The Impact of Negative Acknowledgments in Shared Memory Scientific Applications

Mainak Chaudhuri, *Student Member*, *IEEE*, and Mark Heinrich, *Member*, *IEEE*

Abstract—Negative ACKnowledgments (NACKs) and subsequent retries, used to resolve races and to enforce a total order among shared memory accesses in distributed shared memory (DSM) multiprocessors, not only introduce extra network traffic and contention, but also increase node controller occupancy, especially at the home. In this paper, we present possible protocol optimizations to minimize these retries and offer a thorough study of the performance effects of these messages on six scalable scientific applications running on 64-node systems and larger. To eliminate NACKs, we present a mechanism to queue pending requests at the main memory of the home node and augment it with a novel technique of combining pending read requests, thereby accelerating the parallel execution for 64 nodes by as much as 41 percent (a speedup of 1.41) compared to a modified version of the SGI Origin 2000 protocol. We further design and evaluate a protocol by combining this mechanism with a technique that we call write string forwarding, used in the AlphaServer GS320 and Piranha systems. We find that without careful design considerations, especially regarding atomic read-modify-write operations, this aggressive write forwarding can hurt performance. We identify and evaluate the necessary micro-architectural support to solve this problem. We compare the performance of these novel NACK-free protocols with a base bitvector protocol, a modified version of the SGI Origin 2000 protocol, and a NACK-free protocol that uses dirty sharing and write string forwarding as in the Piranha system. To understand the effects of network speed and topology the evaluation is carried out on three network configurations.

Index Terms—Distributed shared memory, cache coherence protocol, negative acknowledgment, node controller occupancy.

1 Introduction

DSM multiprocessors employing home-based cache coherence protocols assign a *home node* to each cache line. Every memory request in such a system is first sent to the home node of the requested cache line where the corresponding *directory entry* is consulted to find out the *sharing status* of that line. Eventually, an appropriate reply message arrives at the requester. Negative acknowledgments serve as replies when a read or a write request finds the directory entry in a pending or busy state (i.e., any transient unstable state that may arise because of the distributed nature of a coherence protocol) or fails to find the data at a third owner node. The latter case arises from two kinds of intervention races: early intervention and late intervention. An early intervention race occurs when a forwarded intervention reaches the dirty third node (the owner) before that node has even received its write reply. A late intervention race occurs when a forwarded intervention reaches the owner after it has issued a writeback message to the home node. Also, a request may be negatively acknowledged if the home node fails to allocate all the resources necessary to serve it. However, this can be solved by either properly sizing all the resources or by carefully designing the coherence protocol. None of the protocols presented in this paper generate NACKs because of resource shortage.

NACKs not only introduce extra network traffic but also increase node controller occupancy, which is known to be a critical determinant of performance [5]. This paper presents novel scalable mechanisms to minimize NACKs, shows that effective elimination of NACKs can lead to significant performance improvement, and offers a thorough analysis of the performance impact of these messages across a family of new and existing bitvector protocols on 32, 64, and 128-node DSM multiprocessors. Starting from a basic bitvector protocol [14], [15], we first improve it to get the benefits of the SGI Origin 2000 protocol [20] as far as the NACKs are concerned (Section 2). Although this protocol eliminates all intervention races, the home node still generates NACKs if the directory entry is in a pending state.

To eliminate the remaining NACKs, we present a mechanism to store pending requests in the main memory of the home node and introduce the novel concept of *pending read combining* (Section 3). Further, we implement *pending write combining* by augmenting this mechanism with a technique that we call *write string forwarding* following the write forwarding idea of the AlphaServer GS320 [8] and Piranha [3] protocols. However, our evaluation (Sections 4 and 5) shows that write forwarding may hurt the performance of atomic read-modify-write operations and heavily-contended critical sections in large-scale systems. We propose small microarchitectural changes in the cache subsystem to improve the performance of read-modify-write operations in these protocols. The proposed architectural changes are similar to the delayed response scheme introduced in [26], but we do not resort to the time-out technique proposed there. Finally, a quantitative comparison of our NACK-free protocols against a NACK-free protocol that uses dirty sharing and write string forwarding as in the Piranha system shows that an increased number of intervention misses severely hurts

- M. Chaudhuri is with the Computer Systems Laboratory, Cornell University, Ithaca, NY 14853. E-mail: mainak@csl.cornell.edu.
- M. Heinrich is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816. E-mail: heinrich@cs.ucf.edu.

Manuscript received 21 Feb. 2003; revised 21 July 2003; accepted 23 July 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0015-0203.

1045-9219/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

the performance of the latter in the presence of large-scale producer-consumer sharing.

1.1 Related Work

Organizing the sharers as a bitvector in DSM cache coherence protocols is popular in both academia [4], [21], [22] and industry [3], [8], [20] because its simplicity yields efficient and high-performance implementations. While this article focuses on a family of bitvector protocols, it is relevant to any home-based protocol [13], [14], [15], [29]. Cache-based protocols, such as IEEE Scalable Coherent Interface [14], [15], [23], [24], [28], may have fewer NACKs, but their distributed linked list directory structure leads to substantial design complexity and results in large invalidation latency, and typically poor performance [15].

The SGI Origin 2000 [20] protocol eliminates the negative acknowledgments related to forwarded interventions (i.e., the three-hop races), but the presence of busy states in the directory still forces the home node to generate NACKs. Our implementation of a modified version of this protocol that eliminates the three-hop races is described in Section 2.

The designs of the AlphaServer GS320 [8] and the Piranha chip-multiprocessor [3] introduce coherence protocols that do not generate NACKs. But the GS320 protocol and the intrachip protocol of Piranha have to rely on point-to-point network ordering and total ordering properties among certain types of messages. The internode protocol of Piranha eliminates this constraint and still remains NACK-free. The concepts of dirty sharing and continuous write forwarding in these protocols, as discussed in Section 3, help remove the busy states from the directory entry. These two optimizations may help accelerate migratory data accesses protected by largely uncontended locks, as observed in commercial workloads [2], [27]. Our evaluation shows that without extra design considerations these optimizations may hurt the performance of heavily contended read-modify-write operations (e.g., in contended lock acquires using LL/SC instructions), relatively long critical sections, and large-scale producer-consumer sharing. Instead, our NACK-free protocols store the pending requests in the protocol section of the main memory at the home node and combine the pending reads, thereby considerably accelerating the execution of scientific codes.

The Sun WildFire [12] connects four large snoopy SMP nodes with a directory-based protocol that uses extra messages to resolve three-hop races. Writebacks use a three-phase protocol to first get permission from the home node before sending the data. Also, forwarded three-hop intervention replies must send completion acknowledgment messages to the home node. These design choices, along with the combination of snooping within a directory-based scheme, lead to higher occupancy and message counts than the directory-based protocols used in this paper.

The Cray SV2 system [1] uses a blocking protocol that does not have NACKs. However, the mechanism used in this system is different from our pending request combining technique. In the Cray SV2 protocol, the messages that cannot be processed put back-pressure on the virtual channels and the pending messages are serviced in order to preserve point-to-point ordering in the virtual network. None of our protocols presented in this paper are constrained by network ordering requirements.

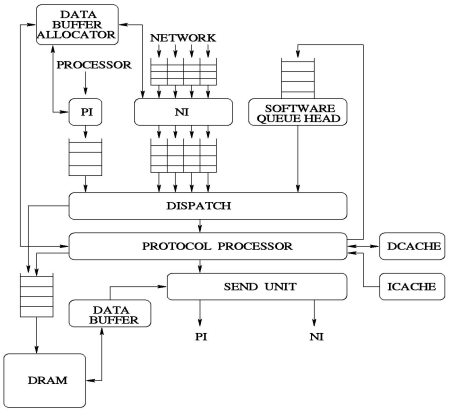


Fig. 1. Node controller architecture.

Our study shows that for applications with heavily contended read-modify-write operations (e.g., in centralized flat barriers, lock acquires etc.) the majority of the NACKs arise from load-linked (LL) and store-conditional (SC) instructions. But there are applications for which the remaining NACKs play the most important role. In this paper, we use simple LL/SC-based locks since this is the most common ABI provided by most systems. Queue-on-SyncBit (QOSB) [10], Queue-on-LockBit (QOLB) [18], and Implicit Queue-on-LockBit (IQOLB) [26] present mechanisms to form a queue of lock contenders. In contrast, our technique of queuing pending requests at the home node does not require any processor ISA, cache subsystem, or software modifications. If timings are favorable, our technique can lead to the formation of the same orderly queue of lock contenders as proposed in these studies. We find that our technique of buffering at the home node in conjunction with read combining can substantially reduce the lock acquire time in relevant applications. Further, our technique lends itself naturally to accelerating any other heavily-contended producer-consumer accesses that are carried out through normal load/store operations or atomic fetch-and-ϕ operations (as in centralized flat barriers). This is not possible even in a system providing hardware or ABI support for simple queue-based locks. The high performance of our technique results solely from the efficient elimination of negative acknowledgments.

2 Baseline Coherence Protocols

We start our protocol evaluation with a basic bitvector protocol and gradually improve it to eliminate negative acknowledgments. In this paper, we present the protocols in sufficient detail, however, interested readers can find the complete protocol state machines in the appendix of [6].

Our node controller architecture shown in Fig. 1 is directly derived from the Memory And General Interconnect Controller (MAGIC) of the Stanford FLASH multiprocessor. It has similar functionality to the hub of the SGI Origin 2000, except that our node controller is programmable and can execute any cache coherence protocol. Coherence messages arrive at the processor interface (PI) or the network interface (NI) and wait for the dispatch unit to schedule them. The processor interface has an outstanding transaction table (OTT) to record the outstanding read, upgrade, and read-exclusive requests issued by the local processor. The network interface is equipped with four virtual lanes to implement a deadlock-free protocol and obviates the need for a strict request-reply protocol. The dispatch unit carries out a round robin selection among the PI queue, four NI queues, and the software queue (see below). After a message is selected, a table lookup decides which protocol handler is invoked to service the message. It may happen that while running a handler (e.g., one that sends out invalidations) the protocol processor finds that it needs more space in an outgoing network queue. At this point, the incomplete message is stored on the software queue, which is a reserved space in main memory. At some point, the dispatch unit will re-schedule this message from the head of the software queue, and the handler can continue where it left off. The protocol processor has its own instruction and data caches and communicates with main memory via cache line-sized data buffers. During handler execution, the protocol processor may instruct the send unit to send out certain types of messages (such as requests/replies/interventions) to either the local processor (via the PI) or remote nodes (via the NI).

In the following, we discuss our baseline protocols that resort to NACKs to resolve races.

2.1 Baseline Bitvector Protocol

This protocol has a 64-bit directory entry, 48 bits of which are dedicated to a sharer vector; two bits are used to mark dirty and pending states, and the remaining bits are used to keep track of the number of invalidation acknowledgments to receive on a write. The sharer vector is used as a bit vector when the memory line is in the shared state; otherwise it is used to store the identity of the dirty exclusive owner. The protocol runs under a relaxed consistency model that sends eager-exclusive replies where upgrade acknowledgments and read-exclusive data replies are sent to the requester even before all the invalidation messages are sent and all the acknowledgments are collected. After the home node receives the last invalidation acknowledgment, it clears the pending state of the directory entry and sends a write-completion message to the writer. Our relaxed consistency model guarantees "global completion" of all writes on release boundaries thereby preserving the semantics of flags, locks, and barriers. The pending state in the directory entry is also set when a request is forwarded to a third dirty node. The third node is expected to send a sharing writeback message (for forwarded read) or an ownership transfer message (for forwarded write) or a pending clear message (if the forwarded request fails to find the cache line in the third node) to the home node that clears the pending state of the directory entry. In this protocol, NACKs are generated by the home node when a request arrives for a memory line with the corresponding directory entry in the pending state. NACKs are also generated by third party nodes in case of early and late intervention races.

2.2 Modified Origin 2000 Protocol

This simplified version of the SGI Origin 2000 protocol differs from the actual Origin protocol in four major ways. First, our protocol is MSI as opposed to MESI and, therefore, the home node does not send speculative replies to the requester on three-hop misses. Second, our protocol supports eager-exclusive replies as opposed to strict sequential consistency. Third, our protocol sends an exclusive data reply (versus a NACK), if an upgrade request comes from a node that is not marked as a sharer in the directory. Finally, our protocol uses three virtual lanes as opposed to a two-lane strict request-reply mechanism. The third lane is used to send invalidation and intervention messages. This eliminates the necessity of the back-off mechanism used in the SGI Origin 2000.

The directory entry is 64 bits wide. Among these 64 bits, four bits are dedicated to maintain state information: pending shared, pending dirty exclusive, dirty, and local. The sharer vector is 32 bits wide. The remaining 28 bits are left unused for future extensions of the protocol. The pending shared and pending dirty exclusive states are used to mark the directory entry busy when read and read exclusive requests are forwarded by the home node to the current owner. The dirty bit is set when a memory line is cached by one processor in the exclusive state. The local bit indicates whether the local processor caches the line. As in the Origin protocol, our protocol collects the invalidation acknowledgments at the requester, though we again support eager-exclusive replies. Our modified Origin protocol also eliminates NACKs in the case of early or late intervention races. Early interventions are buffered in the outstanding transaction table (OTT) of the designated owner and delayed until the write reply arrives. Late interventions are handled by the home node when it receives the writeback and are ignored by the third party nodes. To properly decide which interventions to ignore, the node controller requires a writeback buffer recording the addresses of the outstanding writeback messages and the protocol needs to support two types of writeback acknowledgments.

In this protocol, NACKs are generated only by the home node when a read, upgrade, or read-exclusive request finds the corresponding directory entry in one of the two pending states. Since this protocol properly resolves all intervention races, the third party nodes do not generate NACKs.

3 Eliminating the Negative Acknowledgments

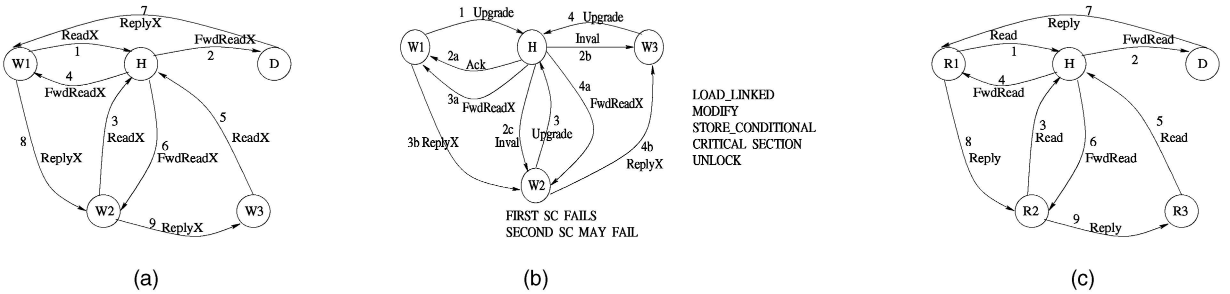
This section describes our coherence protocols that eliminate the remaining NACKs of the modified SGI Origin 2000 protocol. We first present a brief overview of continuous write forwarding and dirty sharing as implemented in the Piranha internode coherence protocol.

3.1 Write String Forwarding and Dirty Sharing

Write string forwarding and dirty sharing are the two major optimizations of the Piranha inter-node coherence protocol that eliminate the pending states from the directory. We will discuss the salient features of these two techniques along with the problems each can face with heavily contended critical sections and large-scale producer-consumer sharing on scalable DSM machines.

3.1.1 Write String Forwarding

To eliminate the pending dirty exclusive state, the protocol, on a write request (e.g., an upgrade or a read-exclusive request), changes the directory entry immediately to reflect the new owner and forwards the request to the old owner if the state of the line is not unowned. As a result, a string of



(a)

(b)

(c)

Fig. 2. (a) Write string forwarding, (b) effects of write string forwarding on critical sections, and (c) dirty sharing.

write requests continuously gets forwarded to the previous owners obviating the need for ownership transfer messages. In Fig. 2a, three nodes, namely, W1, W2, and W3, try to write to a cache line that is in the dirty exclusive state in node D. The request from W1 (message 1) gets forwarded to D (message 2) and the directory entry is changed to reflect the new owner W1. When the request from W2 (message 3) arrives at the home node it gets forwarded to W1 (message 4). Similarly, the request from W3 (message 5) gets forwarded to W2 (message 6). Finally, when W1 receives the reply (message 7) it completes its write and satisfies the waiting intervention from W2 via the reply message 8. Similarly, W2 sends a reply to W3 via message 9 after completing its own write. The home node relies on the third party nodes to *always* be able to satisfy forwarded requests. Although it is possible for the dirty node to have written back the line before the forwarded intervention arrives, this is easily solved by making the writeback buffer hold the cache line in addition to its address until the writeback is acknowledged by the home node.

Let us analyze the effects of write string forwarding on the performance of heavily contended read-modify-write operations implemented using LL/SC pairs. Although the following discussion focuses only on lock acquire, the same effects will be observed in any other atomic read-modify-write operations. A simple implementation of a critical section is shown on the right of Fig. 2b. The load-linked (normally includes LL and a branch), modify (normally an increment) and store-conditional (normally includes SC and a branch) form the lock acquire section. The unlock operation at the end is a simple store operation. In Fig. 2b, we show three nodes W1, W2 and W3 competing to acquire a lock. We assume that all three nodes have successfully executed the LL and modify sections and are all trying to execute the SC. The upgrade request from W1 (message 1) arrives at the home node H first. The home replies to W1 with an upgrade acknowledgment (message 2a) and invalidates the cached copies of the line in W2 (message 2b) and W3 (message 2c). The invalidation acknowledgments are not shown for brevity. The request from W2 (message 3) gets forwarded to W1 (message 3a) while the request from W3 (message 4) gets forwarded to W2 (message 4a). The SC of W1 will succeed in the first attempt while that of W2 will fail in the first attempt because of the invalidation message (the invalidation resets the lock bit in the cache controller). Assume that the reply message 3b to W2 carries the cache line with the released lock (i.e., by the time the intervention message 3a arrives at W1 it has already executed the critical section and released the lock). Therefore, the second LL attempt by W2 will not cause a network transaction. But, it may happen that before W2 gets a chance to execute the second SC, the intervention from W3 (message 4a) takes away the cache line. Such a situation can hurt the performance of read-modify-write operations in large-scale systems where the number of failed store-conditionals may increase dramatically. While this situation can also arise in normal protocols, aggressive write string forwarding increases the probability of this happening and, as we will show in Section 5, we observe this problem in practice.

To solve this problem, we propose simple microarchitectural changes to delay the intervention in such situations (so that W2’s second SC will succeed). For detecting this situation, we introduce one bit of state that indicates whether the last SC succeeded. An incoming intervention looks up the L1 cache tag RAM to decide whether the line is in the dirty exclusive state, compares the intervention cache line address to the contents of the lock address register, and finds whether the last SC failed. If all these checks match then we have detected a potential intervention conflicting with an upcoming SC that may succeed—provided we delay the intervention. In this case, we block the intervention, maintain a pending intervention state bit, and initialize a one bit LL_loop_counter to zero to be used by a failing LL instruction to decide whether to unblock an intervention. The execution of an LL instruction is changed as follows: If there is a pending intervention and the LL_loop_counter is zero, the LL instruction sets the counter to 1. If there is a pending intervention and the counter is 1, the LL instruction failed to pass the branch and hence the SC will not be executed until the lock-holder releases the lock. At this point the pending intervention is unblocked and the pending intervention state is cleared. Finally, a graduating SC instruction always unblocks any pending intervention and clears the pending intervention state bit. This hardware optimization improves the performance of any protocol with aggressive write forwarding, including one of our two new protocols discussed in Section 3.2.

Our solution does not use the time-out technique to unblock pending interventions proposed in [26]. Instead, our technique relies on the semantics of simple read-modify-write operations and works equally well for any read-modify-write operations (e.g., lock acquire, centralized flat barrier, etc.). However, our solution assumes that the code between the LL and SC does not depend on the execution of other nodes in the system (such as wait on a shared flag etc.). This is not restrictive since if this condition is not met it is unclear whether even a conventional LL/SC implementation would make any forward progress due to the large amount of time spent between the LL and the SC. However, our technique can be easily augmented with a fallback time-out mechanism where a counter is initialized upon completion of a successful LL and is incremented on every cycle until an SC graduates. As soon as the counter crosses a threshold any pending intervention is unblocked. This technique is different from the one used in [26] and still provides fast execution for the common case of properly written LL/SC sequences.

Finally, a different problem may arise with write string forwarding if the critical section shown in Fig. 2b is relatively long. In this case the reply message 3b to W2 from W1 may bring in the cache line with the lock, causing the unlock of W1 to suffer a cache miss which in turn will invalidate all the cached copies in the other nodes (by this time all the other competing nodes would be looping on an LL) generating more network traffic. Note that without write string forwarding the delay introduced by the ownership transfer messages may actually cause the next intervention to be sent to the lock-holder after it has already released the lock leading to a timely critical section execution. We could augment our technique with IQOLB [26] for solving this problem, but that is not the central focus of this study.

3.1.2 Dirty Sharing

Write string forwarding eliminates the pending dirty exclusive state and the associated ownership transfer messages. Dirty sharing eliminates the pending shared state and the sharing writeback messages. The protocol maintains an owner for the cache lines in the shared state if the home node does not have the most updated version of the line. Thus, a write followed by a string of reads causes the owner to ripple along the chain just as in write string forwarding. This is shown in Fig. 2c. Initially, the node D is the dirty exclusive owner. Along the read string, the shared ownership ripples from R1 to R3 through R2. Though this eliminates sharing writeback messages, it may convert many potential two-hop transactions into slow three-hop ones since all read requests now have to be forwarded to an owner even though the line is shared. Although this optimization favors migratory sharing, as we will find in Section 5 it may hurt large-scale producer-consumer sharing patterns, especially if many consumers try to simultaneously access the produced value. This kind of sharing pattern is commonly observed in contended lock acquire phases where every node becomes the producer in turn while the number of consumers gradually decreases as the processors enter and exit critical sections. Note that unlike the situation in write forwarding, the reason for this poor performance is inherent in the observed sharing pattern. Further, this optimization will hurt performance even more with the current trend of large L2 and L3 caches causing cache lines to be written back less frequently and increasing the likelihood of three-hop interventions.

3.2 Buffering at the Home Node

This section presents our two NACK-free protocol designs. Instead of resorting to dirty sharing, our protocols buffer the pending requests at the home node. We reserve space in the protocol portion of main memory to store pending requests and make this space selectively visible to the dispatch unit only when appropriate.

3.2.1 Mechanism

Our protocols store pending read and write requests in memory in two separate *pending lists*. The protocol execution builds directly upon our modified SGI Origin 2000 protocol. The only differences are in the execution of the request handlers that find the directory entry in a pending state and in the execution of the handlers that clear the pending states. A request finding the directory entry in a pending state, where it would be NACKed in the Origin 2000 protocol, instead gets stored either in the directory entry (if it is the first read) or in one of the pending lists depending on the type of request (read or write). The message that clears the pending state (e.g., a sharing writeback) first sends out two pending read replies or one pending write reply if there is no pending read. This design decision favors short pending lists without making the protocol too complex. At this time, a message is also enqueued on the software queue to handle any remaining pending requests, making the pending requests visible to the dispatch unit. When the message on the software queue gets selected, the corresponding handler gives priority to the pending reads and first walks through the pending read list sending out as many replies as possible given the available outgoing network queue space. Note that this handler reads the requested cache line only once from memory into a data buffer and uses that data buffer to send out all the replies. We call this scheme *read combining*. This leads to a reduction in both memory occupancy and average handler execution time. However, due to limited space in the outgoing reply lane, aggressive read combining requires extra design considerations. Although reply messages are meant to travel only on the reply lane, in the software queue message handlers it is safe to send replies on any lane. This is deadlock-free because 1) replies are guaranteed to drain and 2) the draining of the incoming queues does not depend on the state of the software queue. In other words, virtual lane usage policy of the software queue messages does not introduce a cycle in the lane allocation dependence graph. Therefore, in our design we use three of the four virtual lanes to aggressively empty the pending read chain. Inspection of the protocol code shows that pending read replies can be generated at a sustained peak rate of one reply every 17 protocol processor cycles.

After the pending read list empties, the software queue message handler sends pending write replies one at a time. If more than one write is pending, the second write will generate an intervention to the first writer and the directory will transition to the pending dirty exclusive state. In general, if at any point during the execution of the software queue handler the directory entry transitions to a pending state, the handler finishes execution and retires. The subsequent message that clears the pending state of the directory entry is responsible for scheduling another software queue message if the pending lists are not empty.

An interesting feature of this protocol is that since it treats the pending reads and writes separately, the order in which replies are sent may not correspond to the order of request arrival. However, this does not affect correctness because the order among the pending requests is determined only when the controller updates the directory and sends the reply.

We augment the protocol discussed above with aggressive write combining and develop a protocol with both read and write combining enabled by write string forwarding. In the protocol discussed above, a string of pending writes (in fact, the second one in a string) will transition the directory state to pending dirty exclusive, preventing any further pending requests from being served. In our second protocol, we solve this problem by making use of write string forwarding, as discussed in Section 3.1.1. We eliminate the pending dirty exclusive state from the directory entry and let the software queue message handler aggressively forward a string of pending writes until the outgoing network queues fill. We call this *write combining*, which opens up the additional opportunity of write string forwarding. However, since intervention messages generate replies, they cannot be sent on arbitrary virtual lanes. We therefore use two lanes out of four during write combining. As discussed in Section 3.1.1, without our delayed intervention optimization, this protocol may suffer from the adverse performance effects of write string forwarding related to atomic read-modify-write operations and long critical sections.

Finally, request combining could also be done in hardware in the incoming request lane(s) of the network interface. But due to the small incoming queue sizes and the service rate of the controller, there is little opportunity to combine messages in the NI. In Section 5, we will show that the maximum number of combined requests achieved by our protocols far exceeds any reasonable size of network interface queues. Also, request combining in the network interface would require an associative search on the addresses of the incoming requests thereby increasing the inbound latency of the network interface.

3.2.2 Implementation

In the following, we present the implementation details of our NACK-free protocols. Each pending list entry has a nominal storage overhead of 20 bytes including an entry index, the requesting node number, a next pointer, an upgrade bit (relevant for the write pending list only) and some system-specific information. Sizing the pending list is important for performance. The theoretical limit on the size is given by the maximum number of requests that the system can generate at any point of time. Assuming P nodes, $\text{Size}_{\text{MSHR}}$ miss status holding registers in the processor, and Size_{OTT} slots in the OTT, this limit turns out to be $P * \min(\text{Size}_{\text{MSHR}}, \text{Size}_{\text{OTT}})$. But in practice, contention happens for only one cache line. In that case, P entries would be sufficient in each list. Our protocol currently supports up to 128 entries in each list requiring 5 KB of total DRAM storage for two lists. If at any point the protocol runs out of pending list storage it resorts to NACKs until at least one pending list entry is available. Although in our experiments this case never arises, we can modify the design to accommodate larger lists in DRAM to keep the protocols truly NACK-free.

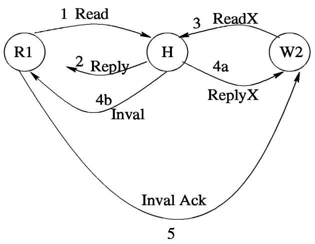


Fig. 3. Read-invalidate race.

The directory entry is the same as that in the SGI Origin 2000 protocol discussed in Section 2.2 except that the unused 28 bits are used to maintain states regarding the pending requests. The directory entry stores the starting indices (7 bits each) of the read and write pending lists for the corresponding cache line. Note that the reserved space for the pending lists acts as a centralized pool of pending list entries for a particular node and is not allocated for each directory entry. When a directory entry needs to queue a pending request it tries to get one pending list entry from the pool; if it fails it sends a NACK to the requester. Two bits in the directory entry are needed to indicate whether the pending lists are empty or not, and one more bit indicates whether a pending list handler has already been scheduled on the software queue. To favor short sharing sequences the remaining 11 bits are used for storing the first pending reader in the directory entry itself (7 bits for reader node number, one valid bit, and 3 bits for system-specific information about the requested address).

3.3 Residual Negative Acknowledgments

In these NACK-free protocols there remain a very small number of NACKs arising from what we call read-invalidate races. We show one such race in Fig. 3.

The read request from `R1` (message 1) gets replied to by the home (message 2) but the reply gets delayed in the network. In the meantime, `W2` sends a write request (message 3), the home replies (message 4a) and also sends an invalidation request to `R1` (message 4b). Since the invalidation requests and the replies travel along different network lanes, the invalidation can race past the read reply. On receiving the invalidation, the OTT in `R1` marks the outstanding read invalidated and acknowledges the invalidation (message 5). Eventually the read reply arrives, but since the invalidation has already been acknowledged the replied data cannot be used and still guarantee write atomicity. In this case, the processor interface in `R1` sends a local NACK to the processor instead of sending the read reply. This is the solution used in the Stanford DASH multiprocessor [21], [22]. In the AlphaServer GS320 some read-invalidate races are eliminated by sending a marker message to the requester as soon as the request arrives at the home node. All invalidations for an outstanding read arriving before the marker are ignored. We decided not to introduce the extra marker messages in the system given the extremely small number of read-invalidate races as we find in Section 5. Also, the invalidation messages need to be ordered with the marker messages necessitating point-to-point ordering in the network, which none of our protocols rely on.

TABLE 1
Summary of Evaluated Protocols

Protocol Name	Description	Source of NACKs
BaseBV	Baseline bitvector (Section 2.1)	Home and third party nodes
OriginMod	Modified SGI Origin 2000 (Section 2.2)	Home node only
OriginMod+RComb	Read combining, but no write combining (first part of Section 3.2.1)	Read-invalidate races only
OriginMod+RWComb+WSF	Read and write combining (second part of Section 3.2.1)	Read-invalidate races only
OriginMod+RWComb+WSF+OPT	Previous one with delayed intervention optimization (OPT)	Read-invalidate races only
OriginMod+DSH+WSF(+OPT)	Dirty sharing and write string forwarding (Sections 3.1.1 and 3.1.2)	Read-invalidate races only

3.4 Summary of Protocols

Table 1 summarizes the six protocols that we evaluate. Since the OriginMod+DSH+WSF(+OPT) protocol may return a cache line to the reader in the owned state, the conventional read-modify-writes implemented with LL/SC may livelock. Therefore, for the applications with read-modify-writes we turn on our delayed intervention optimization (OPT) to guarantee forward progress. Note that this also eliminates the bad effects of write string forwarding that this protocol would have otherwise. This protocol supports four L2 cache states, namely, M, O, S, and I. Other protocols do not support an O state. All the protocols other than BaseBV collect invalidation acknowledgments at the writer and require writeback acknowledgments. BaseBV collects invalidation acknowledgments at the home and does not require writeback acknowledgments.

4 Evaluation Methodology

This section discusses the applications and the simulation environment we use to evaluate our protocols.

4.1 Applications

Table 2 shows six programs selected from the SPLASH-2 benchmark suite [30]. There are three complete applications (Barnes-Hut, Ocean, and Water) and three computational kernels (FFT, LU, and Radix-Sort). The programs represent a variety of important scientific computations with different communication patterns and synchronization requirements. As a simple optimization, in Ocean the global error lock in the multigrid phase has been changed from a lock-test-set-unlock sequence to a more efficient test-lock-test-set-unlock sequence [17].

4.2 Simulation Environment

We present detailed results for 64-node systems and selected results for 32 and 128-node systems. The main processor runs at 1 GHz and is equipped with separate 32 KB primary instruction and data caches that are two-way set associative and have line sizes of 64 bytes and 32 bytes respectively. The secondary cache is unified, 2 MB, two-way set associative and has a line size of 128 bytes. The processor ISA includes prefetch and prefetch exclusive instructions and the cache controller uses a critical double-word refill scheme. The processor model also contains fully-associative 8-entry instruction TLB, 64-entry data TLB, and 4 KB pages. We accurately model the latency and cache effects of TLB misses. On two different occasions our processor model has been validated against real hardware [5], [9].

The embedded protocol processor is a dual-issue core running at 400 MHz system clock frequency. The instruction and data cache behavior, and the contention effects of the protocol processor are modeled precisely via a cycle-accurate simulator similar to that for the protocol processor in [9]. We simulate a 32 KB direct-mapped protocol instruction cache and a 512 KB direct-mapped protocol data cache. The access time of main memory SDRAM is fixed at 125 ns (50 system cycles). The memory queue is 16 entries deep. The input and output queue sizes in the memory controller's processor interface are set at 16 and 2 entries, respectively. The corresponding queues in the network interface are 2 and 16 entries deep. The network interface is equipped with four virtual lanes to aid deadlock-free routing. The processor interface has an 8-entry outstanding transaction table and a 4-entry writeback buffer. Each of the read and write pending lists in each node has 128 entries as discussed in Section 3.2.2. Each node

TABLE 2
Applications and Problem Sizes

Applications	Problem Sizes
Water	1024 molecules, 3 time steps
Barnes Hut	8192 particles, 3 time steps
LU	512×512 matrix, 16×16 blocks
Ocean	514×514 grid
Radix-sort	2M keys, radix=32
FFT	1M points

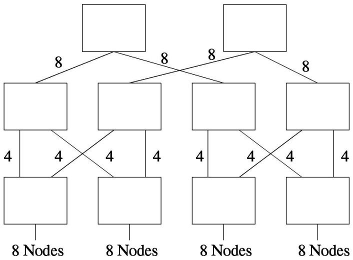


Fig. 4. An example 32-node fat tree topology using 10 16-port crossbar switches.

controller has 32 cache line-sized data buffers used for holding data as a protocol message passes through various stages of processing.

We present results for three network configurations to understand the effects of different topologies and network speeds. The slowest configuration, named `FT150ns`, uses a fat tree topology connecting crossbar switches containing 16-ports each with a hop time of 150 ns. The fastest configuration, named `FT50ns`, is the same as `FT150ns` but has a hop time of 50 ns. The fat tree topology presents an economical way of using large crossbar switches (e.g., with 16 ports, but a relatively large hop time) to build a scalable network. Fig. 4 shows an example 32-node topology using only 10 switches. The numbers beside the links show the number of wires. The 64-node and 128-node topologies (not shown for brevity) used in this paper need 20 and 40 switches, respectively. We also present results for a medium-speed two dimensional mesh topology, named `Mesh50ns`, using 6-port switches like the SGI Spider router [7]. Although it has a 50 ns hop time, it is less scalable than `FT50ns`. For all three configurations, the simulated node-to-network link bandwidth is 1 GB/s.

5 Simulation Results

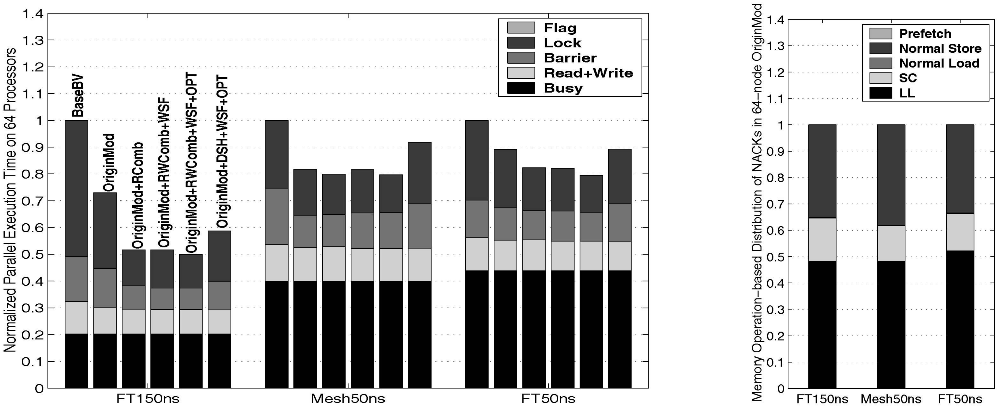
This section presents detailed simulation results for five selected applications running on a 64-node system with uniprocessor nodes. Later, in Section 5.7, we present selected results for 128 and 32-node systems. The details of the results for FFT are omitted due to space constraints, but a summary is provided in Section 5.6. All six bitvector protocols discussed in Section 3.4 scale by becoming coarsevector protocols [11] with a coarseness of 2 and 4 for 64 and 128-node systems respectively. The chosen applications exhibit acceptable scalability up to 64 nodes. The speedup of Water, Barnes Hut, LU, Ocean, Radix-Sort, and FFT for the `OriginMod` protocol on the `FT50ns` topology and 32 nodes is 24.8, 27.2, 21.3, 55.1 (superlinear due to cache and TLB effects), 26.7, and 31.4, respectively. On 64 nodes the speedup numbers are 32.0, 45.1, 32.5, 69.5, 46.6, and 55.7, respectively. Our NACK-free protocols, by reducing the parallel execution time, improve the scalability even further. However, with the input sizes used in this study none of the applications achieve speedup of even 64 (50 percent parallel efficiency) on 128 nodes with the `OriginMod` protocol. Ocean and FFT are the only two applications showing efficiency close to 50 percent. Since the performance of FFT is not affected much by NACKs, we will only discuss the results for Ocean on 128 nodes and show that our NACK-free protocol significantly improves scalability (on `FT50ns` configuration speedup improves from 55.0 with `OriginMod` to 70.4 with `OriginMod+RComb`).

5.1 Water

This section presents the simulation results for Water on 64 nodes. This application is optimized with page placement and software tree barriers, but does not use software prefetch.

Fig. 5a presents the execution time normalized to `BaseBV` for three network configurations, while Fig. 5b presents the distribution of NACKs based on which load/store instructions (i.e., `LL`, `SC`, other loads, other stores and prefetches) are negatively acknowledged in the `OriginMod` protocol. We divide the execution time into busy cycles, read and write stall cycles, and synchronization cycles. We further break down the synchronization time into time spent on lock acquires, barriers and flags. Note that Water does not use flags. The network configurations are arranged from left to right in increasing order of speed as indicated by increasing busy cycle percentages.

For the `FT150ns` configuration, the `OriginMod` protocol runs 37 percent faster than `BaseBV` (corresponding to a speedup of 1.37) while the `OriginMod+RComb` protocol is



(a) (b)

Fig. 5. (a) Normalized execution time. (b) Distribution of NACKs for Water.

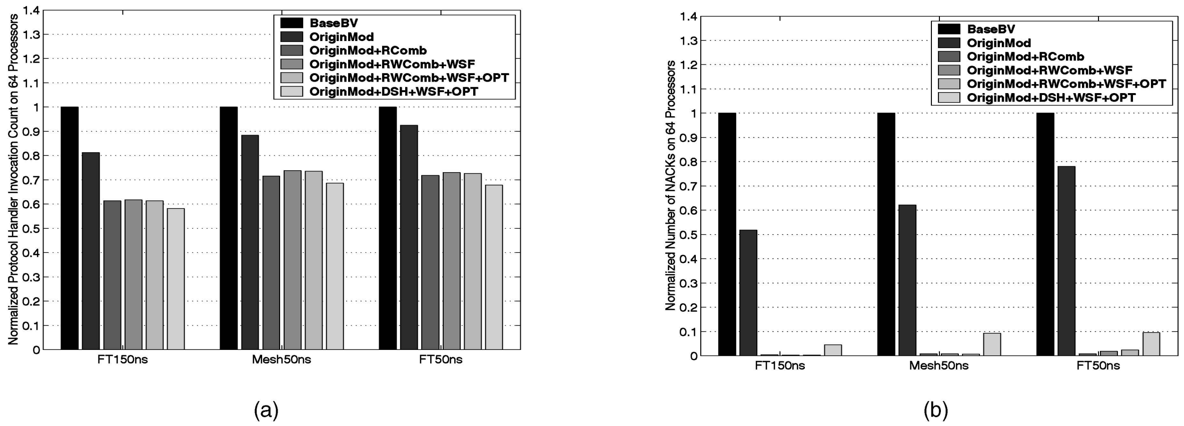


Fig. 6. (a) Normalized protocol handler invocation count. (b) Normalized NACK count for Water.

93.4 percent faster than BaseBV and 41.2 percent faster than OriginMod. Compared to OriginMod, most of the benefits of OriginMod+RComb come from reducing the lock time while the rest comes from reducing the barrier time. It is clear that eliminating the 48.3 percent of the NACKs that are from LL instructions and the 16.4 percent coming from SC instructions, as shown in Fig. 5b, helps accelerate lock acquires. Note that even if a scheme like IQOLB could achieve a similar performance benefit for lock acquires it could not eliminate the rest of the NACKs arising from normal loads and stores. The reduction in barrier time is due to better load balance resulting from elimination of these remaining 35.3 percent of NACKs. Adding write string forwarding to OriginMod+RComb does not affect performance while the delayed intervention optimization executes 3.3 percent faster. The OriginMod+RComb protocol is 13.6 percent faster than the OriginMod+DSH+WSF+OPT protocol due to the latter's increased time spent on lock acquires resulting from many three-hop misses as discussed in Section 3.1.2.

For the Mesh50ns configuration, the OriginMod+RComb protocol executes 25 percent faster than BaseBV and 2.1 percent faster than OriginMod. For the FT50ns configuration, this protocol runs 21.4 percent faster than BaseBV and 8.2 percent faster than OriginMod.

We note that in OriginMod+RComb for all the three network configurations the maximum number of combined reads in one handler invocation is 60, while in OriginMod+RWComb+WSF, the maximum number of forwarded writes in one handler invocation is 39 for FT150ns, 56 for Mesh50ns, and 48 for FT50ns. Other applications show similar trends.

Fig. 6a shows the total count of executed protocol message handlers normalized to BaseBV. This metric has direct correlation with the message count in the system. For all three network configurations OriginMod+DSH+WSF+OPT achieves the lowest message count due to elimination of sharing writebacks and ownership transfers, but an increased number of three-hop transactions in the lock acquire phases hurts the performance of this protocol. For example, in the FT150ns configuration this protocol has 29.2 percent more three-hop read misses than the OriginMod+RComb protocol. Fig. 6b shows the normalized count of NACKs. OriginMod achieves a substantial reduction in the NACKs over BaseBV: 48.1 percent for FT150ns, 37.8 percent for Mesh50ns and 21.9 percent for FT50ns. The NACKs in the remaining four cases are solely due to read-invalidate races. The OriginMod+DSH+WSF+OPT protocol suffers from the maximum number of such races due to aggressive write and read forwarding.

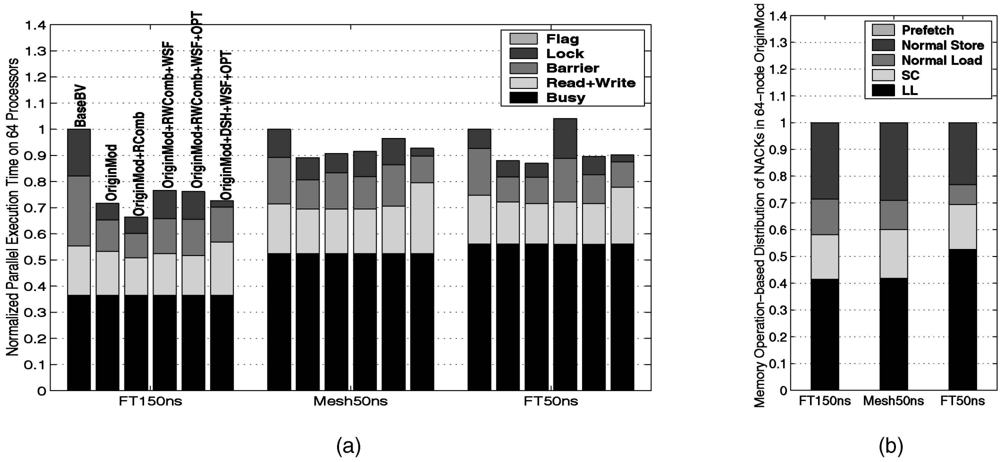
5.2 Barnes Hut

This section presents the results for two versions of Barnes Hut on 64 nodes—one with 64K array-locks and the other with 2048 array-locks used to protect the cells of the octtree. Both versions use page placement and software tree barriers, but no software prefetch.

5.2.1 *Barnes-Hut with 64K Locks*

Fig. 7a presents the execution time normalized to BaseBV for three network configurations while Fig. 7b presents the distribution of NACKs in the OriginMod protocol. For the FT150ns configuration the OriginMod protocol executes 39.5 percent faster than BaseBV while the OriginMod+RComb protocol is 50.6 percent faster than BaseBV and 7.9 percent faster than OriginMod. Compared to OriginMod most of the benefits of OriginMod+RComb come from reducing the barrier time and therefore improving load balance by eliminating NACKs, while the rest comes from reducing the read/write stall time. As shown in Fig. 7b, 41.4 percent of the NACKs arise from LL instructions and 16.7 percent from SC instructions. While the elimination of these NACKs accelerates lock acquires, the remaining 43 percent of the NACKs play the most significant role. Adding write string forwarding to OriginMod+RComb increases the execution time significantly and OriginMod actually runs 6.9 percent faster compared to OriginMod+RWComb+WSF due to the latter's increased failed store-conditionals leading to an increased lock stall time. The OriginMod+DSH+WSF+OPT protocol performs best in terms of lock acquire, but continues to suffer from three-hop misses leading to an increased read/write stall time. The improved lock acquire performance is due to less contended locks compared to Water and extremely low occupancy achieved by this protocol on this application (see below).

For the Mesh50ns configuration, the OriginMod protocol emerges the best, executing 12.2 percent faster than BaseBV.



(a)

(b)

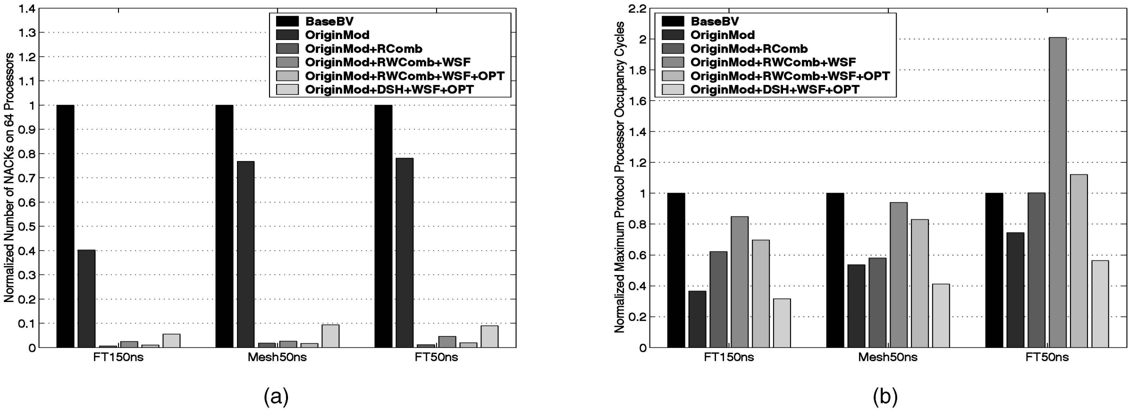
Fig. 7. (a) Normalized execution time. (b) Distribution of NACKs for Barnes Hut with 64K locks.

All other protocols perform worse compared to OriginMod with OriginMod+RComb being the closest yielding 1.8 percent increased execution time over OriginMod. For the FT50ns configuration the OriginMod+RComb protocol is 14.9 percent faster than BaseBV and 1.2 percent faster than OriginMod. The OriginMod+RWComb+WSF protocol suffers greatly from an increased count of failed store-conditionals and even BaseBV runs 4.1 percent faster compared to it. The delayed intervention optimization effectively eliminates many failed store-conditionals and helps bring down the execution time, but still OriginMod executes 1.8 percent faster than OriginMod+RWComb+WSF+OPT. This is due to relatively long critical sections that compute the forces between the bodies. As pointed out in Section 3.1.1, aggressive write forwarding may hurt performance of relatively long critical sections. Regarding the amount of request combining, we note that in OriginMod+RComb the maximum number of combined reads in one handler invocation is 35 for FT150ns, 40 for Mesh50ns and 30 for FT50ns while in OriginMod+RWComb+WSF the maximum number of forwarded writes in one handler invocation is 21 for FT150ns, 25 for Mesh50ns, and 42 for FT50ns. Thus, a large amount of request combining continues to improve the performance of our protocol.

Fig. 8a shows the normalized count of negative acknowledgments. Fig. 8b shows the normalized protocol processor occupancy cycles on the most contended node. The notably high occupancy of OriginMod+RWComb+WSF for FT50ns explains its poor performance. Also, OriginMod+DSH+WSF+OPT shows extremely low occupancy which results from resolving the three-hop races at the periphery and keeping the home nodes free as much as possible. But, the large-scale producer-consumer sharing patterns continue to result in poor performance with this protocol. Finally, Fig. 9a shows the normalized dynamic count of executed SC instructions. In the OriginMod+RWComb+WSF protocol the number of failed store-conditionals increases enormously due to aggressive write string forwarding.

5.2.2 Barnes-Hut with 2,048 Locks

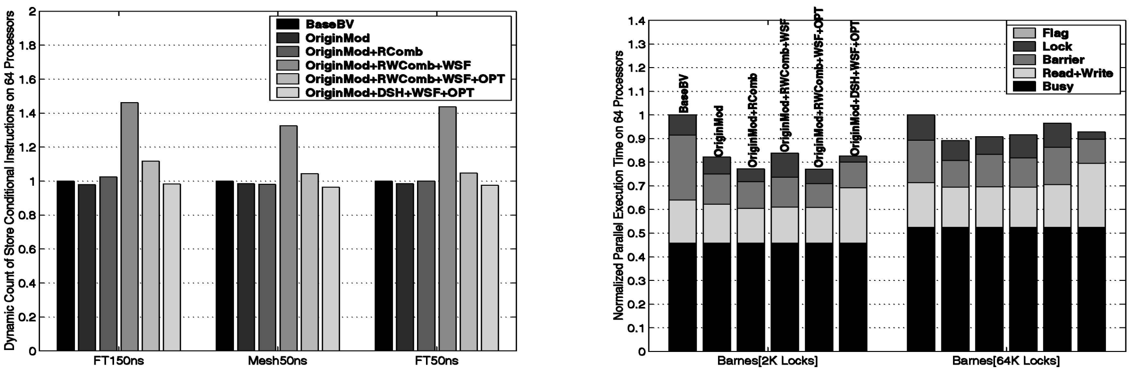
Since the majority of NACKs arise from lock acquires, we experimented with a smaller number of array locks. With 2,048 locks the lock contention for each cell in the oct-tree structure is expected to increase. We show the normalized execution time for the Mesh50ns configuration in Fig. 9b. For comparison we also present the results with 64K locks alongside. Clearly, with 2,048 locks all the protocols show greater performance improvement relative to BaseBV as



(a)

(b)

Fig. 8. (a) Normalized NACK count. (b) Normalized protocol processor occupancy cycles for Barnes Hut with 64K locks.



(a) (b)

Fig. 9. (a) Normalized dynamic count of SC instructions for Barnes Hut with 64K locks. (b) Normalized execution time for Barnes Hut with 2,048 locks on Mesh50ns.

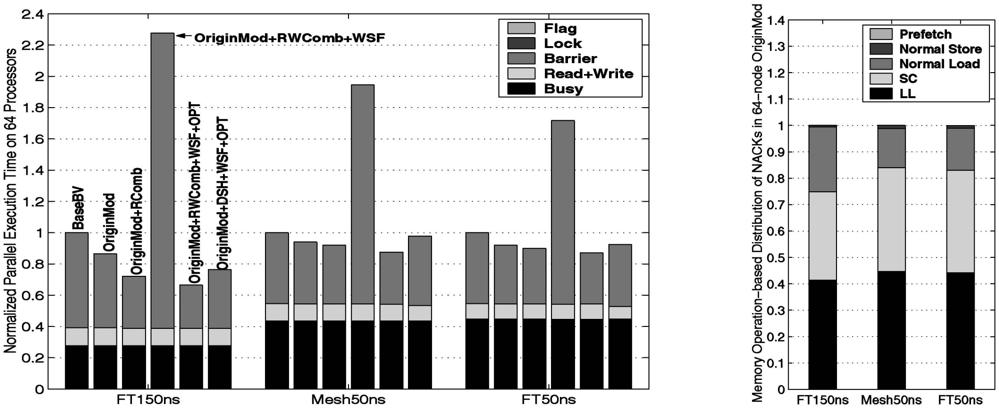
compared to that with 64K locks. The OriginMod+RComb protocol is 29.5 percent faster than BaseBV and 6.4 percent faster than OriginMod. A comparison between the performance of OriginMod+RComb with 64K and 2,048 locks clearly brings out the importance of this protocol in the presence of lock contention. Adding write combining and write string forwarding to OriginMod+RComb continues to hurt performance because of failed store-conditionals. The delayed intervention optimization (OriginMod+RWComb+WSF+OPT) achieves a parallel execution time very close to that of OriginMod+RComb.

5.3 LU

This section presents the results for LU on 64 nodes. Optimized LU shows little variation across different protocols. This agrees with the findings from other protocol studies [14], [15]. Therefore, in the following we present results for an unoptimized version of LU, representative of less-tuned parallel programs. In unoptimized LU, we turn off page placement (i.e., rely on a default round robin placement policy) and software prefetch and instead of point-to-point synchronization, we use more costly centralized barriers implemented with atomic read-modify-write via LL/SC.

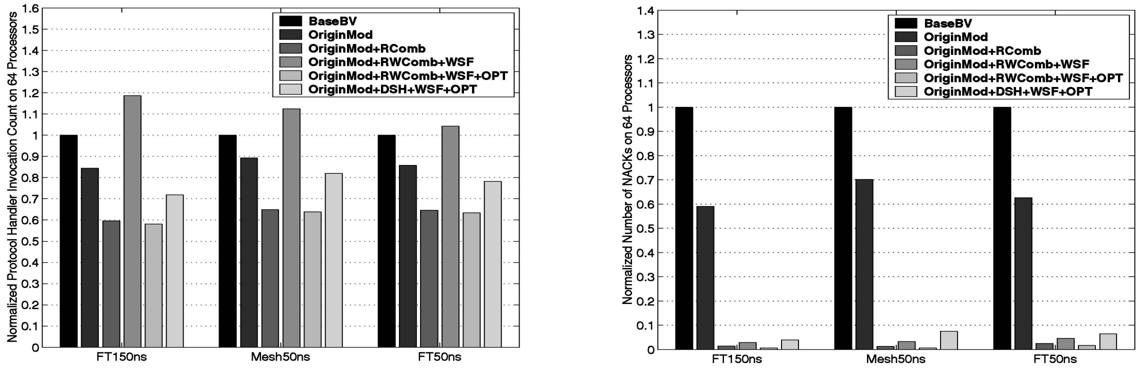
Fig. 10a presents the execution time normalized to BaseBV for three network configurations while Fig. 10b presents the distribution of NACKs in the OriginMod protocol for unoptimized LU. Due to heavily contended execution of centralized barriers (which rely on atomic read-modify-write operations), a large fraction of NACKs arise from LL and SC instructions: 74.8 percent in FT150ns, 84 percent in Mesh50ns, and 83 percent in FT50ns. Another interesting observation is that the OriginMod+RWComb+WSF protocol shows extremely poor performance. For FT150ns, it runs 2.28 times slower compared to BaseBV, while for Mesh50ns and FT50ns the slowdown is, respectively, 1.95 and 1.72. Increased count of failed SC is the reason for this.

For the FT150ns configuration, the OriginMod protocol runs 15.7 percent faster than BaseBV while the OriginMod+RComb protocol is 38.7 percent faster than BaseBV and 19.8 percent faster than OriginMod. Compared to OriginMod most of the benefits of OriginMod+RComb come from reducing the barrier time. The reduction in the barrier time mostly results from accelerated read-modify-write operations. Adding write string forwarding to OriginMod+RComb increases the execution time significantly as already discussed. However, the delayed intervention optimization executes



(a) (b)

Fig. 10. (a) Normalized execution time. (b) Distribution of NACKs for unoptimized LU.



(a) (b)

Fig. 11. (a) Normalized protocol handler invocation count. (b) Normalized NACK count for unoptimized LU.

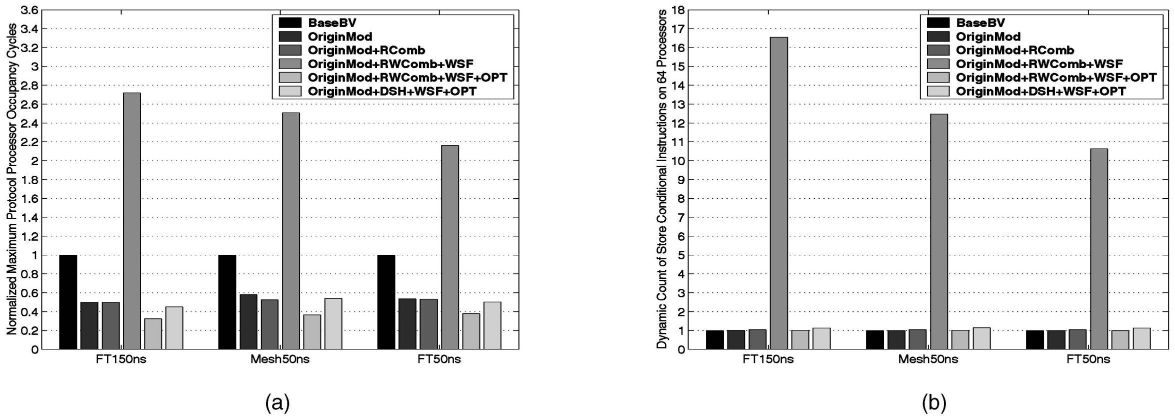
8.5 percent faster compared to OriginMod+RComb and clearly performs better than all other five cases. The OriginMod+DSH+WSF+OPT continues to suffer from an increased barrier time resulting from a large number of three-hop read misses. In this protocol, 83.5 percent of all read misses are three-hop misses which is 7.1 times larger than the number of three-hop read misses in OriginMod+RComb. This is due to the heavily contended producer-consumer sharing pattern observed by the read-modify-write variable implementing the centralized barrier. For the Mesh50ns and the FT50ns configurations, the same trend is observed. We also note that in OriginMod+RComb the maximum number of combined reads in one handler invocation is 60 for FT150ns, 56 for Mesh50ns and 39 for FT50ns, while in OriginMod+RWComb+WSF, the maximum number of forwarded writes in one handler invocation is 34 for FT150ns, 57 for Mesh50ns and 51 for FT50ns.

Fig. 11a shows the protocol handler invocation count normalized to BaseBV. For all three network configurations, OriginMod+RWComb+WSF has the highest message count. Note again the increased message count of OriginMod+DSH+WSF+OPT resulting from a large number of three-hop transactions that outweigh the reduction achieved due to elimination of sharing writeback and ownership transfer messages. Fig. 11b shows the normalized count of NACKs. Fig. 12a shows the normalized protocol processor occupancy cycles on the most contended node. The notably high occupancy for OriginMod+RWComb+WSF across all three network configurations explains its poor performance. Finally, Fig. 12b shows the normalized dynamic count of executed SC instructions. For all three network configurations OriginMod+RWComb+WSF shows a significantly higher number of store-conditionals than the other five cases. In fact, compared to BaseBV it has 16.6 times (FT150ns), 12.5 times (Mesh50ns), and 10.6 times (FT50ns) the number of store-conditionals. The delayed intervention optimization helps bring down the SC count to a value close to the other cases.

5.4 Ocean

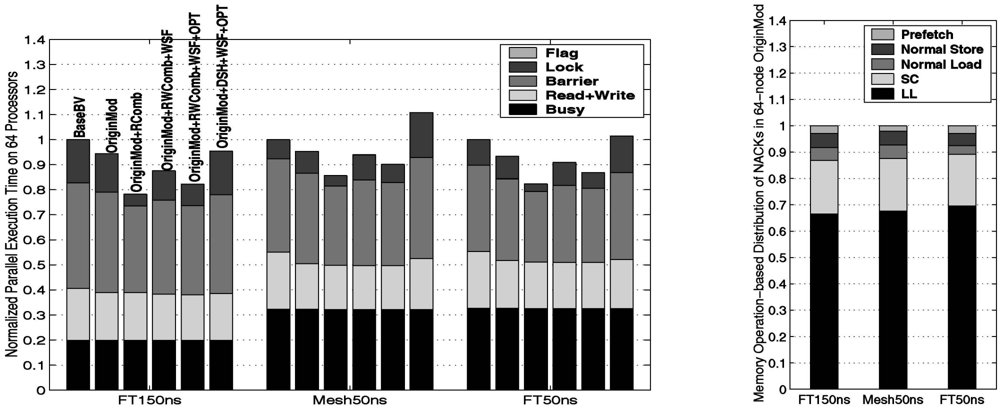
This section presents the results for Ocean on 64 nodes, optimized with page placement, software prefetching and software tree barriers.

Fig. 13a presents the execution time normalized to BaseBV for three network configurations, while Fig. 13b presents the distribution of NACKs in the OriginMod protocol. From Fig. 13b, we note that for FT150ns 86.9 percent of the NACKs result from LL and SC instructions while for Mesh50ns and FT50ns the percentages are 87.6 percent and 89.1 percent,



(a) (b)

Fig. 12. (a) Normalized protocol processor occupancy cycles. (b) Normalized dynamic count of SC instructions for unoptimized LU.



(a) (b)

Fig. 13. (a) Normalized execution time. (b) Distribution of NACKs for Ocean.

respectively. This is due to heavily contended lock acquires in Ocean.

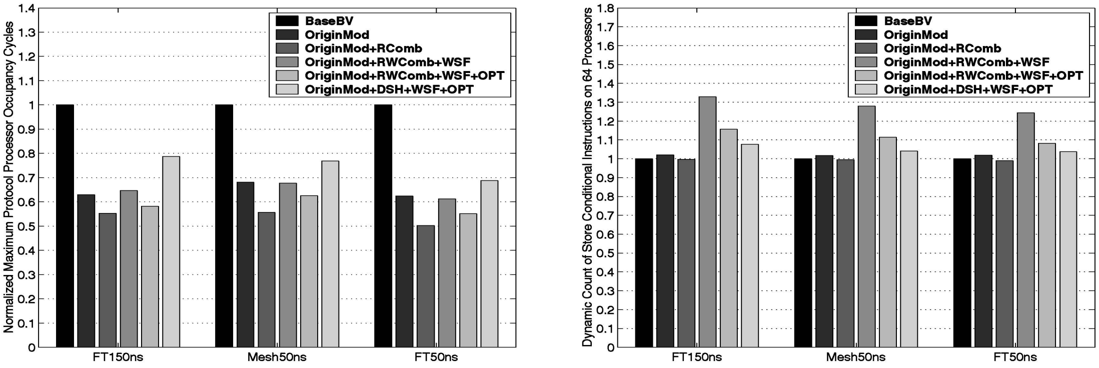
From Fig. 13a, we observe that for all three network configurations our OriginMod+RComb protocol delivers the best performance. For the FT150ns configuration the OriginMod+RComb protocol executes 27.9 percent faster than BaseBV and 20.6 percent faster than OriginMod. Compared to OriginMod most of the benefits of OriginMod+RComb come from reducing the lock time while the rest come from reducing the barrier time. The OriginMod+RWComb+WSF protocol increases the execution time significantly as aggressive write forwarding continues to hurt the performance of contended lock acquires. Adding the delayed intervention optimization solves this problem, but OriginMod+RComb is still 5.2 percent faster than OriginMod+RWComb+WSF+OPT. We found that this is due to the large critical section effect of write forwarding. Ocean has small critical sections, but due to cache misses they take a long time to execute.

For the Mesh50ns configuration, the OriginMod+RComb protocol again emerges the best, 16.8 percent faster than BaseBV and 11.4 percent faster than OriginMod. All other protocols show similar trends as those for FT150ns. For the FT50ns configuration, the OriginMod+RComb protocol continues to excel. It runs 21.4 percent faster than BaseBV and 13.4 percent faster than OriginMod. Finally, we note that in OriginMod+RComb the maximum number of combined reads in one handler invocation is 34 for FT150ns, 45 for Mesh50ns and 32 for FT50ns while in OriginMod+RWComb+WSF the maximum number of forwarded writes in one handler invocation is 40 for FT150ns, 54 for Mesh50ns and 49 for FT50ns.

Fig. 14a shows the normalized protocol processor occupancy cycles on the most contended node. The OriginMod+RComb protocol has the lowest occupancy. The notably high occupancy of the OriginMod+DSH+WSF+OPT protocol clearly brings out its inefficiency in handling large-scale producer-consumer sharing pattern. Finally, Fig. 14b shows the normalized dynamic count of executed SC instructions. For all three network configurations, the count for OriginMod+RWComb+WSF is the highest. Although adding the delayed intervention optimization helps reduce the number of failed store-conditionals, it still suffers from the effect of aggressive write combining on critical sections with large execution time.

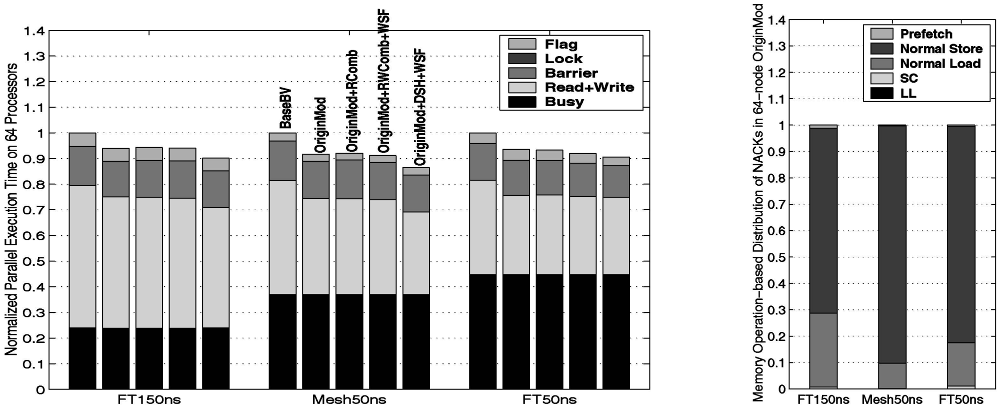
5.5 Radix-Sort

This section presents the results for Radix-Sort on 64 nodes, which uses page placement, software prefetch, software tree barriers, and point-to-point flag synchronization. The



(a) (b)

Fig. 14. (a) Normalized protocol processor occupancy cycles. (b) Normalized dynamic count of SC instructions for Ocean.



(a) (b)

Fig. 15. (a) Normalized execution time. (b) Distribution of NACKs for Radix-Sort.

delayed intervention optimization is not relevant since Radix-Sort does not have contended read-modify-writes. This optimization has practically no effect on the execution time of Radix-Sort.

Fig. 15a presents the execution time normalized to BaseBV for the three network configurations for Radix-Sort, while Fig. 15b presents the distribution of NACKs in the OriginMod protocol. As expected, almost all the NACKs arise from normal loads and stores. The interesting observation is that the OriginMod+DSH+WSF protocol emerges the best for all three network configurations. For the FT150ns configuration this protocol executes 11 percent and 4.2 percent faster than the BaseBV and the OriginMod protocols, respectively. Similar trends continue to hold for the other two configurations. We found that in Radix-Sort about 93 percent of the read misses are local, but are satisfied at a second owner node and no other application presented in this paper has such a high proportion of misses that require interventions. The dominant sharing pattern results from a remote node writing to a cache line followed by the home node reading it. Subsequently, a third node arrives at the home node with a read exclusive request for that cache line and this request can be immediately forwarded to the local processor interface (local processor is the current shared owner) without blocking at the directory in the OriginMod+DSH+WSF protocol. This is also supported by the fact that most of the NACKs arise from stores as shown in Fig. 15b. Although the OriginMod+RComb protocol is also able to eliminate these NACKs, it suffers from a slightly higher occupancy.

5.6 Summary of 64-Node Results

We have presented detailed simulation results for five applications on a 64-node DSM system. The results clearly establish the fact that for the lock-intensive applications (e.g., Water, Barnes Hut, and Ocean) and for the applications with heavily contended read-modify-write operations (e.g., unoptimized LU) our read combining protocol (OriginMod+RComb) can substantially improve the performance over a modified version of the SGI Origin 2000 protocol. For Barnes-Hut it is also able to improve performance by eliminating NACKs unrelated to LL/SC. Further, the results clearly bring out the inefficiency of the aggressive write string forwarding and dirty sharing techniques in acquiring contended locks. We summarize our findings in two tables. Table 3 presents the best protocol for each application across the three network configurations, including the results for FFT that we omitted due to space constraints. While naming the protocols we omit the OriginMod+ portion where there is no ambiguity. Closely performing protocols are considered tied.

In Table 3, other than four cases, protocols with some form of request combining emerge the best, and the OriginMod+RComb protocol is the best in 12 out of 18 cases. So, in Table 4 we summarize the speedup achieved by OriginMod+RComb, the best request combining protocol, with respect to BaseBV and OriginMod with the maximum and the minimum for each network configuration shown in bold. We note that read combining accelerates parallel execution by 6 percent to 93 percent relative to BaseBV and up to 41 percent relative to OriginMod across various network configurations. Other than a few cases of negligible slowdown (at most 2 percent), it is clear that our NACK-free protocols reduce the overall execution time significantly. Further, as the network gets slower and more contended, the relative benefit of our protocols increases in most of the cases, indicating the increased performance impact of NACKs.

5.7 Results for Other System Sizes

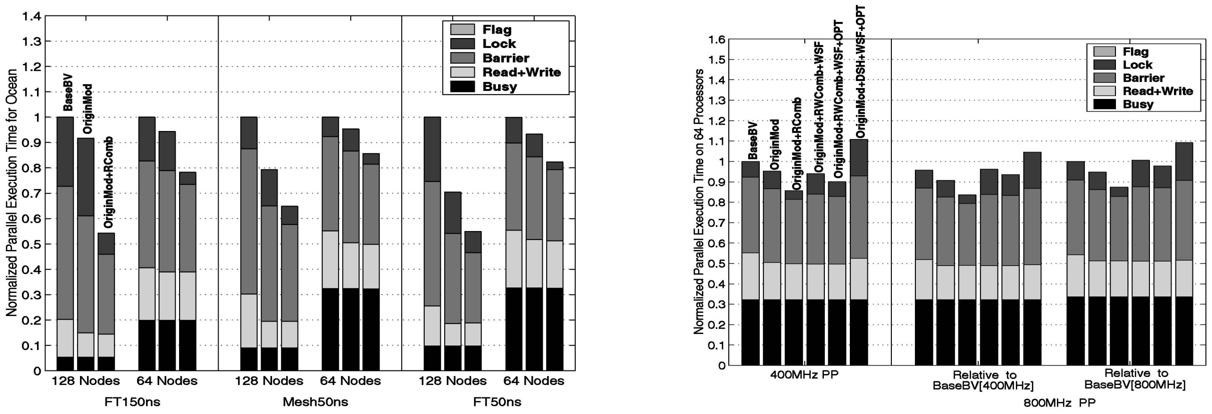
In this section, we compare the performance of OriginMod+RComb, the best request combining protocol, with BaseBV and OriginMod on 128 and 32-node systems. In Fig. 16a we show the performance of BaseBV, OriginMod and OriginMod+RComb for Ocean on 128 nodes. For comparison we have also included the results for 64 nodes. For each group of bars the execution time is normalized to the corresponding (i.e., 64 or 128-node) execution time of the BaseBV protocol. Clearly, as the system scales the relative performance benefit of NACK-free protocols in general, and our read combining protocol in particular, increases significantly. For the FT150ns configuration, on 64 nodes read combining is 17.1 percent faster than

TABLE 3
The Best Protocol

Applications	FT150ns	Mesh50ns	FT50ns
Water	RComb, RWComb+WSF, RWComb+WSF+OPT	RComb, RWComb+WSF+OPT	RComb, RWComb+WSF, RWComb+WSF+OPT
Barnes Hut (64K Locks)	RComb	OriginMod	OriginMod, RComb
LU	RWComb+WSF+OPT	RComb, RWComb+WSF+OPT	RComb, RWComb+WSF+OPT
Ocean	RComb	RComb	RComb
Radix-Sort	DSH+WSF	DSH+WSF	DSH+WSF
FFT	OriginMod, RWComb+WSF	RComb, RWComb+WSF	RComb

TABLE 4
Speedup of the Best Request Combining Protocol Relative to BaseBV and OriginMod

Applications	Relative to **BaseBV**			Relative to **OriginMod**		
	FT150ns	Mesh50ns	FT50ns	FT150ns	Mesh50ns	FT50ns
Water	**1.93**	**1.25**	**1.21**	**1.41**	1.02	1.08
Barnes Hut[64K Locks]	1.51	1.10	1.15	1.08	**0.98**	1.01
LU	1.39	**1.08**	1.11	1.20	1.02	1.02
Ocean	1.28	1.17	**1.21**	1.21	**1.11**	**1.13**
Radix-Sort	**1.06**	1.09	**1.07**	1.00	1.00	**0.99**
FFT	1.11	1.11	1.20	**0.98**	1.02	1.03



(a)

(b)

Fig. 16. (a) Normalized execution time on 128 and 64 nodes for Ocean. (b) Effect of faster PP on Ocean for 64 nodes and Mesh50ns.

OriginMod while on 128 nodes it executes 68.9 percent faster than OriginMod. We observe similar trends for the other network configurations. These results establish the fact that read combining scales much better than either the BaseBV or the OriginMod protocol.

We also looked at the effects of NACKs on medium-scale systems with 32 nodes. In these systems, the contention at the home node as well as in the network is much less leading to reduced impact of NACKs. Only Water and Ocean show some performance gain as NACKs are eliminated. For Water, the OriginMod+RComb protocol executes 3.3 percent, 2.0 percent, and 1.7 percent faster than OriginMod on FT150ns, FT50ns, and Mesh50ns configurations, respectively. For Ocean, the numbers are 3.8 percent, 2.6 percent, and 2.6 percent, respectively. This leads us to conclude that for highly scalable scientific applications NACKs are not important in small to medium-scale DSM multiprocessors while the importance increases significantly beyond 32 nodes.

5.8 Effect of Hardwired Protocol Execution

The results presented thus far assume the existence of an embedded protocol processor in the node controller that runs software code sequences to implement the coherence protocol. This technique, used in the Piranha system [3], the Stanford FLASH multiprocessor [16], [19], STiNG multiprocessor [23], S3.mp [25], etc., allows late binding of the protocol, flexibility in the choice of protocol, and a relatively easy and fast protocol verification phase. It might seem that the trends exhibited by the results will change if the protocols were implemented in hardware. But since all the protocols evaluated in this paper are essentially bitvector, a particular hardware enhancement is expected to improve the performance of all the protocols almost equally, which is tantamount to running the protocol processor faster. In Fig. 16b, we present the performance of the protocols running on a protocol processor twice as fast (i.e., 800 MHz) compared to our base protocol processor. We pick Ocean running on the `Mesh50ns` configuration for this study as a representative of fairly complex scalable applications.

The first group of bars repeats our results for our base 400 MHz protocol processor (PP) with execution times normalized to `BaseBV`. The second group of bars present execution time on a 800 MHz PP, but normalized to the execution time of `BaseBV` running on 400 MHz PP. Finally, the last group of bars present the results for an 800 MHz PP normalized to `BaseBV` running at the same frequency. The second group of bars shows that a faster PP improves the performance of all the protocols other than `OriginMod+RWComb+WSF` and `OriginMod+RWComb+WSF+OPT`. The reason for this anomaly is that with a faster PP the write forwarding becomes even more aggressive leading to an even larger number of failed store-conditionals manifested in the form of an increased lock acquire time. A comparison between the first and the last group of bars establishes the fact that the relative performance trend for `BaseBV`, `OriginMod`, `OriginMod+RComb` and `OriginMod+DSH+WSF+OPT` is largely independent of the frequency of the protocol processor.

6 Conclusions

We have presented a detailed analysis of the performance impact of negative acknowledgments on 64 and 128-node systems. We propose and evaluate two novel request combining techniques in conjunction with buffering at the home node to eliminate the NACKs that remain in a modified version of the SGI Origin 2000 protocol. The protocol with aggressive read combining at the home node achieves the best performance in a majority of the cases and shows that removing NACKs can significantly improve performance. Our protocol achieves speedup as high as 1.93 over a baseline bitvector protocol and up to 1.41 compared to a modified SGI Origin 2000 protocol on a 64-node system. In most cases the advantages of NACK-free protocols increase as the network gets slower and more contended. We also show that as the system scales to larger sizes the read combining protocol continues to achieve better scalability compared to the baseline bitvector or the SGI Origin 2000 protocol. Interestingly, our read combining protocol not only eliminates NACKs, but also significantly accelerates lock acquires in lock-intensive applications.

The second variant of our NACK-free protocol incorporates the idea of write string forwarding as in the AlphaServer GS320 and Piranha system with our read and write combining schemes. But, to our surprise, we find that aggressive write forwarding degrades the performance of heavily contended read-modify-writes and large critical sections. We propose microarchitectural changes in the cache controller to improve the performance of read-modify-writes in these protocols. It does not appear beneficial to implement write forwarding in a cache coherence protocol without supporting some form of delayed intervention optimization in the cache subsystem. Further, our evaluation of dirty sharing used in the NACK-free protocol of the Piranha chip-multiprocessor shows that this technique can greatly hurt performance in the presence of large-scale producer-consumer sharing due to an increased volume of three-hop misses. Our read combining protocol not only remains free of the problems of write forwarding and dirty sharing, but also significantly improves load balance and overall performance by effectively eliminating negative acknowledgments.

Acknowledgments

This research was supported by the US National Science Foundation CAREER Award CCR-9984314. The authors would like to thank José Martínez for his useful discussions and comments.

References

[1] D. Abts, D.J. Lilja, and S. Scott, "Towards Complexity-Effective Verification: A Case Study of the Cray SV2 Cache Coherence Protocol," *Proc. Workshop Complexity-Effective Design, 27th Int'l Symp. Computer Architecture (ISCA),* June 2000.

[2] L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proc. 25th Int'l Symp. Computer Architecture (ISCA),* pp. 3-14, June/July 1998.

[3] L.A. Barroso et al. "Piranha: A Scalable Architecture Based on Single—Chip Multiprocessing," *Proc. 27th Int'l Symp. Computer Architecture (ISCA),* pp. 282-293, June 2000.

[4] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 224-234, Apr. 1991.

[5] M. Chaudhuri et al. "Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation," *IEEE Trans. Computer,* vol. 52, no. 7, pp. 862-880, July 2003.

[6] M. Chaudhuri and M. Heinrich, "The Impact of Negative Acknowledgments in Shared Memory Scientific Applications," Technical Report CSL-TR-2003-1031, Cornell Computer Systems Lab, http://www.csl.cornell.edu/TR/CSL-TR-2003-1031.pdf, Mar. 2003.

[7] M. Galles, "Spider: A High-Speed Network Interconnect" *IEEE Micro,* vol. 17, no. 1, pp. 34-39, Jan.-Feb. 1997.

[8] K. Gharachorloo et al., "Architecture and Design of AlphaServer GS320," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 13-24, Nov. 2000.

[9] J. Gibson et al., "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 49-58, Nov. 2000.

[10] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 64-75, May 1989.

[11] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. 1990 Int'l Conf. Parallel Processing (ICPP),* pp. I.312-I.321, Aug. 1990.

[12] E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," *Proc. Fifth Int'l Symp. High Performance Computer Architecture (HPCA),* pp. 172-181, Jan. 1999.

[13] E. Hagersten, A. Landin, and S. Haridi, "DDM—A Cache-Only Memory Architecture," *IEEE Computer,* pp. 44-54, Sept. 1992.

[14] M. Heinrich, "The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," PhD dissertation, Stanford Univ., Oct. 1998.

[15] M. Heinrich et al., "A Quantitatitve Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," *IEEE Trans. Computer,* vol. 48, no. 2, pp. 205-217, Feb. 1999.

[16] M. Heinrich et al., "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 274-285, Oct. 1994.

[17] M. Heinrich and M. Chaudhuri, "Ocean Warning: Avoid Drowning," *ACM SIGARCH Computer Architecture News,* vol. 31, no. 3, pp. 30-32, June 2003.

[18] A. Kägi, D. Burger, and J.R. Goodman, "Efficient Synchronization: Let Them Eat QOLB," *Proc. 24th Int'l Symp. Computer Architecture (ISCA),* pp. 170-180, June 1997.

[19] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture (ISCA),* pp. 302-313, Apr. 1994.

[20] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture (ISCA),* pp. 241-251, June 1997.

[21] D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture (ISCA),* pp. 148-159, May 1990.

[22] D. Lenoski et al., "The Stanford DASH Multiprocessor," *IEEE Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.

[23] T.D. Lovett and R.M. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA),* pp. 308-317, May 1996.

[24] T.D. Lovett, R.M. Clapp, and R.J. Safranek, "NUMA-Q: An SCI-Based Enterprise Server," Sequent Computer Systems Inc., 1996.

[25] A. Nowatzyk et al., "The S3. mp Scalable Shared Memory Multiprocessor," *Proc. 24th Int'l Conf. Parallel Processing (ICPP),* pp. I1-I10, Aug. 1995.

[26] R. Rajwar, A. Kägi, and J.R. Goodman, "Improving the Throughput of Synchronization by Insertion of Delays," *Proc. Sixth Int'l Symp. High Performance Computer Architecture (HPCA),* pp. 168-179, Jan. 2000.

[27] P. Ranganathan et al., "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 307-318, Oct. 1998.

[28] Scalable Coherent Interface, *ANSI/IEEE Standard 1596-1992,* Aug. 1993.

[29] R. Simoni, "Cache Coherence Directories for Scalable Multiprocessors," PhD dissertation, Stanford Univ., Oct. 1992.

[30] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA),* pp. 24-36, June 1995.

Mainak Chaudhuri received the Bachelor of Technology degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur in 1999, and the MS degree in electrical and computer engineering from Cornell University in 2001, where he is currently working towards a PhD degree. His research interests include microarchitecture, parallel computer architecture, cache coherence protocol design, and cache-conscious parallel algorithms for scientific computation. He is a student member of the IEEE and the IEEE Computer Society.

Mark Heinrich received the PhD degree in electrical engineering from Stanford University in 1998, the MS degree from Stanford in 1993, and the BSE degree in electrical engineering and computer science from Duke University in 1991. He is an associate professor at the School of Electrical Engineering and Computer Science at the University of Central Florida and the founder of its Computer Systems Laboratory. His research interests include active memory and I/O subsystems, novel parallel computer architectures, data-intensive computing, scalable cache coherence protocols, multiprocessor design and simulation methodology, and hardware/software codesign. He is the recipient of the US National Science Foundation CAREER Award supporting novel research in data-intensive computing. He is a member of the IEEE and the IEEE Computer Society.