Fourth Workshop on Scalable Shared Memory Multiprocessors,
Chicago, IL, April 1994

Scalable Atomic Primitives for Distributed Shared Memory Multiprocessors

(Extended Abstract)

Maged M. Michael
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
USA

Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
USA

Abstract

Our research addresses the general topic of atomic update of shared data structures on large-scale shared-memory multiprocessors. In this paper we consider alternative implementations of the general-purpose single-address atomic primitives `fetch_and_Φ`, `compare_and_swap`, `load_linked`, and `store_conditional`. These primitives have proven popular on small-scale bus-based machines, but have yet to become widely available on large-scale, distributed shared memory machines. We propose several alternative hardware implementations of these primitives, and then analyze the performance of these implementations for various data sharing patterns. Our results indicate that good overall performance can be obtained by implementing `compare_and_swap` in the cache controllers, and by providing an additional instruction to load an exclusive copy of a cache line.

1 INTRODUCTION

Distributed shared memory combines the scalability of network-based architectures and the intuitive programming model of shared memory. To ensure the consistency of shared objects, processors perform synchronization operations using hardware-supported primitives. Synchronization overhead (especially for atomic update) is consequently one of the major obstacles to scalable performance on shared memory machines.

Several atomic primitives have been proposed and implemented on DSM architectures. Most of them are special-purpose primitives designed to support particular synchronization operations. Examples include `test_and_set` with special semantics on the Stanford DASH [9], the QOLB primitives of the Wisconsin Multicube [3] and the IEEE Scalable Coherent Interface [10], the full/empty bits of the MIT Alewife [1], and the primitives for locking and unlocking cache lines on the Kendall Square KSR1 [8].

General-purpose primitives such as `fetch_and_`Φ, `compare_and_swap`, and the pair `load_linked/store_conditional` can easily and efficiently implement a wide variety of styles of synchronization (e.g. operations on wait-free and lock-free objects, read-write locks, priority locks, etc.). These primitives are easy to implement in the snooping protocols of bus-based multiprocessors, but there are many tradeoffs to be considered when developing implementations for a DSM machine. `Compare_and_swap` and `load_linked/store_conditional` are not provided by any of the major DSM multiprocessors, and the various `fetch_and_`Φ primitives are provided by only a few.

We propose and evaluate several implementations of these general-purpose atomic primitives on directory-based cache coherent DSM multiprocessors, in an attempt to answer the question: which atomic primitives should be provided on future DSM multiprocessors and how should they be implemented?

Our analysis and experimental results suggest that the best overall performance will be achieved by `compare_and_swap`, with comparators in the caches, a write-invalidate coherence policy, and an auxiliary `load_exclusive` instruction.

In section 2 we present several implementation options for the primitives under study on DSM multiprocessors. Then we present our experimental results and discuss their implications in section 3, and conclude with recommendations and future directions in section 4.

2 IMPLEMENTATIONS

The main design issues for implementing atomic primitives on cache coherent DSM multiprocessors are:

1. Where should the computational power to execute the atomic primitives be located: in the cache controllers, in the memory modules, or both?
2. Which coherence policy should be used for atomically accessed data: no caching, write-invalidate, or write-update?
3. What auxiliary instructions, if any, can be used to enhance performance?

We consider three implementations for each of the primitives. The implementations are categorized according to the coherence policy used for atomically-access data (the same or a different policy could be used for other data):

1. EXC (EXClusive): Computational power located in the cache controllers, with a write-invalidate coherence policy. The main advantage of this implementation is that once the data is in the cache, subsequent atomic updates are executed locally, so long as accesses by other processors do not intervene.

2. UPD (UPDate): Computational power located at the memory, with a write-update policy. The main advantage of this implementation is a high read hit rate, even in the case of alternating accesses by different processors.
3. NOC (NO Caching): Computational power located at the memory, with caching disabled. The main advantage of this implementation is that it eliminates the coherence overhead of the other two policies, which may be a win in the case of high contention or even the case of no contention when updates by different processors alternate.

For `fetch_and_`Φ and `compare_and_swap`, EXC obtains an exclusive copy of the data and performs the operation locally. NOC sends a request to the memory to perform the operation on uncached data. UPD also sends a request to the memory to perform the operation, but retains a shared copy of the data in the local cache. The memory sends updates to all the caches with copies.

In the EXC implementation of `load_linked/store_conditional`, each processing node has a reservation bit and a reservation address register, which function much as they do in implementations for bus-based machines. `Store_conditional` fails if the reservation bit is invalid. It succeeds locally if the bit is valid and the line is cached exclusively. Otherwise it sends a request to the home node. If the directory indicates that the line is exclusive or uncached, `store_conditional` fails, otherwise (the line is shared) `store_conditional` succeeds, an exclusive copy is acquired, and invalidations are sent to the holders of other copies.

In the NOC implementation of `load_linked/store_conditional`, each memory location (at least conceptually) has a reservation bit vector of size equal to the total number of processors. `Load_linked` reads the value from memory and sets the appropriate reservation bit to valid. Any write or successful `store_conditional` to the location invalidates the reservation vector. `Store_conditional` checks the corresponding reservation bit and succeeds or fails accordingly. Various space optimizations are conceivable for practical implementations. In the UPD implementation, `load_linked` requests have to go to memory even if the data is cached, in order to set the appropriate reservation bit. Similarly, `store_conditional` requests have to go to memory to check the reservation bit.

We consider two auxiliary instructions. `Load_exclusive` reads a datum but acquires exclusive access. It can be used with EXC instead of an ordinary `atomic_load` when reading data that is then accessed by `compare_and_swap`. The intent is to make it more likely that `compare_and_swap` will not have to go to memory. `Load_exclusive` is also useful for ordinary operations on migratory data. `Drop_copy` can be used to drop (self-invalidate) a cached datum, to reduce the number of serialized messages required for subsequent accesses by other processors.

3 EXPERIMENTAL RESULTS

The experimental results were collected using an execution-driven simulator that uses MINT [13] as a front end. The back end simulates a 64-node multiprocessor with directory-based caches, 32-byte blocks, memory that queues conflicting accesses, and a 2-D worm-hole mesh network.

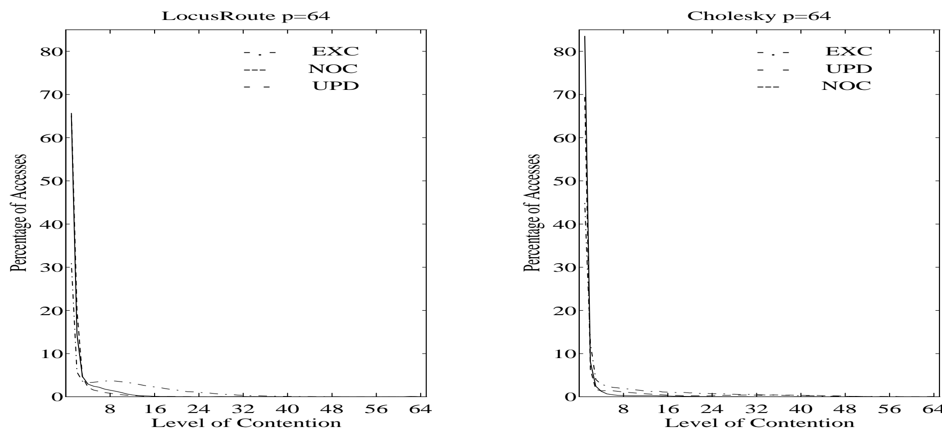

Figure 1: Histograms of the level of contention in LocusRoute and Cholesky.

We used two sets of applications, real and synthetic, to achieve different goals. We studied two lock-based applications from the SPLASH suite [11]—LocusRoute and Cholesky—in order to identify typical sharing patterns of atomically accessed data. The synthetic applications—lock-free, test-and-test-and-set lock-based, and MCS lock-based access to a counter—served to explore the parameter space and to provide controlled performance measurements.

3.1 SHARING PATTERNS

Performance of atomic primitives is affected by two main sharing patterns, contention and average write-run length [2]. In this context, the level of contention is the number of processors that concurrently try to access an atomically accessed shared location. Average write-run length is the average number of consecutive writes (including atomic updates) by a processor to an atomically accessed shared location without intervening accesses (reads or writes) by any other processors.

Our experiments indicate that the average write run length for atomically accessed data in LocusRoute and Cholesky is very small: on 64 processors with different coherence policies it ranged from 1.59 to 1.83. Figure 1 confirms the expectation that the no-contention case is the common one, for which performance should be optimized. At the same time, it indicates that the low and moderate contention cases do arise, so that performance for them needs also to be good. High contention is rare: reasonable differences in performance among the primitives can be tolerated in this case.

3.2 RELATIVE PERFORMANCE OF IMPLEMENTATIONS

Figure 2 shows the performance results for the lock-free counter application. (This application uses a `fetch_and_increment` instruction or an `atomic_load/compare_-and_swap` or `load_linked/store_conditional` pair to increment a counter; results for the other synthetic applications imply the same conclusions.) The bars represent the elapsed time averaged over a large number of counter updates. The graphs to the left represent the no-contention case with different numbers of consecutive

Table 1: Serialized network messages for a lock-free shared counter update without contention.

	faΦ	ll/sc	cas	ldx/cas
EXC to cached exclusive	0	0	0	0
EXC to remote exclusive	4	6	6	4
EXC to uncached	2	4	4	2
UPD to cached	3	5	3	-
UPD to uncached	2	4	4	-
NOC	2	4	4	-

accesses by each processor without intervention from the other processors. The graphs to the right represent different levels of contention. The bars in each graph are categorized according to the three coherence policies used in the implementation of atomic primitives. In EXC and UPD, there are two subsets of bars. The bars to the right represent the results when using the `drop_copy` instruction, while those to the left are without it. In each of the two subsets in the EXC category, the two bars for `compare_and_swap` represent, from left to right, the results for the implementations without and with `load_exclusive`, respectively.

3.2.1 Coherence Policy

With no contention and short write runs, NOC implementations of the three primitives perform nearly as well as their corresponding cached implementations. There are two reasons for this result. First, a write miss on an uncached line takes two serialized messages, while a write miss on a remote exclusive or remote shared line takes 4 or 3 serialized messages respectively (see Table 1). Second, NOC does not incur the overhead of invalidations and updates as EXC and UPD do. Furthermore, with contention (even very low), NOC outperforms the other policies (with the exception of EXC `load_exclusive`/`compare_and_swap` when simulating `fetch_and_`Φ), as the effect of avoiding excess serialized messages, and invalidations or updates, is more evident as ownership of data changes hands more frequently. The EXC `load_exclusive`/`compare_and_swap` pair for simulating `fetch_and_`Φ is an exception as the timing window between the read and the write in the read-modify-write cycle is narrowed substantially, thereby reducing the effect of contention by other processors. Also, in the EXC implementation, successful `compare_and_swap`'s after `load_exclusive`'s are mostly hits, while all NOC accesses are misses.

As write-run length increases, EXC increasingly outperforms NOC and UPD, because subsequent accesses in a run length are all hits. Comparing UPD to EXC, we find that EXC is always better in the common case of no and low contention. This is due to the excessive number of useless updates incurred by UPD.

3.2.2 Atomic Primitives

NOC `fetch_and_add` yields superior performance over the other primitives and implementations, especially with contention. The exception is the case of long write-runs, which are not the common case. We conclude that NOC `fetch_and_add`

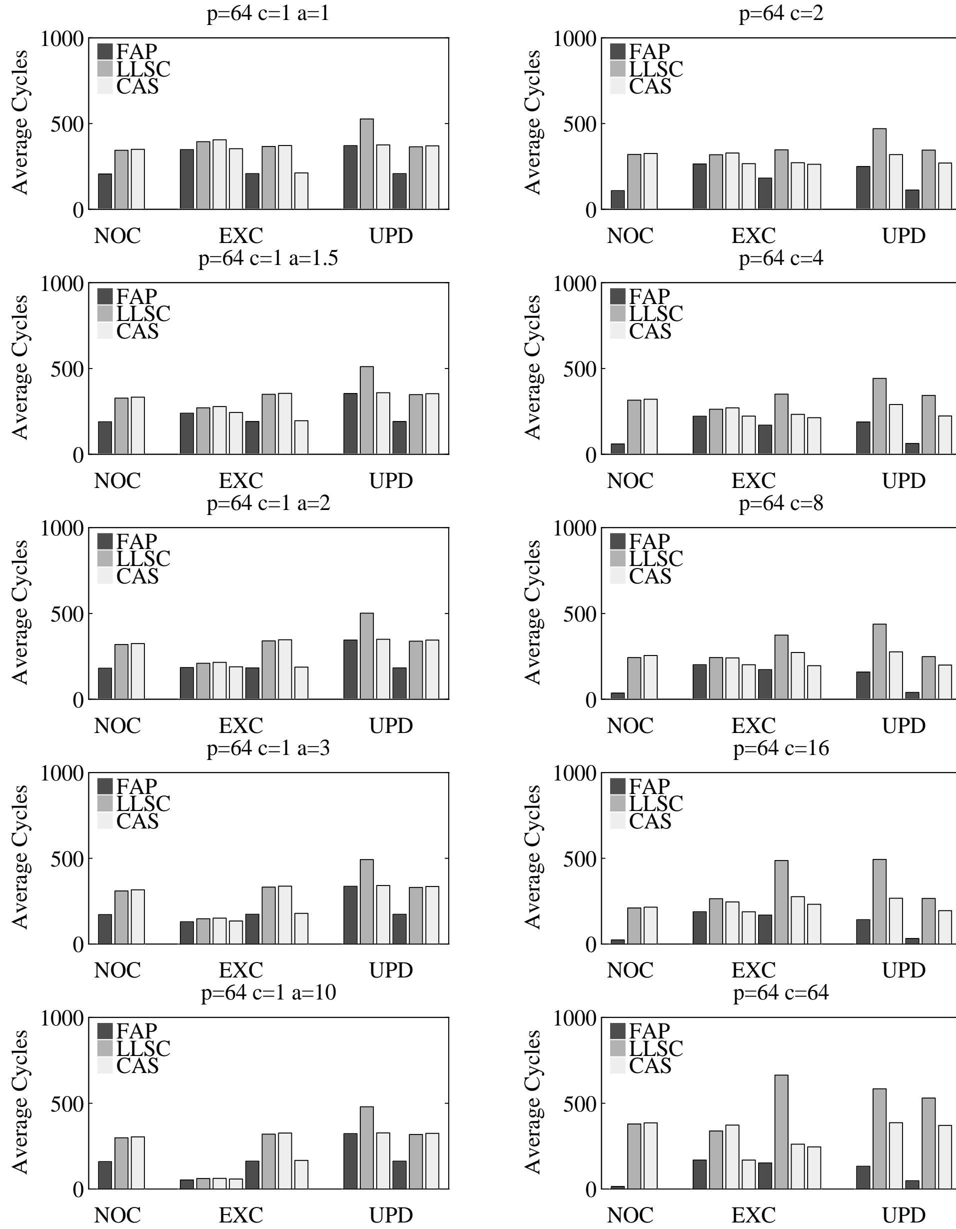

Figure 2: Average time per counter update for the lock-free counter application (p denotes processors, c contention, and a the average number of non-intervened counter updates by each processor).

is a useful primitive to provide for supporting shared counters and other special cases. Because it is limited to only certain kinds of algorithms, however, we recommend it only in addition to a universal primitive [6] (compare_and_swap or load_linked/store_conditional).

EXC compare_and_swap almost always benefits from load_exclusive, because compare_and_swap's are hits in the case of no contention and load_exclusive helps minimize the failure rate of compare_and_swap as contention increases. EXC load_linked cannot be designed to acquire an exclusive copy of data; otherwise livelock is likely to occur. UPD compare_and_swap is always better than UPD load_linked/store_conditional, as most of the time compare_and_swap is preceded by an ordinary read, which is most likely to be a hit with UPD. Load_linked requests have to go to memory even if the data is cached locally, as the reservation has to be set in a unique place that has the most up-to-date version of data—in memory in the case of UPD.

3.2.3 Auxiliary Instructions

Load_exclusive enhances the performance of EXC compare_and_swap in the common case of no contention. With an EXC policy and an average write-run length of one with no contention, drop_copy improves the performance of fetch_and_Φ and load_exclusive/compare_and_swap, because it allows the atomic primitive to obtain the needed exclusive copy of the data with only 2 serialized messages (requesting node to home and back) instead of 4 (requesting node to home to current owner to home and back to requesting node).

4 CONCLUSIONS

Based on the experimental results and the relative power of atomic primitives [6], we recommend implementing compare_and_swap in the cache controllers of future DSM multiprocessors, with a write-invalidate coherence policy. We also recommend supporting load_exclusive to enhance the performance of compare_and_swap, in addition to its benefits in efficient data migration. Finally, we recommend supporting drop_copy to allow programmers to enhance the performance of compare_and_swap/load_exclusive in the common case of no or low contention with short write runs. Although we do not recommend it as the sole atomic primitive, we find fetch_and_add to be useful with lock-free counters (and with many other objects [4]). We recommend implementing it in uncached memory as an extra atomic primitive.

Our plans for future research include extending this study to cover multiple-address atomic primitives such as transactional memory [5] and the Oklahoma update [12], and other alternatives for atomic update of multiple-address objects such as Herlihy's lock-free methodology [7], function shipping (active messages [14]), and special-purpose concurrent lock-based and lock-free implementations.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, New York, June 1990.

[2] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 1989.

[3] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[4] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Trans. on Programming Languages and Systems*, 5(2):164–189, April 1983.

[5] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 16-19, 1993.

[6] M. P. Herlihy. Wait-Free Synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[7] M. P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, November 1993.

[8] *KSR1 Principles of Operation.* Kendall Square Research Corporation, 1991.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[10] *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE, Inc., 1993.

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

[12] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

[13] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.

[14] T. von Eicken, D. E. Culer, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceeedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.