Speeding-up Synchronizations in DSM Multiprocessors [⋆]

A. de Dios[1], B. Sahelices[1], P. Ibáñez[2], V. Viñals[2], and J.M. Llabería[3]

[1] Dpto. de Informática. Univ. de Valladolid
agustin,benja@infor.uva.es
[2] Dpto. de Informática e Ing. de Sistemas, I3A and HiPEAC. Univ. de Zaragoza
imarin, victor@unizar.es
[3] Dpto. de Arquitectura de Computadores. Univ. Polit. de Cataluña
llaberia@ac.upc.es

Abstract. Synchronization in parallel programs is a major performance bottleneck. Shared data is protected by locks and a lot of time is spent in the competition arising at the lock hand-off. In this period of time, a large amount of traffic is targeted to the line holding the lock variable. In order to be serialized, the requests to the same cache line can either be bounced (NACKed) or buffered in the coherence controller. In this paper we focus on systems whose coherence controllers buffer requests.
During lock hand-off only the requests from the winning processor contribute to the computation progress, because the winning processor is the only one that will advance the work. This key observation leads us to propose a hardware mechanism named Request Bypass, which allows requests from the winning processor to bypass the requests buffered in the home coherence controller keeping the lock line. The mechanism does not require compiler or programmer support nor ISA or coherence protocol changes.
By simulating a 32 processor system we show that Request Bypass reduces execution time and lock stall time up to 35% and 75%, respectively. The programs limited by synchronization benefit the most from Request Bypass.

1 Introduction

The scalability of shared-memory programs is often limited by highly-contended critical sections guarded by mutual exclusion locks [1,2], where a large amount of traffic is generated during the lock hand-off. This traffic increases the time that the parallel program spends in serial mode, which reduces the benefits of parallel execution. Thus, optimizing lock transfer among processors is essential to achieve high performance in applications having highly-contended critical sections [3,4,5,6,7].

[⋆] This work was partly funded by grants TIN2004-07739-C02-01/02 (Spanish Ministry of Education/Science and European RDF) and the Diputación General de Aragón.

The processor architecture provides specific instructions to perform an atomic read-modify-write operation on a memory location. A lock is acquired by using these instructions and it is released by performing a regular write. A Highly contended lock hand-off generates a burst of traffic aimed at the same memory location. This situation may be alleviated by queue-based software [4,8,9,1] or hardware locks [5,7,10,11]. Software mechanisms have a large overhead per synchronization access, even in the absence of contention. The proposed hardware mechanisms require modifications in software and/or in the coherence protocol, they need to handle queue breakdowns, and some of them require a hardware predictor to identify synchronization operations [11]. Request Bypass, the mechanism proposed in our work, does not add a significative overhead, does not use a predictor, and it does not require changes to cache or directory protocol.

In Distributed Shared-Memory (DSM) multiprocessors a coherence request is handled by the coherence controller of the node owning the corresponding line (home node). Moreover, the coherence controller is in charge of serializing all requests targeted to the same memory address. So, requests coming to a busy directory entry cannot be attended until the directory entry becomes free. A directory entry is busy whenever the coherence controller has started a coherence operation on such entry involving a third node whose reply has not been received yet. Requests to busy entries are handled in three ways in commercial DSM multiprocessors or in the literature: either bounced [12,13], forwarded to third nodes [14,15,16] or queued within the coherence controller [17]. Our base system uses request queuing because it has the potential to reduce network traffic, contention and coherence controller occupancy as it is shown by Chaudhuri and Heinrich in [17].

In this paper we are concerned with the lock hand-off in a DSM multiprocessor that queue requests to busy lines within the coherence controller. In order to speed up lock hand-off we propose to change the order in which the coherence controller selects the request to be processed once a line leaves the busy state. Instead of always selecting an already queued request we suggest processing first the request in the input port, if it exists, a technique we call Request Bypass. At the acquire phase of the lock hand-off, Request Bypass allows the request of the winning processor (that which is going to acquire the lock) to bypass the requests to the same line pending in the queue. A similar bypassing situation can arise when accessing shared variables inside critical sections and when releasing a critical section. The implementation we propose of Request Bypass does not require compiler or programmer support nor ISA or coherence protocol changes.

In Section 2 we use an example to describe a lock hand-off for a highly contended critical section in a baseline system, and in Section 3 we analyze the same example under Request Bypass. In Section 4 we present simulation results using Splash-2 benchmarks for 32 processors. We include a comparison of our proposal with Read Combining [17]. In Section 5 we discuss related work and we conclude in Section 6.

2 Lock-transfer contention

We first describe the baseline coherence controller. Next, we elaborate on an example case of a lock transfer among several processors. This example allows us to identify inefficiency sources and motivates the main idea of the paper

2.1 Coherence controller model

The baseline model is based on a CC-NUMA multiprocessor with a MESI cache coherence protocol similar to the SGI Origin 2000 system [13]. Every memory line is allocated to one directory entry within a coherence controller which stores the line state and processes its requests.

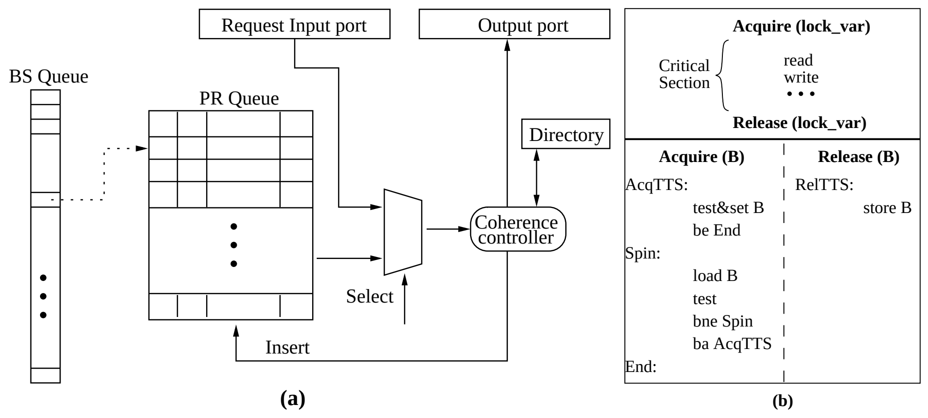
Figure 1.a shows a logical view of part of the coherence controller structure. The coherence controller receives cache requests from the nodes. Requests can be of three different types: READ_SH, READ_OWN and UPGRADE. READ_SH and READ_OWN are used to request a line in Shared or Exclusive state, respectively, and UPGRADE is used to change the line state from Shared to Exclusive. Once the coherence requests are processed, the controller sends three types of replies: REPLY_SH and REPLY_EXCL supply a line in Shared or Exclusive state, respectively, and REPLY_UPGRADE acknowledges the change from Shared to Exclusive. When needed, the coherence controller sends line invalidation (INV) or cache-cache transfer (COPYBACK) requests.

Request processing is based on two structures handled by the coherence controller: a Busy State Queue (BSQ) and a Pending Request Queue (PRQ); each PRQ entry is in turn another queue. The incoming coherence requests are taken from the input port and processed. If a request requires some third-node reply, the involved line is flagged as busy and stored in BSQ. Any request targeted to such a busy line appearing in the meantime is serialized by enqueuing its identity (originating processor, request type, etc.) into the PRQ entry associated to the corresponding BSQ entry. Otherwise, a request targeted to a non-busy lines is processed.

After receiving the last reply a busy line is waiting for, the busy state in BSQ is cleared and the coherence controller begins processing the list of pending requests to such a line in PRQ. As before, if during such processing a request requires a third-node reply, the line is tagged as busy and the coherence controller stops processing the list. When there are no pending requests in PRQ that can be processed the controller listens to the input port. Whenever the protocol runs out of BSQ or PRQ entries the coherence controller resorts to sending NACKs.

2.2 Lock hand-off example

Figure 1.b shows a typical critical section and the code used to acquire and release a lock variable. If the lock variable is already closed the code spins on a regular load instruction. Once the lock variable is released, the atomic test&set instruction tries to acquire it. Releasing is done by a regular store instruction.



```
Acquire (lock_var)
Critical  { read
Section   { write
          { • • •
Release (lock_var)
```

```
Acquire (B)           Release (B)
AcqTTS:               RelTTS:
        test&set B            store B
        be End
Spin:
        load B
        test
        bne Spin
        ba AcqTTS
End:
```

Fig. 1. (a) Logical view of part of the Coherence Controller. **(b)** Lock-based critical section skeleton (up), and code to acquire and release a lock variable (down).

Next we make a detailed study of the lock hand-off in a highly contended (n competing processors) critical section controlled by the lock variable B. Assume the lock is initially owned by a processor we call Owner, while the other $n-1$ processors are spinning on a local copy of B in Shared state. Consider that Owner is going to execute *Release(B)* and leave the critical section. After the lock hand-off is accomplished, one among the $n-1$ contending processors, we call Winner, will enter the critical section. The example in Figure 2.a shows such a scenario of contention, where the Owner, the Winner, the Home node and the remaining $n-2$ contending processors are plotted from left to right, respectively.

When Owner executes *Release(B)* it sends out an UPGRADE request to the Home node, which in turn sends INVALIDATION requests to the $n-1$ processors having the line in Shared state. The $n-1$ processors invalidate the line and send INVALIDATION replies to Home, which collects all replies and sends the UPGRADE reply to Owner. This UPGRADE reply is the first activity ploted in Figure 2.a. The example continues as follows:

– The $n-1$ processors miss loading variable B and send READ_SH to Home.
– When Home receives the first READ_SH request, it sends a cache-cache transfer request to Owner (see (1) in Figure 2.a), puts the line in busy state, and buffers the remaining $n-2$ READ_SH into PRQ.
– The Owner replies (ACK) to Home and (REPLY_SH) to Winner.
– The Winner executes *test&set* instruction and sends an UPGRADE request to Home (see (2) in Figure 2.a). The UPGRADE request of the Winner may be delayed in the input port while PRQ is emptied of READ_SH requests.
– The Home process the $n-2$ READ_SH sending REPLY_SH (now the lock is open) to every processor.
– The $n-2$ contenders receive the lock open and execute test&set. All they send UPGRADE requests to Home (not shown in figure).

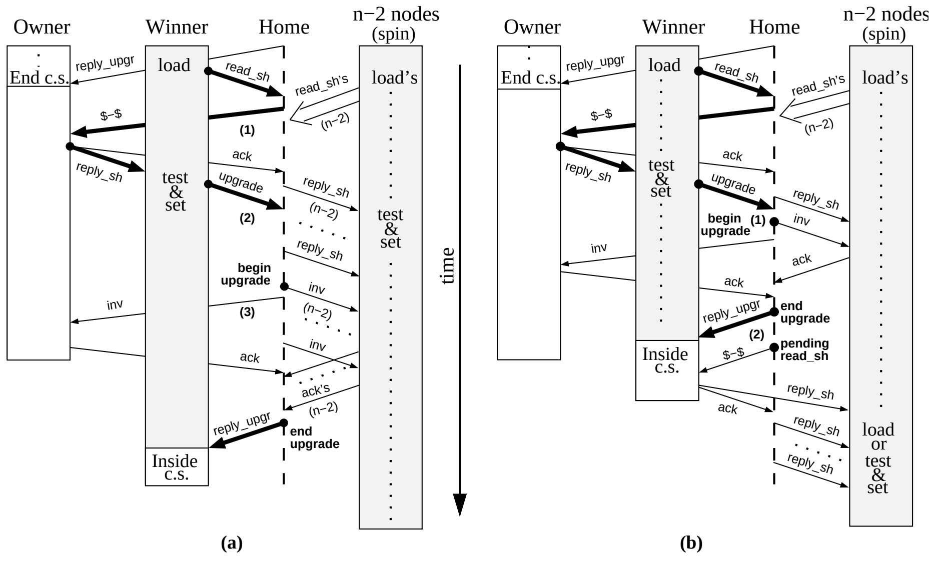


Fig. 2. Example of how a processor (Owner) frees a critical section and $n-1$ processors are contending for it. **(a)** Without bypassing. **(b)** With bypassing.

– The Home process the UPGRADE request of Winner and sends INVALIDATION requests to the Owner and the $n-2$ contenders (see (3) in Figure 2.a).
– The Home waits for all INVALIDATION ACKs and then sends an UPGRADE reply to Winner.
– The lock hand-off has been accomplished and the Winner can execute the critical section.

When processing the UPGRADE request of the remaining processors, the Home invalidates the previous copies of the line and sends it to the requesting processor, which sees the lock closed and resumes spinning. This line will be invalidated when processing the next UPGRADE request, so the processor will generate a new READ_SH that will be kept in PRQ to be serialized. While the contenders are generating the described traffic, the Winner processor is inside the critical section accessing the shared variables. Therefore, if the shared data and the lock variable are allocated to the same coherence controller, the accesses to shared variables may be delayed. This happens as long as the coherence controller is processing PRQ requests to the lock variable that do not put the lock line in Busy state.

This is because the coherence controller can be processing PRQ requests to the lock variable that does not put the lock line in Busy state. Later on, the winner releases the lock (store B) and generates a READ_OWN request. Again, it is delayed by all requests in PRQ that still contend for the line holding B.

Putting it all together, it can be expected that all of this message overhead will significantly increase the execution time of small, highly-contended, critical sections.

3 Bypassing PRQ requests

From the above example we can stablish the following: whenever several processors compete to enter into a critical section, the request traffic originated by the loosing contenders can delay all the Winner execution phases (lock acquiring, shared data accesses and lock release). In this situation, processing PRQ requests before listen to the input port does not contribute to the progress of the Winner, the only one that will advance the work. In order to favor the Winner progress, we propose that the coherence controller listens to the input port before attending the pending requests of PRQ directed to non-busy lines. So, a request in the input port to a non-busy line is going to bypass PRQ requests directed to non-busy lines.

Notice that by issuing replies in a different order as requests arrive, correctness is not affected because the serialization order among requests to the same line is only determined when the coherence controller updates the directory and sends the reply.

3.1 Request Bypass implementation in the coherence controller

As usual, an input port request targeted to a busy line is stored in PRQ, otherwise it is processed immediately. However, under a Request Bypass policy after receiving the last reply a busy line is waiting for, instead of processing PRQ requests associated to that line, the input port will be attended. Moreover, if a request appears in the input port while processing a PRQ entry, such a request will bypass all the outstanding work in PRQ. The Request Bypass policy can be easily implemented by adding a new state to each BSQ entry: the Ready state, which indicates the existence of outstanding work in PRQ.

Anyway, a Ready BSQ entry can become Busy if a request (coming either from the input port or from PRQ) require a third-node communication.

3.2 Lock hand-off example with Request Bypass

Figure 2.b shows the previous example under PRQ bypassing. We suppose that, at the time the Winner's UPGRADE reaches Home, the coherence controller is processing the first READ_SH request of the remaining contenders (the losers). When such a request is completed, the coherence controller visits the input port and processes the UPGRADE request, bypassing the n-3 READ_SH requests kept in PRQ. In our example, processing the UPGRADE request requires only two invalidations to be sent out (see (1) in Figure 2.b), one to the owner processor and another one to the single contending processor having a copy of the lock line ($n - 1$ invalidations required without bypassing).

Once the Winner's UPGRADE completes (see (2) in Figure 2.b), all the $n - 3$ remaining READ_SH requests that were bypassed will be processed. However, in contrast with the previous situation, the losers receive the READ_SH reply with the lock closed and therefore remain spinning locally, not executing the test&set instruction nor generating any request (UPGRADE or READ OWN).

While the coherence controller is servicing READ_SH requests from PRQ, the Winner is inside the critical section, sending requests (may be some of them to the same controller) to access the shared variables, and sending a final request to release the lock. However, such requests are not delayed because they bypass PRQ.

3.3 Forward progress warranty

Bypassing PRQ requests can delay execution endlessly. As an example let us suppose that the code to acquire a critical section spins on a test&set instruction. In a highly contended critical section the coherence controller is receiving READ_OWN requests continuously. If the owner of the critical section is trying to release it by sending a READ_OWN request, and this request is queued in PRQ, then the owner stays indefinitely in the critical section. Bypassing READ_SH has a similar problem.

In order to warrant forward progress, we suggest limiting the maximum number of consecutive bypasses. We can implement this idea by incrementing a counter each time an input port request bypasses PRQ and decreasing the counter each time a request is processed in PRQ. If the counter has a value between 0 and $Max - 1$ then the controller works in Request Bypass mode. Otherwise, when the counter gets its maximum value the controller switches to default mode and remains in it until the counter decreases. Our experiments show good results with a 5-bit counter.

4 Experimental results

Our simulations have been conducted with RSIM [18,19]. It is an execution-driven simulator performing a detailed cycle-by-cycle simulation of an out-of-order processor, a memory hierarchy, and an interconnection network. The processor implements a sequential consistency model using speculative load execution [20]. Coherence is based on a MESI protocol similar to the SGI Origin 2000 system [13]. The network is a wormhole-routed two-dimensional mesh network. Port contention, switches and links are accurately modeled. Table 1.a lists the processor, cache and memory system parameters.

As a workload we have chosen a SPLASH-II subset [21] having a significant amount of synchronization, see Table 1.b. In *Ocean* we use the optimization suggested in [22]. The applications have been compiled with a *test and test&set*-based synchronization library (Figure 1.b) implemented with the RMW instruction. Barriers are implemented with a simple binary tree.

Table 1. Simulated system parameters and applications.

Processor	1 Ghz
ROB	64−entry, 32−entry LS queue
Issue	out−of−order issue/commit 4−ops/cyc.
Branch	512−entry branch predictor buffer
Cache	
L1 inst.	Perfect
L1 data	128−Kbyte, direct mapped, write−back 2 ports, 1−cycle, 16 outstanding misses
L2	1−Mbyte, 4−way associative, write−back 10−cycle access, 16 outstanding misses
L1/L2 bus	Runs at processor clock
Line size	64 bytes

L2/Memory Bus	Split. 32−bits 3−cycle+1 arbit.
Memory	4−way interleaved, 50−cycle DRAM
Directory	SGI Origin−2000 based MESI
Cycle	16−cycle (without memory)
Interleaving	4 controllers per node
BSQ/PRQ size	64/16−entries
Network	Pipelined point−to−point
Network width	8−bytes/flit
Switch buffer size	64 flits
Switch latency	4−cycles/flit + 4 arbit.

(a) Simulated system parameters

Code	Ocean	Barnes	Volrend	Water−Nsq	Water−Spt	FMM
Input	130x130	4K particles	head−scaleddown2	512 molecules	512 molec.	2K partic.

(b) Applications

Our results show execution time broken down into four categories: *lock*, *barrier*, *memory* and *compute*. The algorithm used to add a cycle into a given category works as follows: if the maximum allowed number of instructions can be committed from the ROB, the cycle is added to *compute*. Otherwise, the cycle is added to the stall category to which belongs the oldest instruction that can not be committed, as suggested in [19].

4.1 Results

In this section we present results for a baseline system without bypassing, and for a system enhanced with Request Bypass. We also consider a third system, by enhancing the baseline with Read Combining [17]. Chaudhuri and Heinrich propose Read Combining in the context of queuing coherence controllers, in order to speed up multiple read requests to the same line. In order to achieve this, Read Combining dictates that once the controller gets the line, it is stored in a fast data buffer which is repeatedly used to send out all the read requests replies. They show that Read Combining also benefits lock transfer by enabling a faster distribution of the cache line storing the lock variable. Finally, we evaluate the performance of merging Request Bypass with Read Combining.

Figure 3 shows the parallel execution time break into the former categories and normalized to the baseline system. From left to right we show, for each application, the baseline system (Baseline), the baseline system enhanced with the Read Combining (RC) [17], the baseline system enhanced with Request Bypass (Byp) and two ways of merging Request Bypass with Read Combining (RC+Byp

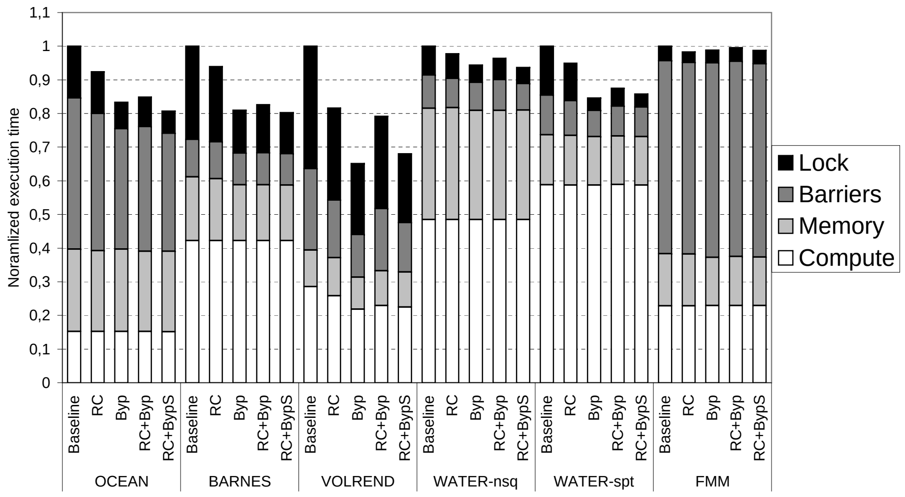


Fig. 3. Normalized execution time with 32 processors for the Baseline, Read Combining (RC), and Request Bypass (Byp) systems. RC+Byp (Blind merging) and RC+BypS (Selective merging) are the merged systems.

and RC+BypS, see details below). We only show data for 32 processors because Splash-2 applications have small Acquire-related times with 16 processors [23].

By applying Request Bypass to the Baseline system, the Lock time becomes greatly reduced, from 12% to 75%. Reductions in the Barrier time can also be observed, to a greater or lesser extent, for all applications, from 1% to 48%. This reduction in the Barrier stall time can be explained as follows: when a critical section executes before a nearby barrier, reducing Lock-related stalls also reduces Barrier-related stalls, because the Barrier stall time of a given processor starts from when it reaches the barrier until the slowest processor exits the critical section and crosses the barrier the last. Summarizing, the overall time reduction obtained with Request Bypass varies from 1% to 35%.

The Lock time reduction achieved by Request Bypass is greater than Read Combining for most applications. Read Combining itself does not alleviate the delay experienced by the Winner processor because it does not acquire the critical section until the contending processor requests kept in the buffer have been replied. Moreover, Read Combining exposes subsequent delays when accessing protected data and releasing the lock. Until the contending processors start busy-waiting on a local copy of the line, their requests will delay the progress of the Winner processor, firstly by delaying the requests made to the same coherence controller within the critical section, and secondly by delaying the request to release the lock.

Next we analyze the interaction between Request Bypass and Read Combining when applied simultaneously. Figure 3 presents data for two experiments merging Request Bypass and Read Combining namely Blind (RC+Byp) and Selective (RC+BypS). Blind merging implements both techniques simultaneously as they have been defined, resulting in a Lock time increase for all applications. This is because Read Combining speeds-up the READ_SH replies to contenders when a lock is released, and as a consequence, the UPDATE request in a Blind merging system bypasses less READ_SH requests than with Request Bypass working alone. So, more processors receive the lock variable opened, execute the test&set instruction, and have to be invalidated. Moreover, when the Winner wants to release the lock, its UPGRADE (or READ_OWN) request cannot bypass PRQ because the lock line is busy most of the time (the coherence controller is servicing the READ_OWN requests of the test&set instructions).

A Selective merging of Request Bypass and Read Combining tries to overcome the above problem by applying Read Combining only to lines which do not contain a lock variable. The execution time of Selective merging is similar to that of Request Bypass alone in our benchmarks. However, we can expect a better behavior in programs with a communication pattern where one processor produces for many consumers.

Finally, we have analyzed the sensitivity of results to the latency of some key components such as the router, the coherence controller and the memory, verifying that conclusions hold across the considered design space [23].

5 Related work

Goodman et al. propose a very aggressive hardware support for locks (QLB - originally called QOSB) [5]. In their proposal a distributed linked list of processors waiting on a lock is maintained entirely in hardware, and the release transfers the lock to the first waiting processor without affecting the other contending processors. QOLB has proven to offer substantial speed up, but at the cost of software support, ISA changes and protocol complexity [7].

The DASH project provided a concept of queue locks in hardware for directory-based multiprocessors [24]. On a release, the lock is sent to the directory which randomly selects a waiting processor to acquire the lock.

Rajwar et al. propose to predict synchronization operations in each processor by building a speculative hardware-based queuing mechanism (IQOLB) for snoop-based and directory systems [10,11]. They use the notion of buffering external requests, applying it to cache lines supposed to contain a synchronization variable. The mechanism does not require any change to existing software or ISA, but requires changes in the cache or in the directory protocol in order to make the intelligent choices needed to implement the mechanism and some additional bits in directory entries.

The above described hardware queue-based mechanisms need to handle queue breakdowns (due to line eviction or multiprogramming). The mechanism proposed in our work does not use a predictor and does not require changes to cache or directory protocol.

The *combining pending read request* technique as proposed by Chaudhuri et al. [17], was initially intended to eliminate NACKs, but significantly accelerates lock acquiring in lock-intensive applications. It is based on buffering pending requests, so our work requires the same hardware support but uses a different selection heuristic.

6 Concluding remarks

In this paper we introduce Request Bypass, a technique to speed-up the lock hand-off in DSM multiprocessors which use queuing at the coherence controller in order to serialize requests to busy lines. Under Request Bypass, the requests in the input port of the coherence controller are attended before the requests queued in the coherence controller which are directed to non-busy lines. The mechanism does not require compiler or programmer support nor ISA or coherence protocol changes.

When accessing a highly contended critical section, Request Bypass allows the Winner processor requests to bypass the queued requests of the contending processors, speeding-up the Winner execution of all critical section phases, namely lock acquiring, shared data accessing, and lock releasing.

Simulations performed in 32-processor systems show that Request Bypass reduces the overall execution time to some extent in all our tested benchmarks. The reduction is noticeable in programs with a large synchronization overhead, reaching 35% of execution time reduction and 75% of Lock time reduction. Read Combining also reduces both execution time and synchronization overhead, but to a lesser extent.

We have also merged naively Request Bypass and Read Combining. In this merged mode, when Read Combining operates on lock lines, it eliminates some of the benefits obtained with Request Bypass. This negative effect disappears when both techniques are applied selectively.

References

1. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared memory multiprocessors. ACM Trans. on Computer Systems **9**(1) (1991) 21–65
2. Michael, M., Scott, M.: Implementation of atomic primitives on distributed shared memory multiprocessors. In: Proc. 1st HPCA. (1995) 221–231
3. Anderson, T.: The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In: Proc. ICPP, volume II. (1989) 170–174
4. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. on Parallel and Distributed Systems **1**(1) (1990) 6–16
5. Goodman, J., Vernon, M., Woest, P.: Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In: Proc. 3th ASPLOS. (1989) 64–75
6. Kagi, A.: Mechanisms for Efficient Shared-Memory, Lock-Based Synchronization. PhD thesis, University of Wisconsin. Madison (1999)

7. Kagi, A., Burger, D., Goodman, J.: Efficient synchronization: let them eat QOLB. In: Proc. 24th ISCA. (1997) 170–180
8. Graunke, G., Thakkar, S.: Synchronization algorithms for shared memory multiprocessors. IEEE Computer **23**(6) (1990) 60–69
9. Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proc. 8th ISPP. (1994) 165–171
10. Rajwar, R., Kagi, A., Goodman, J.: Improving the throughput of synchronization by insertion of delays. In: Proc. 6th HPCA. (2000)
11. Rajwar, R., Kagi, A., Goodman, J.: Inferential queueing and speculative push for reducing critical communication latencies. In: Proc. 17th ICS. (2003) 273–284
12. Kuskin, J., et al.: The stanford FLASH multiprocessor. In: Proc. 21th ISCA. (1994) 302–313
13. Laudon, J., Lenoski, D.: The SGI Origin: A CC-NUMA highly scalable server. In: Proc. 24th ISCA. (1997)
14. Barroso, L., et al.: Piranha: A scalable architecture based on single-chip multiprocessing. In: Proc. 27th ISCA. (2000) 282–293
15. Gharachorloo, K., et al.: Architecture and design of ALPHASERVER GS320. In: Proc. 9th ASPLOS. (2000) 13–24
16. James, D., Laundrie, A., Gjessing, S., Sohni, G.: Distributed directory scheme: Scalable coherence interface. IEEE Computer **23**(6) (1990)
17. Chaudhuri, M., Heinrich, M.: The impact of negative acknowledgments in shared memory scientific applications. IEEE Trans. on Parallel and Distributed Systems **15**(2) (2004) 134–152
18. Pai, V., Ranganathan, P., Adve, S.: RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In: WCAE-3. (1997)
19. Pai, V., Ranganathan, P., Adve, S.: RSIM reference manual version 1.0. Technical report 9705, Dept. of Electrical and Computer Engineering, Rice University (1997)
20. Gharachorloo, K., Gupta, A., Hennessy, J.: Two techniques to enhance the performance of memory consistency models. In: Proc. ICPP. (1991) 355–364
21. Woo, S., et al.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. 22th ISCA. (1995) 24–36
22. Heinrich, M., Chaudhuri, M.: Ocean warning: Avoid drowing. Computer Architecture News **31**(3) (2003) 30–32
23. de Dios, A., Sahelices, B., Ibáñez, P., Viñals, V., Llabería, J.M.: Speeding-up synchronizations in DSM multiprocessors. Tech. rep. DIIS RR-06-07, University of Zaragoza. Spain. (2006)
24. Lenoski, D., et al.: The stanford DASH multiprocessor. IEEE Computer **25**(3) (1992) 63–79