

Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors*

Maged M. Michael

Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY 14627-0226

{michael,scott}@cs.rochester.edu

July 1994

Abstract

Many hardware primitives have been proposed for synchronization and atomic memory update on shared-memory multiprocessors. In this paper, we focus on general-purpose primitives that have proven popular on small-scale bus-based machines, but have yet to become widely available on large-scale, distributed-memory machines. Specifically, we propose several alternative implementations of `fetch_and_Φ`, `compare_and_swap`, and `load_linked/store_conditional`. We then analyze the performance of these implementations for various data sharing patterns, in both real and synthetic applications. Our results indicate that good overall performance can be obtained by implementing `compare_and_swap` in a multiprocessor's cache controllers, and by providing an additional instruction to load an exclusive copy of a line.

Keywords: synchronization, scalability, fetch-and- Φ , compare-and-swap, load-linked, store-conditional, cache coherence

1 Introduction

Distributed shared memory multiprocessors combine the scalability of network-based architectures and the intuitive programming model provided by shared memory [21]. To ensure the consistency of shared data structures, processors perform synchronization operations using hardware-supported primitives. Synchronization overhead (especially atomic update) is one of the obstacles to scalable performance on shared memory multiprocessors.

*This work was supported in part by NSF grants nos. CDA-8822724 and CCR-9319445, and by ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

Several atomic primitives have been proposed and implemented on DSM architectures. Most of them are special-purpose primitives that are designed to support some particular style of synchronization operations. Examples include `test_and_set` with special semantics on the DASH multiprocessor [17], the QOLB primitives on the Wisconsin Multicube [6] and the IEEE Scalable Coherent Interface standard [24], the full/empty bits on the Alewife [1] and Tera machines [3], and the primitives for locking and unlocking cache lines on the KSR1 [15].

While it is possible to implement arbitrary synchronization mechanisms on top of special-purpose locks, greater concurrency, efficiency, and fault-tolerance may be achieved by using more general-purpose primitives. Examples include `fetch_and_Φ`, `compare_and_swap`, and the pair `load_linked/store_conditional`, which can easily 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 on bus-based multiprocessors, where they are efficiently embedded in snooping cache coherence protocols, but there are many tradeoffs to be considered in designing their implementations on a DSM machine. `Compare_and_swap` and `load_linked/store_conditional` are not provided by any of the major DSM multiprocessors.

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.

The rest of this paper is organized as follows. In section 2 we discuss the differences in functionality and expressive power among the primitives under consideration. In section 3 we present several implementation options for these primitives on DSM multiprocessors. Then we present our experimental results and discuss their implications in section 4, and conclude with recommendations in section 5.

2 Atomic Primitives

2.1 Functionality

A `fetch_and_Φ` primitive [7] takes (conceptually) two parameters: the address of the destination operand, and a value parameter. It atomically reads the original value of the destination operand, computes the new value as a function Φ of the original value and the value parameter, stores this new value, and returns the original value. Examples of `fetch_and_Φ` primitives include `test_and_set`, `fetch_and_store`, `fetch_and_add`, and `fetch_and_or`.

The `compare_and_swap` primitive was first provided on the IBM System/370 [4]. `Compare_and_swap` takes three parameters: the address of the destination operand, an expected value, and a new value. If the original value of the destination operand is equal to the expected

value, the former is replaced by the latter (atomically) and the return value indicates success, otherwise the return value indicates failure.

The pair `load_linked/store_conditional`, proposed by Jensen *et al.* [13], are implemented on the MIPS II [14] and the DEC Alpha [2] architectures. They must be used together to read, modify, and write a shared location. `Load_linked` returns the value stored at the shared location and sets a *reservation* associated with the location and the processor. `Store_conditional` checks the reservation. If it is valid a new value is written to the location and the operation returns success, otherwise it returns failure. Conceptually, for each shared memory location there is a reservation bit associated with each processor. Reservations for a shared memory location are invalidated when that location is written by any processor. `Load_linked` and `store_conditional` have not been implemented on network-based multiprocessors. On bus-based multiprocessors they can easily be embedded in snooping cache coherence protocol, in such a way that should `store_conditional` fail, it fails locally without causing any bus traffic.

In practice, processors are generally limited to one outstanding reservation, and reservations may be invalidated even if the variable is not written. On the MIPS R4000 [22], for example, reservations are invalidated on context switches and TLB exceptions. We can ignore these spurious invalidations with respect to lock-freedom, so long as we always try again when a `store_conditional` fails, and so long as we never put anything between `load_linked` and `store_conditional` that may invalidate reservations deterministically. Depending on the processor, these things may include loads, stores, and incorrectly-predicted branches.

2.2 Expressive Power

Herlihy introduced an impossibility and universality hierarchy [9] that ranks atomic operations according to their relative power. The hierarchy is based on the concepts of lock-freedom and wait-freedom. A concurrent object implementation is *lock-free* if it always guarantees that some processor will complete an operation in a finite number of steps, and it is *wait-free* if it guarantees that each process will complete an operation in a finite number of steps. Lock-based operations are neither lock-free nor wait-free. In Herlihy's hierarchy, it is impossible for atomic operations at lower levels of the hierarchy to provide a lock-free implementation of atomic operations in a higher level. Atomic reads, loads, and stores are at level 1. The primitives `fetch_and_store`, `fetch_and_add`, and `test_and_set` are at level 2. `Compare_and_swap` is a universal primitive—it is at level ∞ of the hierarchy [11]. `Load_linked/store_conditional` can also be shown to be universal if we assume that reservations are invalidated *if and only if* the corresponding shared location is written.

Thus, according to Herlihy's hierarchy, `compare_and_swap` and `load_linked/store_conditional` can provide lock-free simulations of `fetch_and_Φ` primitives, and it is impossible for `fetch_and_Φ` primitives to provide lock-free simulations of `compare_and_swap` and `load_linked/store_conditional`. It should also be noted that although `fetch_and_store` and `fetch_and_add` are at the same level (level 2) in Herlihy's hierarchy, this does not imply that there are lock-free simulations of one of these primitives using the other. Similarly, while both `compare_and_swap` and the pair `load_linked/store_conditional` are univer-

sal primitives, it is possible to provide a lock-free simulation of `compare_and_swap` using `load_linked` and `store_conditional`, but not vice versa.

A pair of `atomic_load` and `compare_and_swap` cannot simulate `load_linked` and `store_conditional` because `compare_and_swap` cannot detect if a shared location has been written with the same value that has been read by the `atomic_load` or not. Thus `compare_and_swap` might succeed where `store_conditional` should fail. This feature of `compare_and_swap` can cause a problem if the data is a pointer and if a pointer can retain its original value after deallocating and reallocating the storage accessed by it. Herlihy presented methodologies for implementing lock-free (and wait-free) implementations of concurrent data objects using `compare_and_swap` [10] and `load_linked/store_conditional` [12]. The `compare_and_swap` algorithms are less efficient and conceptually more complex than the `load_linked/store_conditional` algorithms due to the pointer problem [12].

On the other hand, there are several algorithms that need or benefit from `compare_and_swap` [18, 19, 20, 27]. A simulation of `compare_and_swap` using `load_linked` and `store_conditional` is less efficient than providing `compare_and_swap` in hardware. A successful simulated `compare_and_swap` is likely to cause two cache misses instead of the one that would occur if `compare_and_swap` were supported in hardware. (If `load_linked` suffers a cache miss, it will generally obtain a shared (read-only) copy of the line. `Store_conditional` will miss again in order to obtain write permission.) Also, unlike `load_linked/store_conditional`, `compare_and_swap` is not subject to any restrictions on the loads and stores between `atomic_load` and `compare_and_swap`. Thus, it is more suitable for implementing atomic update operations that require memory access between loading and comparing (e.g. an atomic update operation that requires a table lookup based on the original value).

3 Implementations

The main design issues for implementing atomic primitives on cache coherent DSM multi-processors 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 focus our attention on `fetch_and_Φ`, `compare_and_swap`, and `load_linked/store_conditional` because of their generality, their popularity on small-scale machines, and their prevalence in the literature. We consider three implementations for `fetch_and_Φ`, five for `compare_and_swap`, and three for `load_linked/store_conditional`. The implementations can be grouped into three categories according to the coherence policies used:

1. EXC (EXclusive): Computational power in the cache controllers with 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 in 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 in 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 accesses by different processors alternate.

Other implementation options, such as computational power in the memory with a write-invalidate coherence policy, or computational power in the caches with a write-update or no-caching policy, always yield performance inferior to that of EXC.

EXC and UPD implementations are embedded in the cache coherence protocols. Our protocols are mainly based on the directory-based protocol of the DASH multiprocessor [16].

For `fetch_and_Φ`, 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 multicasts all updates to all the caches with copies.

The EXC, NOC, and UPD implementations of `compare_and_swap` are analogous to those of `fetch_and_Φ`. In addition, however, we introduce two variants of EXC: EXCd (d for deny) and EXCs (s for share). If the line is not cached exclusive, comparison of the old value with the expected value takes place in the home node or the owner node, whichever has the most up-to-date copy of the line (the home node is the node at which the memory resides). If equality holds, EXCd and EXCs behave exactly like EXC. Otherwise, the response to the requesting node indicates that `compare_and_swap` must fail, and in the case of EXCd, no cached copy is provided, while in the case of EXCs, a read-only copy is provided (instead of an exclusive copy in the case of EXC). The rationale behind these variants is to prevent a request that will fail from invalidating copies cached in other nodes.

The implementations of `load_linked/store_conditional` are somewhat more elaborate, due to the need for reservations. In the EXC implementation, each processing node has a reservation bit and a reservation address register. `Load_linked` sets the reservation bit to valid and writes the address of the shared location to the reservation register. If the cache line is not valid, a shared copy is acquired, and the value is returned. If the cache line is invalidated and the address corresponds to the one stored in the reservation register, the reservation bit is set to invalid. `Store_conditional` checks the reservation bit. If it is invalid, `store_conditional` fails. If the reservation bit is valid and the line is exclusive, `store_conditional` succeeds locally. Otherwise, the request is sent 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 and invalidations are sent to 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; see section 3.2 below.

The UPD implementation `load_linked/store_conditional` also has (conceptually) a reservation vector. `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.

3.1 Auxiliary Instructions

In order to enhance the performance of some of these implementations, we consider the following auxiliary instructions:

1. `Load_exclusive`: reads data and acquires exclusive access. If the implementation is EXC, this instruction can be used instead of an ordinary `atomic_load` when reading data that is then accessed by `compare_and_swap`. The intent is to make it likely that `compare_and_swap` will not have to go to memory. Aside from atomic primitives, `load_exclusive` is also useful in decreasing coherency operations for migratory data.
2. `Drop_copy`: if the implementation is EXC or UPD, this instruction can be used to drop (self-invalidate) cached data, if they are not expected to be accessed before an intervening access by another processor. The intent is to reduce the number of serialized messages required for subsequent accesses by other processors: a write miss will require 2 serialized messages (from requesting node to the home node and back), instead of 4 for remote exclusive data (requesting node to home to owner to home and back to requesting node) and 3 for remote shared data (from requesting node to home to sharing nodes and acknowledgments are sent back to the requesting node).

3.2 Hardware Requirements

If the base coherence policy is different from the coherence policy for access to synchronization variables, the complexity of the cache coherence protocol increases significantly. However, the directory entry size remains the same with any coherence policy on directory-based multiprocessors (modulo any requirements for reservation information in the memory).

Computational power (e.g. adders and comparators) needs to be added to each cache controller if the implementation is EXC, or to each memory module if the implementation is UPD or NOC, or to both caches and memory modules if the implementation for `compare_and_swap` is EXCd or EXCs.

If `load_linked` and `store_conditional` are implemented in the caches, one reservation bit and one reservation address register are needed to maintain ideal semantics, assuming

that `load_linked` and `store_conditional` pairs are not allowed to nest. On the MIPS R4000 processor [22] there is an `LLbit` and an on-chip system control processor register `LLAddr`. The `LLAddr` register is used only for diagnostic purposes, and serves no function during normal operation. Thus, invalidation of any cache line causes `LLbit` to be reset. A `store_conditional` to a valid cache line is not guaranteed to succeed, as the data might have been written by another process on the same physical processor. Thus, a reservation bit is needed (at least to be invalidated on a context switch).

If `load_linked` and `store_conditional` are implemented in the memory, the hardware requirements are more significant. A reservation bit for each processor is needed for each memory location. There are several options:

- A bit vector of size equal to the number of processors is added to each directory entry. This option limits the scalability of the multiprocessor, as the (total) directory size increases quadratically with the number of processors. The bits cannot be encoded, because any subset of them may legitimately be set.
- A linked list can be used to hold the ids of the processors holding reservations on a memory block. The size overhead is reduced to the size of the head of the list, if the memory block has no reservations associated with it. However, a free list is needed and it has to be maintained by the cache coherence protocol.
- A limited number of reservations (e.g. 4) can be maintained. Reservations beyond the limit will be ignored, so their corresponding `store_conditional`'s are doomed to fail. If a failure indicator can be returned by beyond-the-limit `load_linked`'s, the corresponding `store_conditional`'s can fail locally without causing any network traffic. This option eliminates the need for bit vectors or a free list. Also, it can help reduce the effect of high contention on performance. However, it compromises the semantics of lock-free objects based on `load_linked` and `store_conditional`.
- A hardware counter associated with each memory block can be used to indicate a serial number of writes to that block. `Load_linked` will return both the data and the serial number, and `store_conditional` must provide both the data and the expected serial number. A `store_conditional` with a serial number different from that of the counter will fail. The counter should be large enough (e.g. 32 bits) to eliminate any problems due to wrap around. The message sizes associated with `load_linked` and `store_conditional` increase by the counter size.

In each of these options, if the space overhead is too high to accept for all of memory, atomic operations can, with some loss of convenience, be limited to a subset of the physical address space.

For the purposes of this paper we do not need to fix an implementation for reservations in memory, but we recommend the last option. It has the potential to provide the advantages of both `compare_and_swap` and `load_linked/store_conditional`. `Load_linked` resembles a load that returns a longer datum; `store_conditional` resembles a `compare_and_swap` that provides a longer datum. The serial number portion of the datum eliminates the pointer problem mentioned in section 2.2. In addition, the lack of an explicit reservation means

that `store_conditional` does not have to be preceded closely in time by `load_linked`; a process that expects a particular value (and serial number) in memory can issue a bare `store_conditional`, just as it can issue a bare `compare_and_swap`. This capability is useful for algorithms such as the MCS queue-based spin lock [19], in which it reduces by one the number of memory accesses required to relinquish the lock. It is not even necessary that the serial number reside in special memory: `load_linked` and `store_conditional` could be designed to work on doubles. The catch is that “ordinary” stores to synchronization variables need to update the serial number. If this number were simply kept in half of a double, special instructions would need to be used instead of ordinary stores.

4 Experimental Results

4.1 Methodology

In this section we present experimental results that compare the performance of the different implementations of the atomic primitives under study. The results were collected from an execution driven cycle-by-cycle simulator. The simulator uses MINT (Mips Interpreter) [26], which simulates MIPS R4000 object code, as a front end. The back end simulates a 64 node multiprocessor with directory-based caches, 32-byte blocks, queued memory, and a 2-D worm-hole mesh network. The simulator supports directory-based cache coherence protocols with write-invalidate and write-update coherence policies. The base cache coherence protocol is a write-invalidate protocol. In order to provide accurate simulations of programs with race conditions, the simulator keeps track of the values of cached copies of atomically accessed data in the cache of each processing node. In addition to the MIPS R4000 instruction set (which includes `load_linked` and `store_conditional`), the simulated multiprocessor supports `fetch_and_Φ`, `compare_and_swap`, `load_exclusive`, and `drop_copy`. Memory and network latencies reflect the effect of memory contention and of contention at the entry and exit of the network (though not at internal nodes).

We used two sets of applications, real and synthetic, to achieve different goals. We began by studying two lock-based applications from the SPLASH suite [25]—LocusRoute and Cholesky—in order to identify typical sharing patterns of atomically accessed data. We replaced the library locks with an assembly language implementation of the test-and-test-and-set lock [23] with bounded exponential backoff implemented using the atomic primitives and auxiliary instructions under study.

Our three synthetic applications served to explore the parameter space and to provide controlled performance measurements. The first uses lock-free concurrent counters to cover the case in which `load_linked/store_conditional` simulates `fetch_and_Φ`. The second uses a counter protected by a test-and-test-and-set lock with bounded exponential backoff to cover the case in which all three primitives are used in a similar manner. The third uses a counter protected by an MCS lock [19] to cover the case in which `load_linked/store_conditional` simulates `compare_and_swap`.

	NOC	EXC	UPD
LocusRoute	1.83	1.79	1.70
Cholesky	1.62	1.68	1.59

Table 1: Average write-run length in LocusRoute and Cholesky.

4.2 Sharing Patterns

Performance of atomic primitives is affected by two main sharing pattern parameters: contention and average write-run length [5]. In this context, we define the level of contention as 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.

Table 1 shows the average write-run length of atomically accessed data in simulated runs of LocusRoute and Cholesky on 64 processors with different coherence policies. The results indicate that in these applications lock variables are unlikely to be written more than two consecutive times by the same processor without intervening accesses by other processors. In other words, a processor usually acquires and releases a lock without intervening accesses by other processors, but it is unlikely to re-acquire it without intervention.

As a measure of contention, we use histograms of the number of processors contending to access an atomically accessed shared location at the beginning of each access (we found a line graph to be more readable than a bar graph, though the results are discrete, not continuous). Figure 1 shows the contention histograms for LocusRoute and Cholesky, with different coherence policies. The figures confirm the expectation that the no-contention case is the common one, for which performance should be optimized. At the same time, they indicate 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.

4.3 Relative Performance of Implementations

We collected performance results of the synthetic applications with various levels of contention and write-run length. We used constant-time barriers supported by MINT to control the level of contention. Because these barriers are constant-time, they have no effect on the results other than enforcing the intended sharing patterns. In these applications, each processor is in a tight loop, where in each iteration it either updates the counter or not, depending on the desired level of contention. Depending on the desired average write-run length, every one or more iterations are protected by a constant-time barrier.

Figures 2, 3, and 4 show the performance results for the synthetic applications. The bars represent the elapsed time averaged over a large number of counter updates. In each figure, the graphs to the left represent the no-contention case with different numbers of

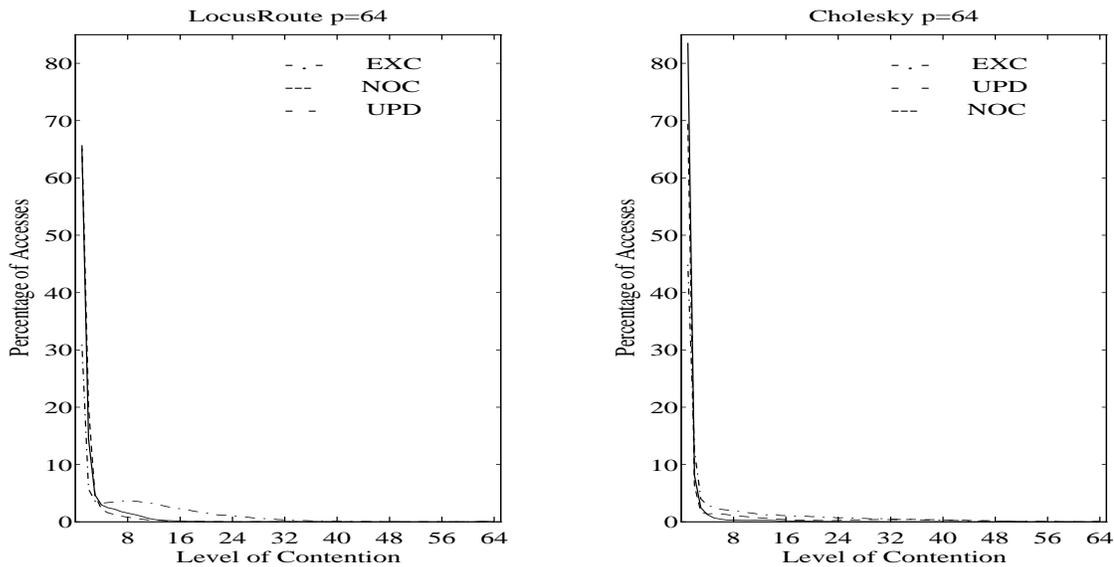


Figure 1: Histograms of the level of contention in LocusRoute and Cholesky.

consecutive 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 with using the `drop_copy` instruction, while those to the left are without using it. In each of the two subsets in the EXC category, there are 4 bars for `compare_and_swap`. They represent, from left to right, the results for the implementations EXC, EXCd, EXCs, and EXC with `load_exclusive`, respectively.

Figure 5 shows the performance results for LocusRoute. Time is measured from the beginning to the end of execution of the parallel part of the application. The order of bars in the graph is the same as in the previous figures.

We base our analysis on the results of the synthetic applications, where we have control over the parameter space. The results for LocusRoute help to validate the results of the synthetic applications. Careful inspection of trace data from the simulator suggests that the relatively poor performance of `fetch_and_Φ` in LocusRoute is due to changes in control flow that occur when very small changes in timings allow processors to obtain work from the central work queue in different orders.

4.3.1 Coherence Policy

In the case of no contention with short write-runs, NOC implementations of the three primitives are competitive, and sometimes better than, their corresponding cached implementations, even with an average write-run length as large as 2. There are two reasons for these results. First, a write miss on an uncached line takes two serialized messages, which

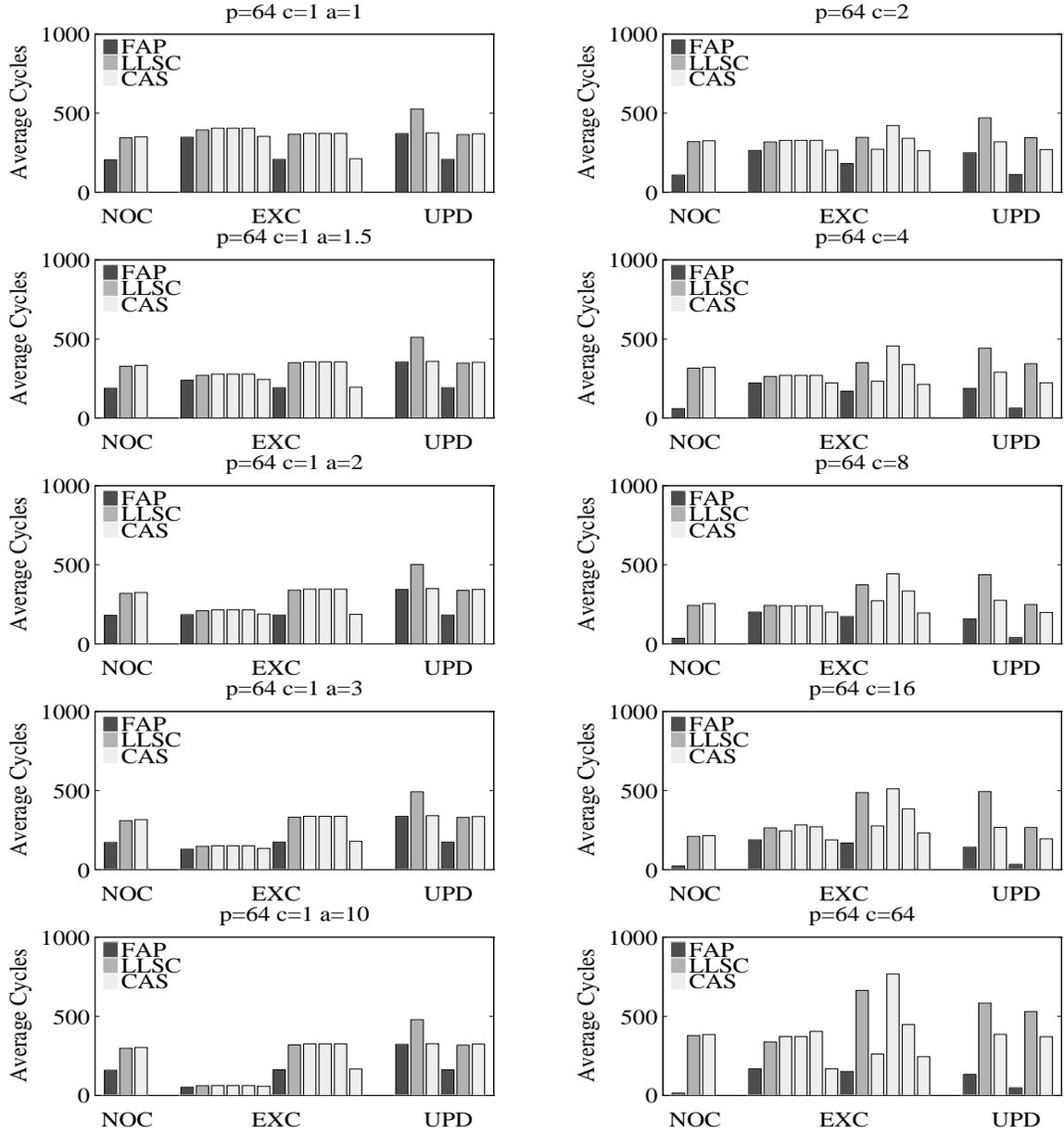


Figure 2: Average time per counter update for the lock-free counter application. P denotes processors, c contention, and a the average number of non-intervened counter updates by each processor.

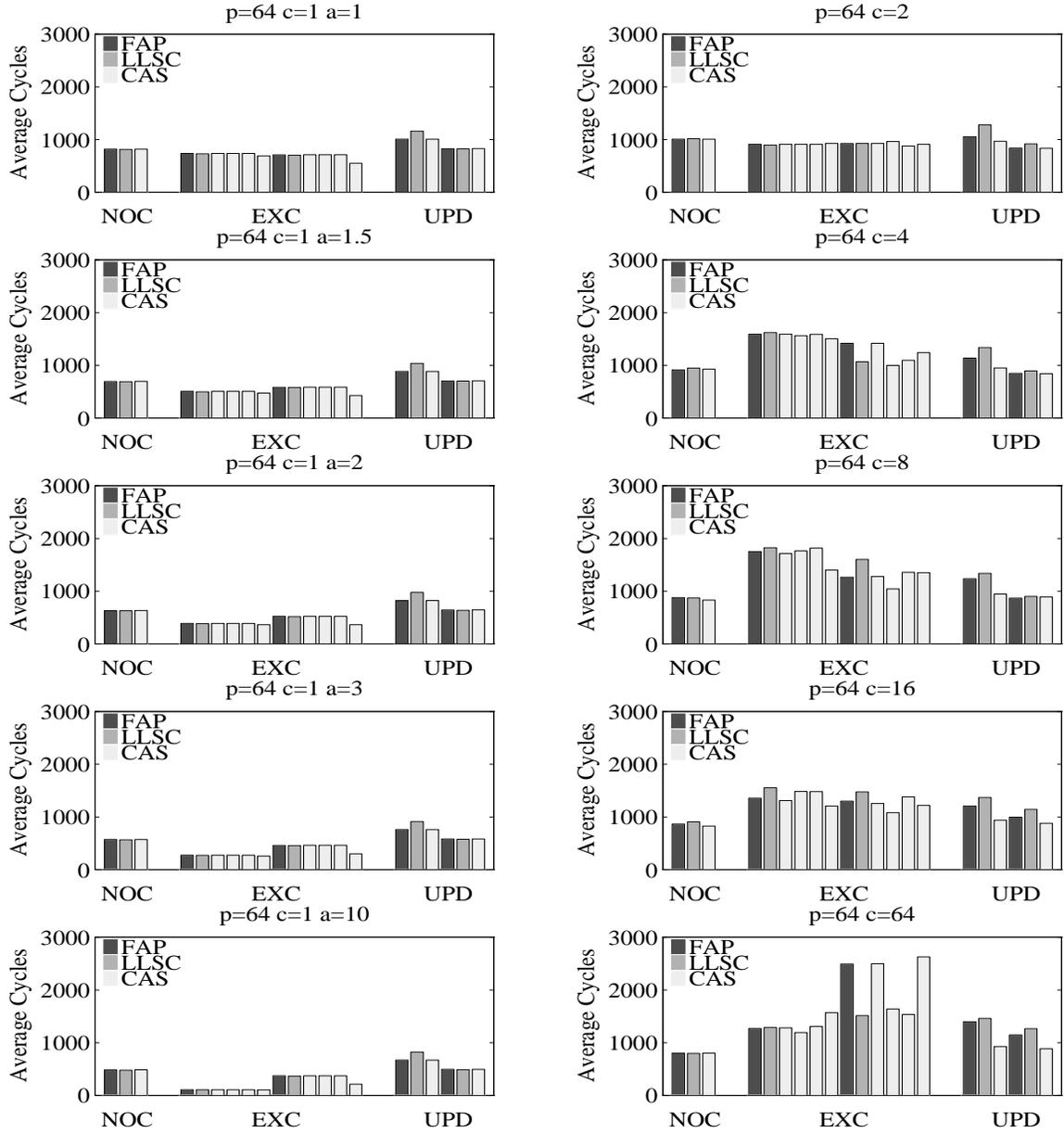


Figure 3: Average time per counter update for the TTS-lock-based counter application. P denotes processors, c contention, and a the average number of non-intervened counter updates by each processor.

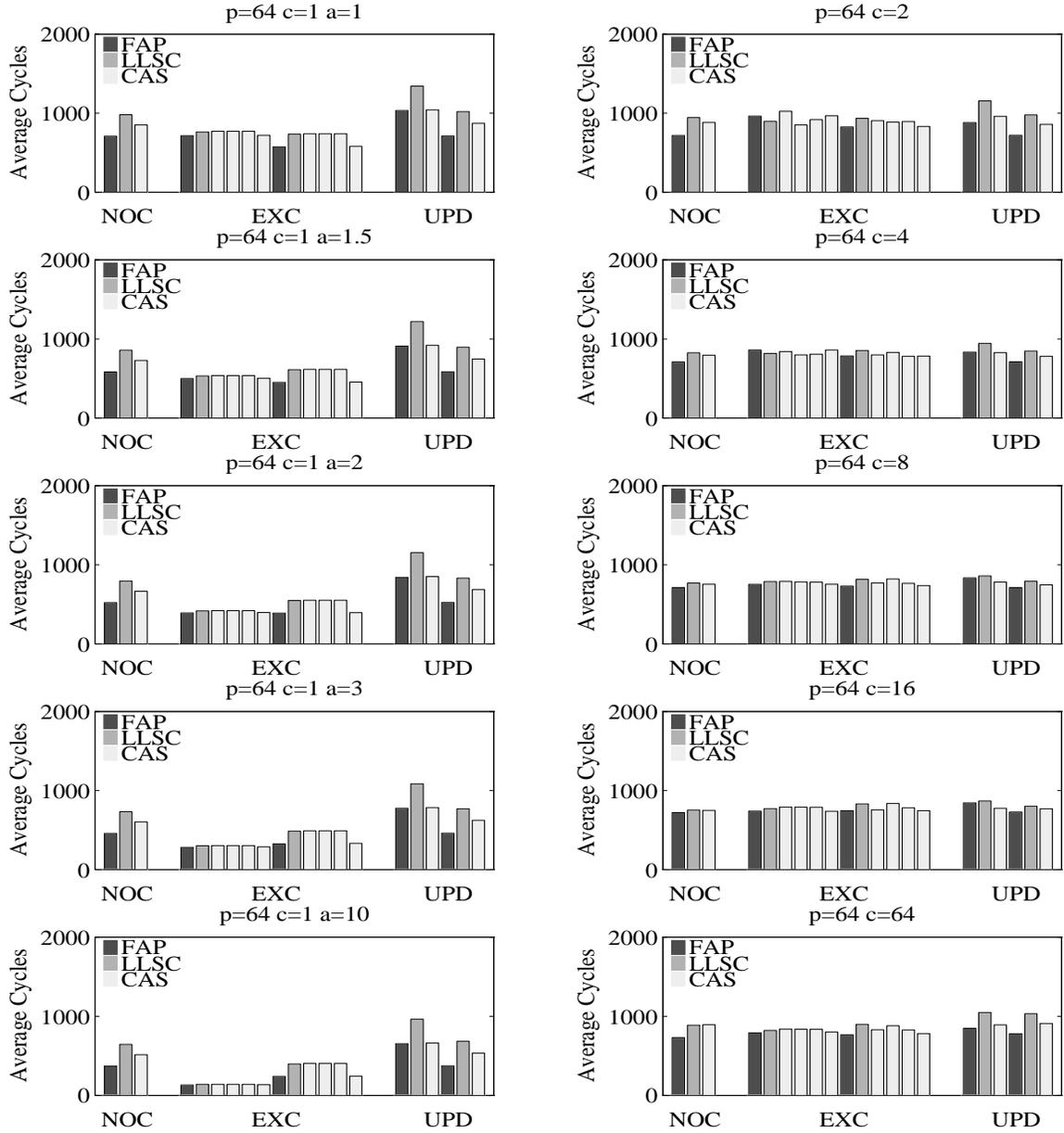


Figure 4: Average time per counter update for the MCS-lock-based counter application. P denotes processors, c contention, and a the average number of non-intervened counter updates by each processor.

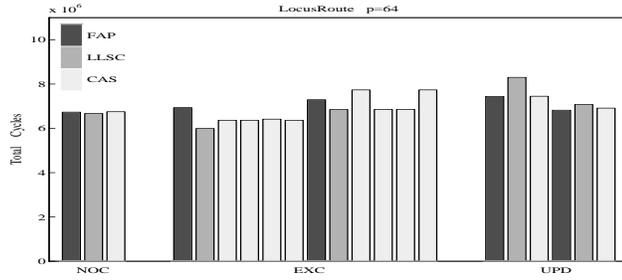


Figure 5: Total elapsed time for LocusRoute with different implementations of atomic primitives.

is always the case with NOC, while a write miss on a remote exclusive or remote shared line takes 4 or 3 serialized messages respectively. Second, NOC implementations do not incur the overhead of invalidations and updates as EXC and UPD implementations do.

Furthermore, with contention (even very low), NOC outperforms the other policies (with the exception of EXC `compare_and_swap/load_exclusive` 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 `compare_and_swap/load_exclusive` combination 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 diminishing 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 by definition, all NOC accesses are misses.

On the other hand, 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. EXC is much better in the case of long write-runs, as it benefits from caching. With higher levels of contention with the test-and-test-and-set lock, UPD is better as every time the lock is released almost all processors try to acquire it by writing to it. With EXC all these processors acquire exclusive copies although only one will eventually succeed in acquiring the lock, while in the case of UPD, only successful writes cause updates. Read-only accesses are always misses under NOC, and most of the time under EXC, but are mostly hits under UPD.

4.3.2 Atomic Primitives

In the case of the lock-free counter, 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, and may well represent bad programs (e.g. a shared counter should be updated only when necessary, instead of being repeatedly incremented). We conclude that NOC `fetch_and_add` is a useful primitive to

provide for supporting shared counters. Because it is limited to only certain kinds of algorithms, however, we recommend it only in addition to a universal primitive.

Among the EXC universal primitives, `compare_and_swap` almost always benefits from `load_exclusive`, because `compare_and_swap`'s are hits in the case of no contention and, as mentioned earlier, `load_exclusive` helps minimize the failure rate of `compare_and_swap` as contention increases. `Load_linked` cannot be exclusive: otherwise livelock is likely to occur.

The EXCd and EXCs implementations of `compare_and_swap` are almost always equal to or worse than `compare_and_swap` or `compare_and_swap/load_exclusive`. Thus, their performance does not justify the cost of extra hardware to make comparisons both in memory and in the caches.

As for UPD universal primitives, `compare_and_swap` is always better than `load_linked` and `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.

4.3.3 Drop Copy

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 `compare_and_swap/load_exclusive`, because it allows the atomic primitive to obtain the needed exclusive copy of the data with only 2 serialized messages instead of 4 (no other processor has location cached; they all have dropped their copies). As contention increases, the effect of `drop_copy` varies with the application.

With an UPD policy, `drop_copy` always improves performance, because it reduces the number of useless updates and in most cases reduces the number of serialized messages for a write from 3 to 2.

5 Conclusions

Based on the experimental results and the relative power of atomic primitives, we recommend implementing `compare_and_swap` in the cache controllers of future DSM multi-processors, with a write-invalidate coherence policy. To address the pointer problem, we recommend consideration of an implementation based on serial numbers, as described for the in-memory implementation of `load_linked/store_conditional` in section 3.2. 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 [8]). We recommend implementing it in uncached memory as an extra atomic primitive.

Acknowledgment

We thank Jack Veenstra for his invaluable effort in developing and supporting MINT and for his comments on this paper, and Leonidas Kontothanassis for his important suggestions. We also thank Robert Wisniewski, Wei Li, Michal Cierniak, and Rajesh Rao for their comments on the paper.

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] *Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.
- [3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, June 11-15, 1990.
- [4] R. P. Case and A. Padege. Architecture of the IBM System 370. *Comm. of the ACM*, 21(1):73–96, January 1978.
- [5] 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.
- [6] 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.
- [7] A. Gottlieb and C. P. Kruskal. Coordinating Parallel Processors: A Parallel Unification. *Computer Architecture News*, 9(6):16–24, October 1981.
- [8] 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.
- [9] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the 7th Symposium on Principles of Distributed Computing*, pages 276–290, Toronto, Canada, August 15-17, 1988.
- [10] M. P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. In *Proceedings of the Second Symposium on Principles and Practices of Parallel Programming*, pages 197–206, Seattle, WA, March 14-16, 1990.

- [11] M. P. Herlihy. Wait-Free Synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [12] M. P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [13] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Lab, November 1987.
- [14] G. Kane. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, N.J., 1989.
- [15] *KSR1 Principles of Operation*. Kendall Square Research Corporation, 1991.
- [16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 28-30, 1990.
- [17] 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.
- [18] H. Massalin. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, 1991.
- [19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.
- [20] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third Symposium on Principles and Practices of Parallel Programming*, pages 106–113, Williamsburg, VA, April 21-24, 1991.
- [21] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 15(8):52–60, August 1991.
- [22] *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems, Inc., 1991.
- [23] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.
- [24] *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE, Inc., 1993.
- [25] 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.

- [26] 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.
- [27] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable Spin Locks for Multiprogrammed Systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, Cancun, Mexico, April 26-29, 1994.