element in Fig. 5. The performance difference between different protocols becomes larger when the number of threads is increased. In particular, the competitive-update protocol is not suitable for the PMP-MP with more threads per PE. However, for the architecture with one thread per PE, i.e., the conventional MP, Comp1 is better than or equal to Inv except for Ocean. Though Grahn and Stenstrom concluded that the competitive-update protocol outperforms the write-invalidation and write-update protocols, we found that Clean is the best protocol for the MP system with non-multithreaded processing elements. (Clean was not evaluated in [9].) As the number of threads per PE increases, the speedup that Clean has over Inv is reduced. Moreover, Inv is better than Clean for MP3D when each PE has four threads. Clean has the disadvantage of having to send an additional message to invalidate the memory copy if the cached copy has become the only one after the last write. Once the amount of additional message becomes large enough, Clean is worse than Inv. For the clean protocol, the normalized network traffic for MP3D for 1, 2, and 4 threads per PE are 141%, 149%, and 155%, respectively. Consequently, Clean is not the best protocol for MP3D for the PMP-MP system with 4 threads per PE. Nevertheless, Clean still outperforms the other protocols for most application programs under different architectures.

V. EFFECTS OF ADDING WRITE CACHES

Write caches can merge several write requests to the same block into a single request. Through cutting the number of write requests, write caches can enhance the performance of cache coherence protocols. As reported in [4], after adding write caches, competitive-update protocols are superior to any other protocol for multiprocessors with non-parallel-multithreaded processing elements. We investigate their effect on PMP-MP systems in this section.

First, we study the distribution of write run lengths to understand the characteristics of individual application programs. Write run length indicates how many global write accesses have been executed by the local processor without any intervening read or write accesses from other processors. During the interval, there is no need to update other copies because they will not be accessed. Therefore, the more long write run lengths, the better the performance of write-invalidate based protocols. The distribution of write run lengths for a PMP-MP architecture without write caches is shown in Fig. 6.

After we incorporate write caches in a multiprocessor system, the distribution of write run lengths is obviously changed as shown in Fig. 7. The percentage of write runs of length-one is significantly increased. A write run length equal to one means that a write is followed by an access from another processor before the original processor writes to it again. If we update instead of invalidate other cached copies, the followed access will hit the block, and thus reduces the access latency. Consequently, such a new distribution of write run lengths is more advantageous for competitive-update protocols. However, we must note that there are still 20% to 40% write run lengths larger than one, which would degrade the performance of competitive-update protocols.

The distributions of write run lengths in write caches is depicted in Fig. 8. Write run lengths in write

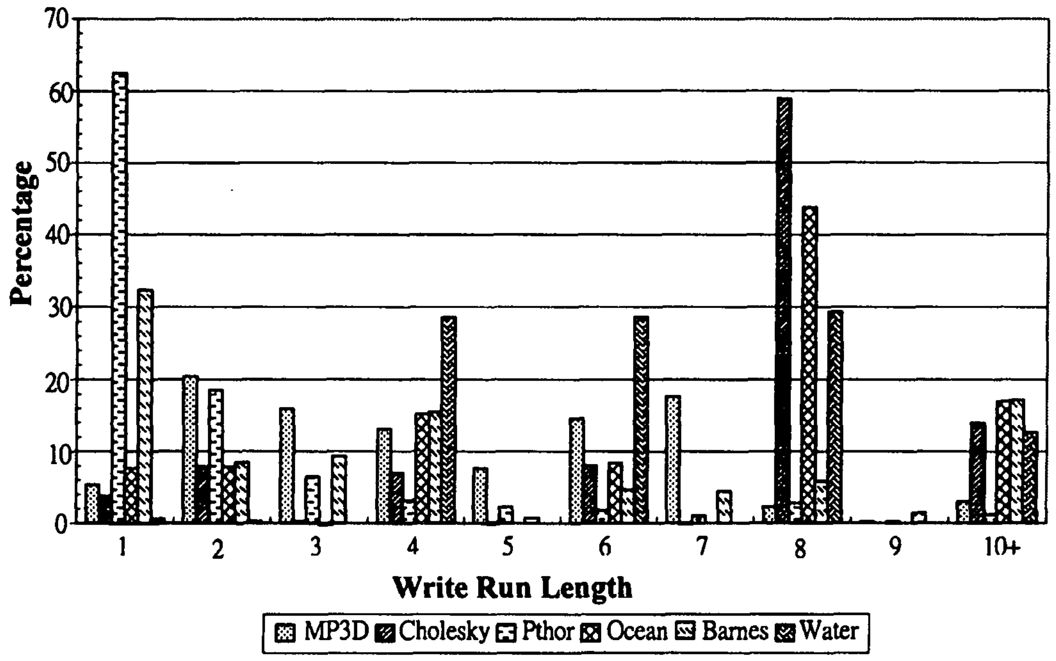

Fig. 6. Distribution of write run length for systems without write caches.

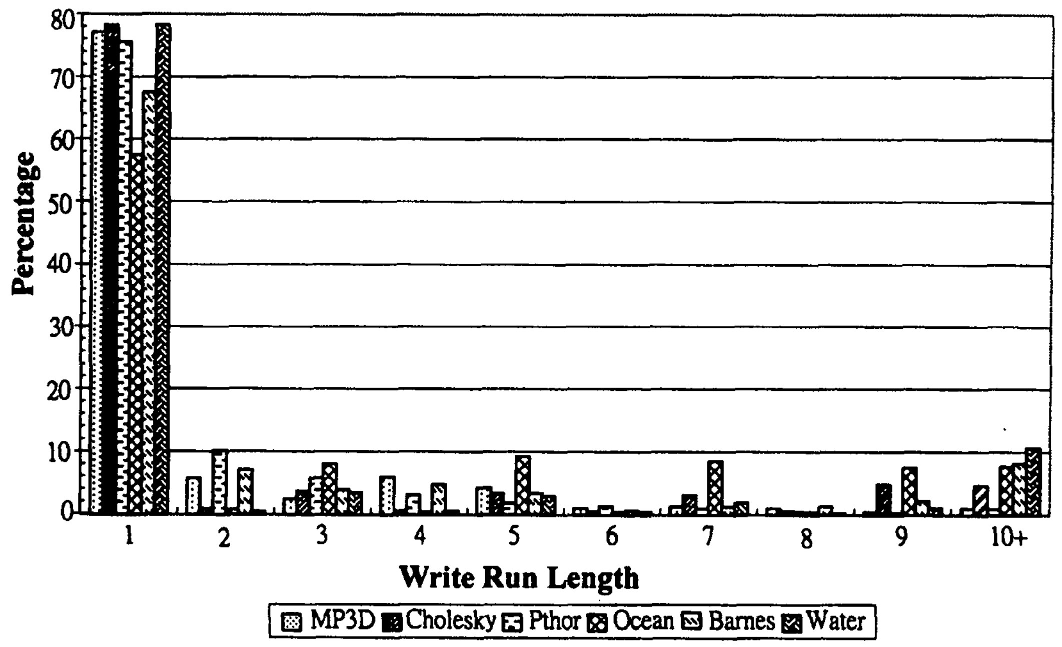

Fig. 7. Distribution of write run length for systems with write caches.

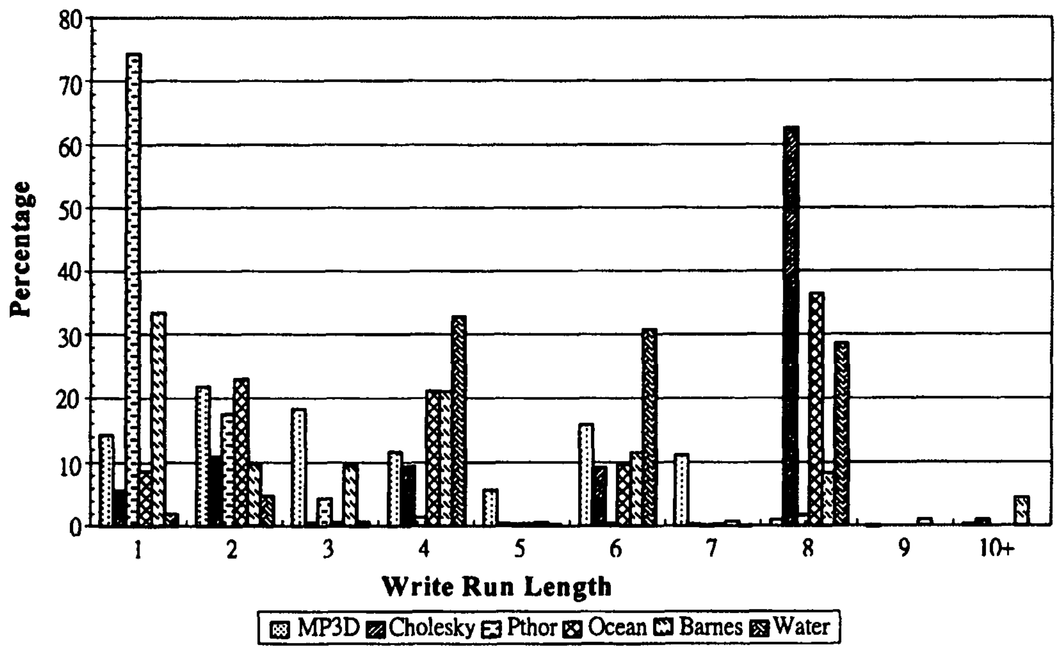

Fig. 8. Distribution of write run lengths in write caches.

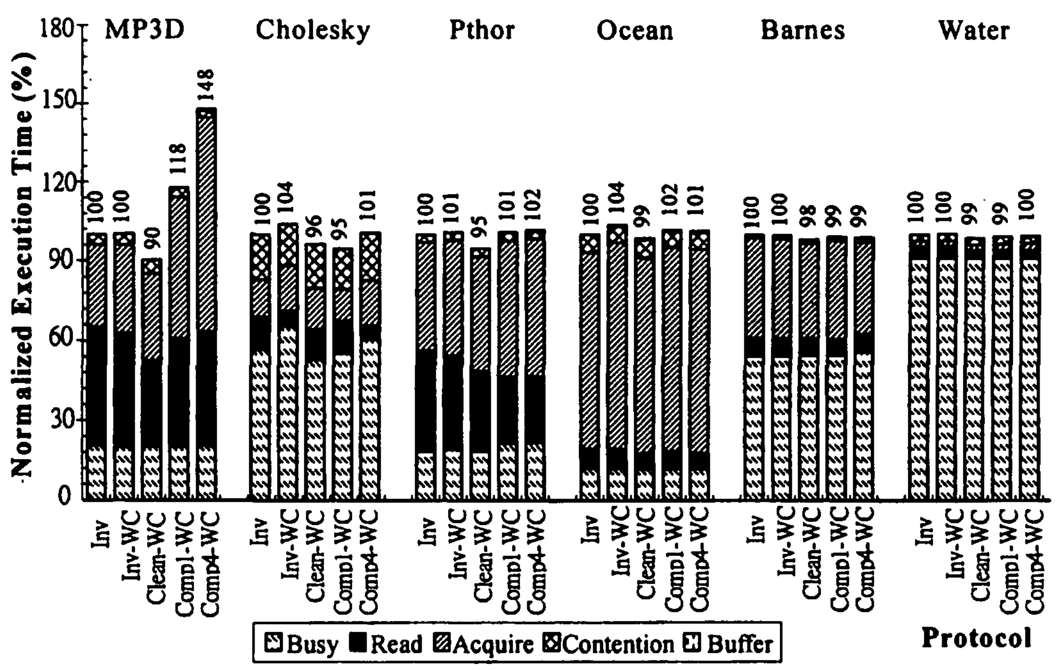

Fig. 9. Normalized execution times for various cache coherence protocols after augmentation with write caches.

caches represent how many write requests to the same block are combined into single ones. Write runs of length one indicate no merging. Therefore, Pthor will have a lower performance gain after addition of write caches because of its larger percentage of length-one runs. On the other hand, the larger the write run length, the more write requests are reduced. That is, the function of write caches is exploited much more fully if write run lengths are larger.

Figure 9 shows the performance improvement after adding write caches to write-invalidate (Inv-WC), clean (Clean-WC), and competitive-update with threshold T (CompT-WC) protocols. The most important observation is that Clean-WC outperformed Inv for all applications. The advantage of the Clean-WC protocol is short read-penalties with slight increase in network traffic. On the other hand, Inv-WC and Comp-WC were generally worse than Inv although the performance difference was small for three of the applications (Cholesky, Pthor, and Ocean). Inv-WC had longer acquire-stall times than Inv for the following reason.

Inv-WC delays the issuance of global writes in write caches until it encounters a synchronization point. The synchronization access then has to wait for the flushing of the write cache. In particular, release accesses cannot be issued until the write requests have all been flushed from the write cache and have all been performed. This waiting keeps other processes from entering the critical section and thus increases the acquire-stall time. On the other hand, the write-invalidate protocol without write caches sends only the first in a sequence of writes to the same block to the memory controller. Once the write-miss reply is returned, (which means the cache controller has acquired ownership of that block), all other write misses are deleted after the block is updated according to the write ordering. The effect of this procedure is similar to the function of write caches: merging all write requests to the same block into a single one. As a result, adding write caches is not beneficial for the write-invalidate protocol.

For competitive-update protocols, each write request definitely incurs a global write. This feature has two extreme effects on write caches. First, write caches can cut considerable network traffic by merging write requests, therefore, performance can be substantially enhanced. Second, there are usually many write requests to be flushed from the write cache whenever a synchronization access is encountered, thus acquire-stall time is increased. We see from Fig. 9 that the second effect dominates the results. Competitive-update protocols have higher acquire-stall time though the network traffic is drastically reduced, as shown in Fig. 10.

In the following, we examine the different effects on total performance of various numbers of threads for each processing element in Fig. 11. Compared with Fig. 5, write caches improve performance significantly for the clean and the competitive-update protocols. For the system with one thread per PE, the competitive-update protocol with write caches outperforms the write-invalidation protocol. However, when the number of threads per PE is increased, the competitive-update protocol with write caches may be worse than Inv. On the other hand, Clean-WC outperforms Inv for all application programs for architectures with various numbers of threads per PE. Moreover, because the competitive-update protocol requires one counter per cache block, the clean protocol is more suitable for PMP-MPs after considering cost/performance tradeoff.

Table 3. Write cache hit ratio for two different mapping policies

	MP3D	Cholesky	Pthor	Ocean	Barnes	Water
Direct-mapped	72.64%	84.15%	32.38%	80.82%	74.47%	82.11%
Fully-associative	74.85%	84.51%	32.41%	82.41%	75.73%	82.30%

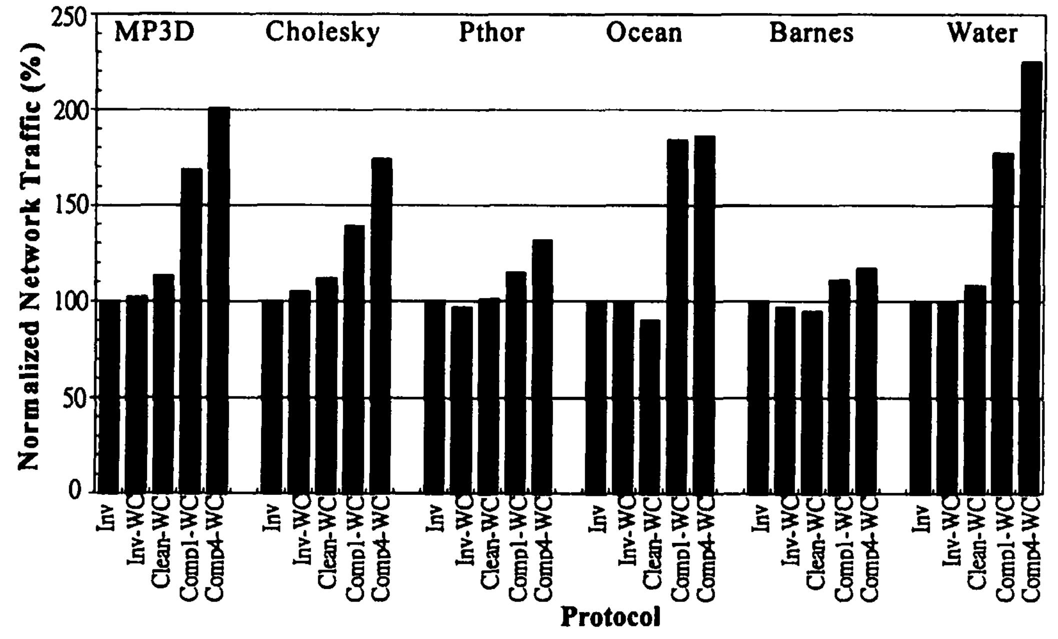

Fig. 10. Normalized network traffic for various cache coherence protocols after augmentation with write caches.

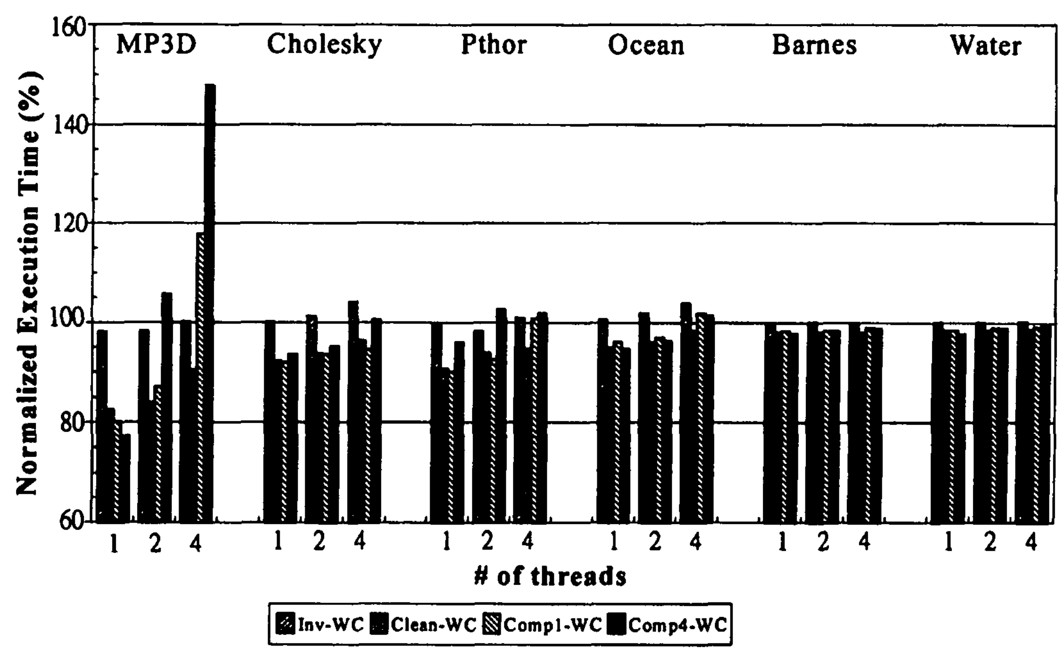

Fig. 11. Normalized execution times for various cache coherence protocols with write caches for various numbers of threads per PE.

VI. IMPACTS OF ARCHITECTURE PARAMETERS

So far, we have assumed that the write cache mapping policy is direct-mapped. Below, we study what occurs when the mapping policy is fully associative with an LRU replacement policy. We use the suffix "-full" to represent the new architecture. Moreover, because Clean-WC outperforms Inv for all application programs, we focus only on this kind of architecture.

Table 3 illustrates the write cache write hit- ratios for two different mapping policies. As expected, the fully-associative policy had higher hit ratios. However, Clean-WC-full was slightly better than Clean-WC on only four programs because of longer acquire-stall times as shown in Fig. 12. Like the effect of augmenting the competitive-update protocol with write caches, a higher hit ratio for write caches means more write requests are merged, and thus network traffic is reduced. However, acquire-stall time is probably increased because write caches must be flushed. Fully-associative write caches do not reduce network traffic much

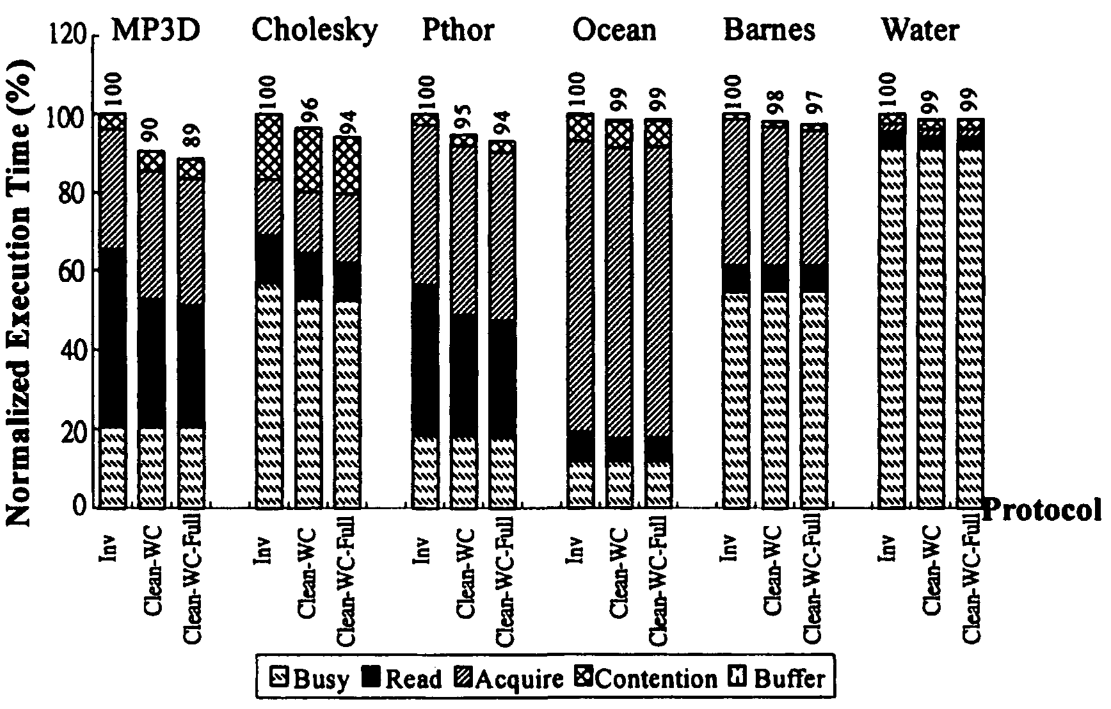

Fig. 12. Normalized execution times for two different mapping policies using the clean cache coherence protocol.

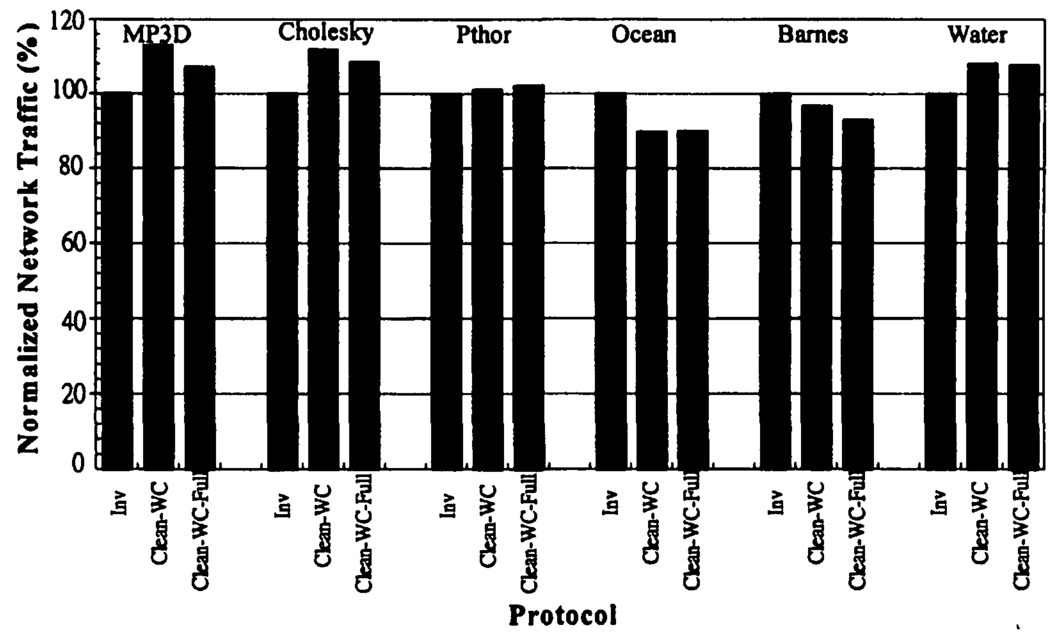

Fig. 13. Normalized network traffic for two different mapping policies using the clean cache coherence protocol.

more than direct-mapped write caches, as shown in Fig. 13. Consequently, the normalized execution times for Clean-WC and Clean-WC-full are almost the same.

We also observed that read-stall times were reduced slightly in Clean-WC-full systems because of higher hit ratios. Because more write requests are buffered in write caches, other cached copies will not be invalidated too early. As a result, other processors will hit the block before invalidation requests arrive. Reducing read-stall times is why the fully associative policy was better.

Size is a critical factor in write cache design because it affects the hardware cost. The disadvantage of a small write cache is that there are fewer write requests to merge. On the other hand, a large write cache will increase hardware cost. The execution times for different write cache sizes, ranging from four to thirty-two blocks, are illustrated in Fig. 14. Performance does not improve along with the increase in size; on the contrary, larger sizes may worsen performance because too many write requests must be flushed from write caches when a synchronization access is encountered. Release accesses cannot be issued to the memory system before all write requests have been flushed and performed. Because of this delay in release access, acquire accesses from other processors have to wait longer to enter the critical section. Consequently, not only is the acquire-stall time increased but the read-stall time is also lengthened because an acquire access is a read request.

The above reason also explains why Dahlgren and Stenstrom [4] suggested that write caches be

Table 4. Comparison of suggestions for conventional-MP and PMP-MP systems

	Conventional MP		PMP-MP
	Stenstrom et al.	Ours	
Protocol for systems without write caches	Com4	Clean	Clean
Protocol for systems with write caches	Compl-WC or Clean-WC	Clean-WC	Clean-WC
Placement policy for write caches	Direct-mapped	Direct-mapped	Direct-mapped
Size of write cache	4 blocks	4 blocks	8 blocks

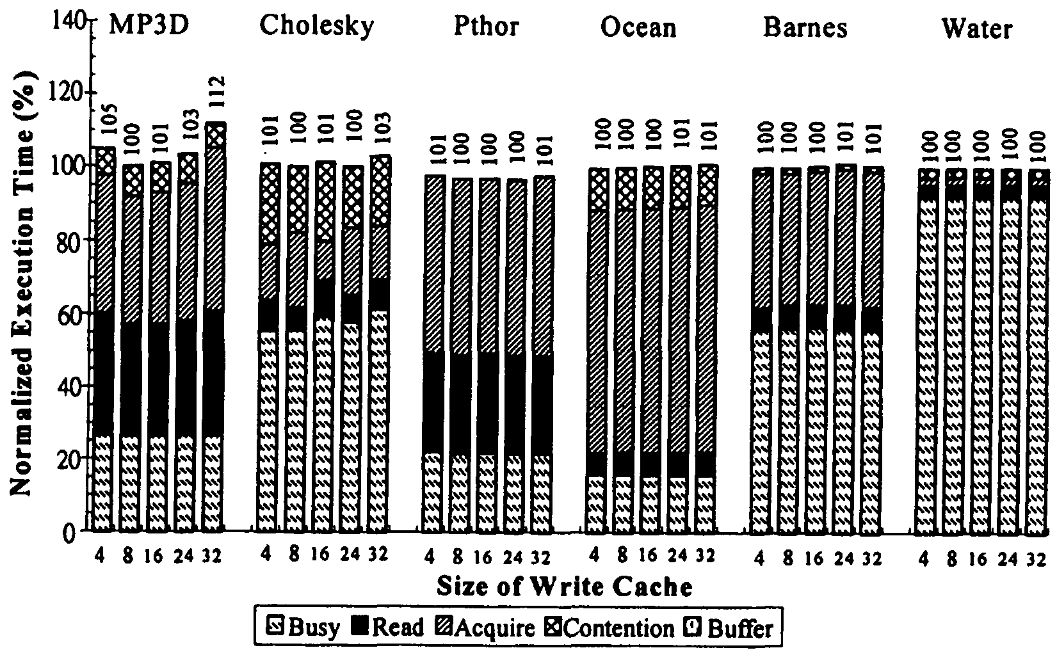

Fig. 14. Normalized execution times for different write cache sizes using the clean cache coherence protocol.

flushed when encountering an acquire access request. The purpose is to prevent write caches from buffering too many write requests when a release access request arrives. In fact, we have no need to flush write caches when encountering an acquire access according to the release consistency model. An acquire access can be performed before its pervious writes in the program order have all become visible to all processor nodes.

VII. CONCLUDING REMARKS

Table 4 shows a comparison of suggestions for conventional-MP and PMP-MP systems. Previous research has reported that directory-based competitive-update protocols are superior to other protocols for conventional multiprocessor systems using relaxed consistency models [9] because their directory structures allow parallel updating, and relaxed consistency models hide write latencies. Moreover, competitive-update protocols are successful in reducing read penalties at the expense of some increase in write traffic. Another important observation is that incorporating write caches improves the performance of clean and competitive-update protocols because write caches can merge several write requests to the same block into single ones, thus reducing network traffic [4].

However, we found different results according to our simulation results. The clean protocol outperforms the competitive-update protocol for most programs. We have to note that the clean protocol was not evaluated in the study [9]. On the other hand, although Compl-WC and Comp4-WC may be better than Clean-WC for some programs, we suggest Clean-WC for conventional MP after considering cost/performance tradeoff. The competitive-update protocol requires one counter per cache block.

In addition, we also found different results when processing elements were replaced by parallel-multithreaded processors. Because more than one thread are executed simultaneously in every processing element, global memory accesses from different threads will interfere with each other. Furthermore, it is more likely that several processing elements will have cached the same block at the same time. According to our simulation results, the clean protocol performed best for five out of six programs. After augmentation with write caches, the clean protocol was still the best for all applications. Though the competitive-update protocol also showed improved performance, it was not better than the write-invalidate protocol for most programs.

Our simulations showed that merging too many write requests has a negative effect. The issuance of release accesses is delayed until all write requests have been flushed from write caches and have all been performed. Consequently, read-stall times and acquire-stall times are both increased. Therefore, write cache functions are exploited most fully when modest numbers of write requests are merged. For this reason, increasing the sizes of write caches and adopting fully-associative mapping policies are not certain to improve performance. Considering the hardware cost, we suggest using direct-mapped policies and 8-block write caches.

The results presented in this paper were obtained from simulations made under several assumptions on architectural parameters. In fact, varying the parameters may influence the results. Therefore, in the future, we will study the performance implications of using different parameter values, including the number of threads per processing element, the bandwidth of the interconnection network, and the number of write buffer entries.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their helpful comments. This research was supported by the National Science Council of the Republic of China under contract number: NSC 86-2213-E009-094.

REFERENCES

1. Agarwal, A., and Gupta, A., "Memory-reference Characteristics of Multiprocessor Applications under MACH," Proceedings of the 15th International Symposium on Computer Architecture, pp. 215-225, (1988).
2. Boyle, J., Bulter, R., Disz, T., Glickfeld, B., Luck, E., Overbeek, R., *Patterson, J., and Stevens, R., Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., New York, (1987).
3. Bray, B.K., and Flynn, M.J., *Write-caches as an alternative to write buffers*, Computer Systems Laboratory Tech. Rep. CSL-TR-91-470, Stanford University, USA, (1991).
4. Dahlgren , F. and Stenstrom, P., "Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 26, pp. 193-210, (1995).
5. Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 434-442, (1986).
6. Eggers, S.J., and Katz, R.H., "Evaluating the Performance of Four Snooping Cache Coherence Protocols," Proceedings of the 16th International Symposium on Computer Architecture, pp. 1-15, (1989).
7. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," Proceedings of the 17th International Symposium on Computer Architectures, pp. 15-26, (1990).
8. Goodman, J.R., *Cache consistency and sequential consistency*, Technical Report No. 61, SCI Committee, (1989).
9. Grahn, H., Stenstrom, P., and Dubois, M., "Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models," *Future Generation Computer Systems*, Vol. 11, pp. 247-271, (1995).
10. Hirata, H., Kimura, K., Nagamine, S., and Mochizuki, Y., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," Proceedings of the 19th International Symposium on Computer Architecture, pp. 136-145, (1992).
11. Iannucci, R.A., Gao, G.R., Halstead, R.H., Smith, Jr., *Multithreading: A Summary of the State of the Art*, Kluwer Academic Publishers, (1993).
12. Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs," *IEEE Transactions on Computers*, Vol. 28, No. 9, pp. 241-248, (1979).
13. Singh, J.P., Weber, W-D., and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, Vol. 20, No. 1, pp. 5-44, (1992).
14. Stenstrom, P., Dahlgren, F., and Lundberg, L., "A Lockup-free Multiprocessor Cache Design," Proceedings of 1991 International Conference on Parallel Processing, Vol. 1, pp. 246-250, (1991).
15. Veenstra, J.E. and Fowler, R.J., *MINT Tutorial and User Manual*, Technical Report No. 452, The University of Rochester, New York, USA, (1994).
16. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Considerations,"

Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24-36, (1995).

Discussions of this paper may appear in the discussion section of a future issue. All discussions should be submitted to the Editor-in-Chief.

Manuscript Received: Jan. 21, 1997
Revision Received: June 12,1997
and Accepted: Sep. 18, 1997

快捷記憶體一致性協定與寫入快捷記憶體在平行多引線共享記憶體多處理機系統中之效能研究

伍朝欽　陳正

國立交通大學資訊工程研究所

摘　要

根據以往之研究，在傳統的多處理機系統中若是採用循序記憶體一致性模式，則沒有任何一種目錄式的快捷記憶體一致性協定會讓所有的程式都有最好的執行效果。然而最近的研究結果指出若是採用較鬆散的記憶體一致性模式，則競爭式更新一致性協定能提供最好的效果。另外，在系統中加入寫入快捷記憶體能增加clean 與競爭式更新這兩種一致性協定的執行效能。

在本論文中，我們將探討如果將處理器取代成平行多引線架構時，會對快捷記憶體一致性協定與寫入快捷記憶體產生何種不同的效果。根據六個SPLASH的標竿程式評估後，我們發現有五個程式在clean一致性協定下會有最佳之效能；而當加入寫入快捷記憶體後，則clean能提供所有標竿程式最好的效果。雖然競爭式更新一致性協定的效能也能改善，但是對大多數的程式而言，其效能仍不及寫入無效的一致性協定。

關鍵詞：寫入快捷記憶體，多引線處理器，共享記憶體多處理機，快捷記憶體一致性協定。