

The Impact of Negative Acknowledgments in Shared Memory Scientific Applications

Mainak Chaudhuri*

Computer Systems Laboratory, Cornell University, Ithaca, NY 14853.

`mainak@csl.cornell.edu`

Mark Heinrich

School of EECS, University of Central Florida, Orlando, FL 32816.

`heinrich@cs.ucf.edu`

*Corresponding author, e-mail: `mainak@csl.cornell.edu`

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% (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 [13, 14] we first improve it to get the benefits of the SGI Origin 2000 protocol [19] 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 [7] 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 micro-architectural 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 [25], 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 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, 20, 21] and industry [3, 7, 19] 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 [12, 13, 14, 28]. Cache-based protocols, such as

IEEE Scalable Coherent Interface [13, 14, 22, 23, 27], 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 [14].

The SGI Origin 2000 [19] 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 [7] and the Piranha chip-multiprocessor [3] introduce coherence protocols that do not generate NACKs. But the GS320 protocol and the intra-chip protocol of Piranha have to rely on point-to-point network ordering and total ordering properties among certain types of messages. The inter-node 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, 26]. 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 [11] 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.

Our study shows that for applications with heavily contended read-modify-write oper-

ations (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) [9], Queue-on-LockBit (QOLB) [17] and Implicit Queue-on-LockBit (IQOLB) [25] 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. Our node controller architecture shown in Figure 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,

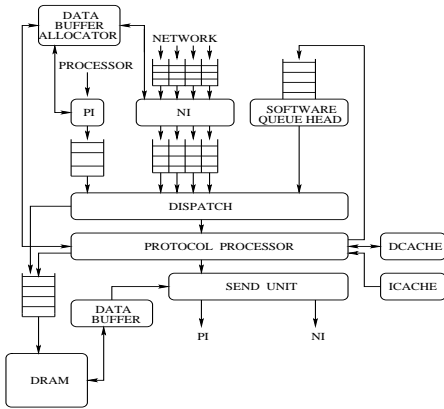


Figure 1. Node controller architecture

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 which 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 inter-node 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 write requests continuously gets forwarded to the previous owners obviating the need for ownership transfer messages. In Figure 2(a) three nodes, namely, W1,

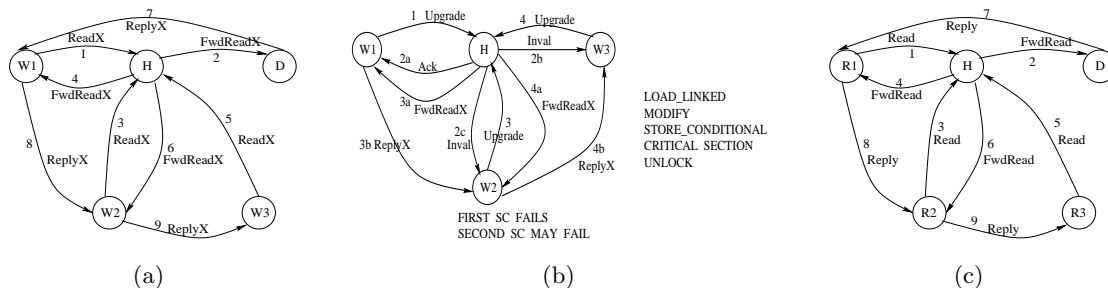


Figure 2. (a) Write string forwarding, (b) Effects of write string forwarding on critical sections, (c) Dirty sharing

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 Figure 2(b). 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 Figure 2(b) 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 micro-architectural 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 [25]. 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 [25] 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 Figure 2(b) 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** [25] 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 Figure 2(c). 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 17 system cycles per reply.

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 5KB 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 match any arbitrary performance requirements.

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 Figure 3. The read request

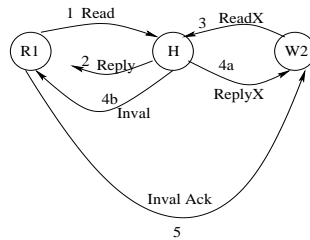


Figure 3. Read-invalidate race

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 [20, 21]. 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.

3.4 Summary of Protocols

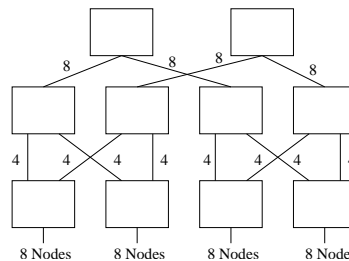
Table 1 summarizes the six protocols that we evaluate. Since the `OriginMod+DSH+WSF(+OPT)` protocol may return a cache line to the reader in the owned state, the conventional read-modify-writes implemented with LL/SC may livelock. Therefore, for the applications with read-modify-writes we turn on our 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 a O state. All the protocols other than `BaseBV` collect invalidation acknowledgments at the writer and require writeback acknowledgments.

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

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

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

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 [29]. 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 multi-grid phase has been changed from a lock-test-set-unlock sequence to a more efficient test-lock-test-set-unlock sequence [16].

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 4KB 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, 8].

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 [8]. 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 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 150ns. The fastest configuration, named **FT50ns**, is the same as **FT150ns** but has a hop time of 50ns. 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. Figure 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 [6]. Although it has a 50ns 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 [10] 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% parallel efficiency) on 128 nodes with the OriginMod protocol. Ocean and FFT are the only two applications showing efficiency close to 50%. 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.

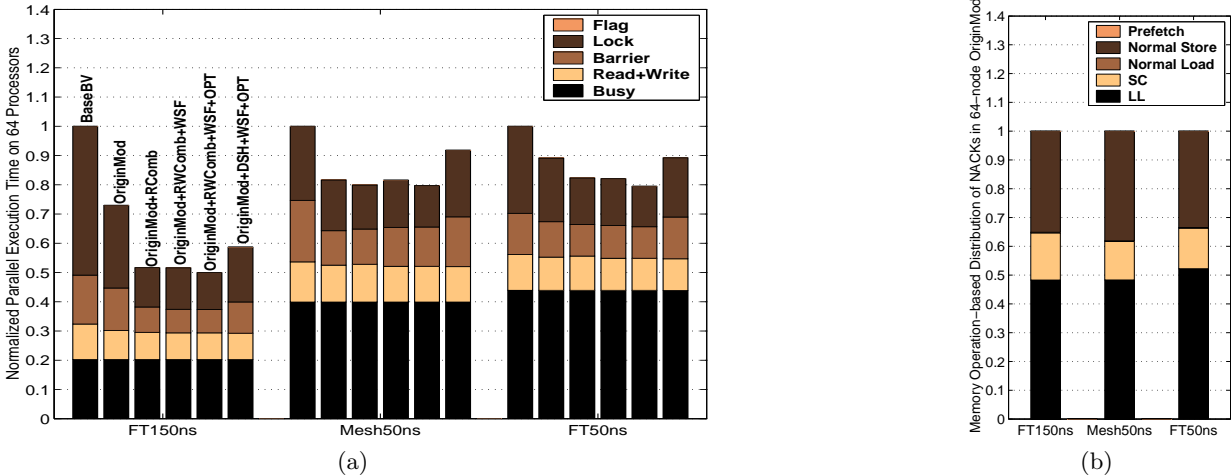


Figure 5. (a) Normalized execution time, (b) Distribution of NACKs for Water

Figure 5(a) presents the execution time normalized to BaseBV for three network config-

urations while Figure 5(b) 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% faster than `BaseBV` (this corresponds to a speedup of 1.37; the percent reduction in execution time can be read from the graph) while the `OriginMod+RComb` protocol is 93.4% faster than `BaseBV` and 41.2% 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% of the NACKs that are from LL instructions and 16.4% coming from SC instructions as shown in Figure 5(b), 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% of NACKs. Adding write string forwarding to `OriginMod+RComb` does not affect performance while the delayed intervention optimization executes 3.3% faster. The `OriginMod+DSH+WSF+OPT` protocol is 13.6% worse compared to `OriginMod+RComb` (this corresponds to 13.6% increased execution time) due to an 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% faster than `BaseBV` and 2.1% faster than `OriginMod`. For the `FT50ns` configuration this protocol runs 21.4% faster than `BaseBV` and 8.2% 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+RComb+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.

Figure 6(a) 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 the 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

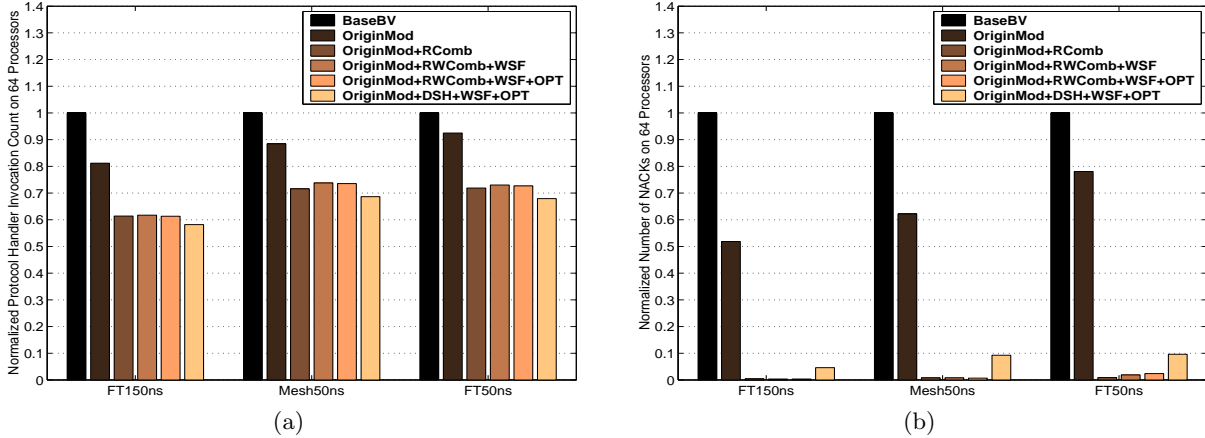


Figure 6. (a) Normalized protocol handler invocation count, (b) Normalized NACK count for Water

example, in the FT150ns configuration this protocol has 29.2% more three-hop read misses than the OriginMod+RComb protocol. Figure 6(b) shows the normalized count of NACKs. OriginMod achieves a substantial reduction in the NACKs over BaseBV: 48.1% for FT150ns, 37.8% for Mesh50ns and 21.9% 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 oct-tree. Both versions use page placement and software tree barriers, but no software prefetch.

5.2.1 Barnes-Hut with 64K Locks

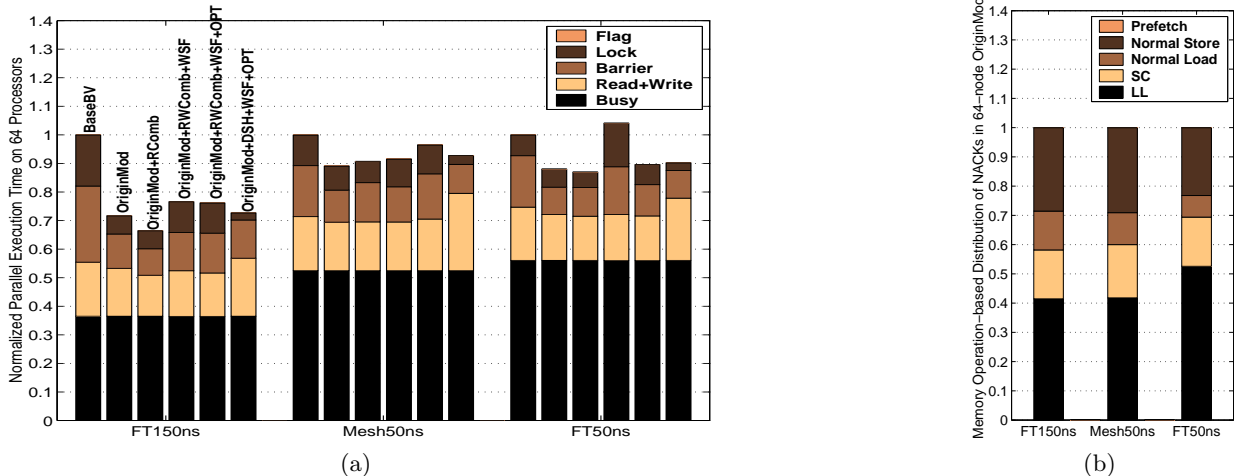


Figure 7. (a) Normalized execution time, (b) Distribution of NACKs for Barnes Hut with 64K locks

Figure 7(a) presents the execution time normalized to BaseBV for three network configurations while Figure 7(b) presents the distribution of NACKs in the OriginMod protocol. For the FT150ns configuration the OriginMod protocol executes 39.5% faster than BaseBV while

the `OriginMod+RComb` protocol is 50.6% faster than `BaseBV` and 7.9% 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 Figure 7(b), 41.4% of the NACKs arise from LL instructions and 16.7% from SC instructions. While the elimination of these NACKs accelerates lock acquires, the remaining 43% of the NACKs play the most significant role. Adding write string forwarding to `OriginMod+RComb` increases the execution time significantly and `OriginMod+RWComb+WSF` actually performs 6.9% worse compared to `OriginMod` due to an increase in 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% faster than `BaseBV`. All other protocols perform worse compared to `OriginMod` with `OriginMod+RComb` being the closest yielding 1.8% increased execution time over `OriginMod`. For the `FT50ns` configuration the `OriginMod+RComb` protocol is 14.9% faster than `BaseBV` and 1.2% faster than `OriginMod`. The `OriginMod+RWComb+WSF` protocol suffers greatly from an increased count of failed store-conditionals and performs 4.1% worse compared to even `BaseBV`. The delayed intervention optimization effectively eliminates many failed store-conditionals and helps bring down the execution time, but still performs 1.8% worse compared to `OriginMod`. 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.

Figure 8(a) shows the normalized count of negative acknowledgments. Figure 8(b) 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

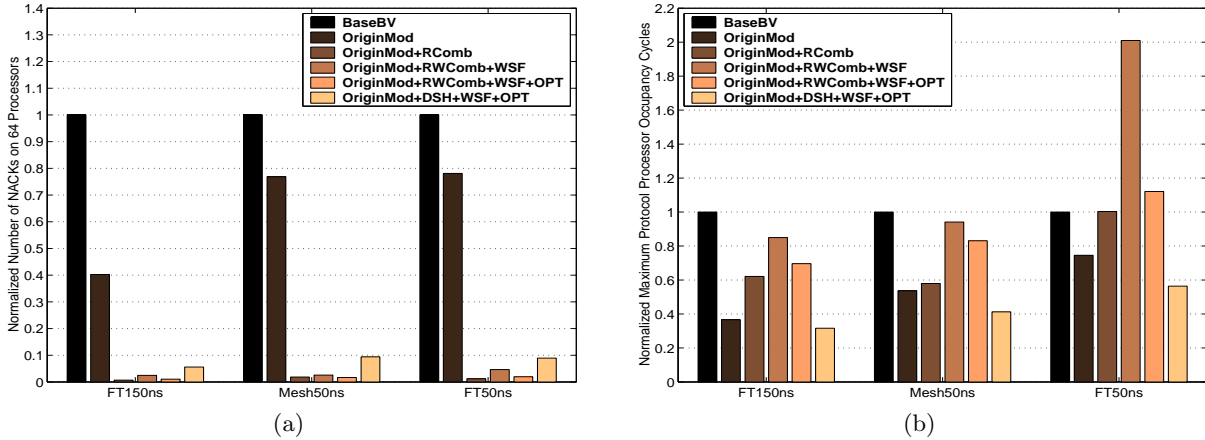


Figure 8. (a) Normalized NACK count, (b) Normalized protocol processor occupancy cycles for Barnes Hut with 64K locks

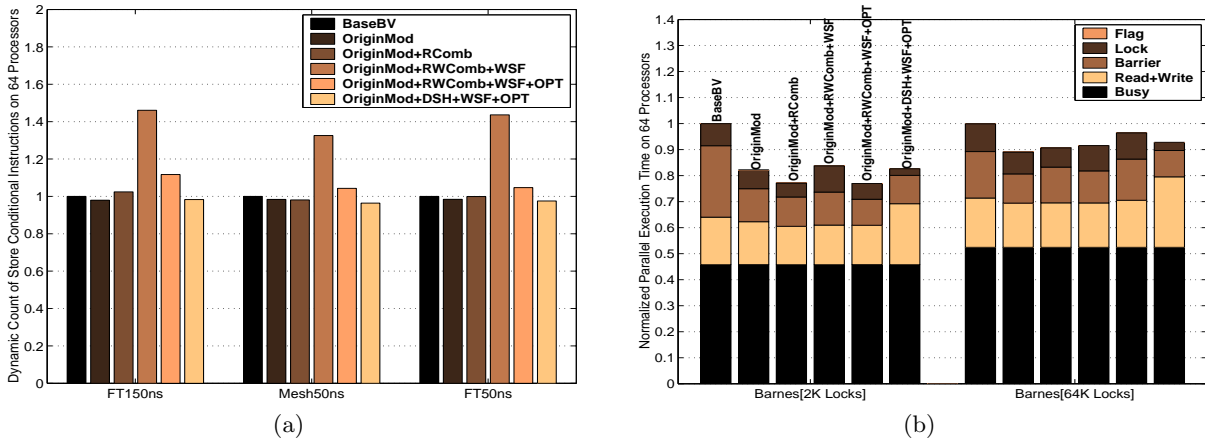


Figure 9. (a) Normalized dynamic count of SC instructions for Barnes Hut with 64K locks, (b) Normalized execution time for Barnes Hut with 2048 locks on Mesh50ns

the large-scale producer-consumer sharing patterns continue to result in poor performance with this protocol. Finally, Figure 9(a) 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 2048 Locks

Since the majority of NACKs arise from lock acquires, we experimented with a smaller number of array locks. With 2048 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 Figure 9(b). For comparison we also present the results with 64K locks alongside. Clearly, with 2048 locks all the protocols show greater performance improvement relative to `BaseBV` as compared to that with 64K locks. The `OriginMod+RComb` protocol is 29.5% faster than `BaseBV` and 6.4% faster than `OriginMod`. A comparison between the performance of `OriginMod+RComb` with 64K and 2048 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 [13, 14]. 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.

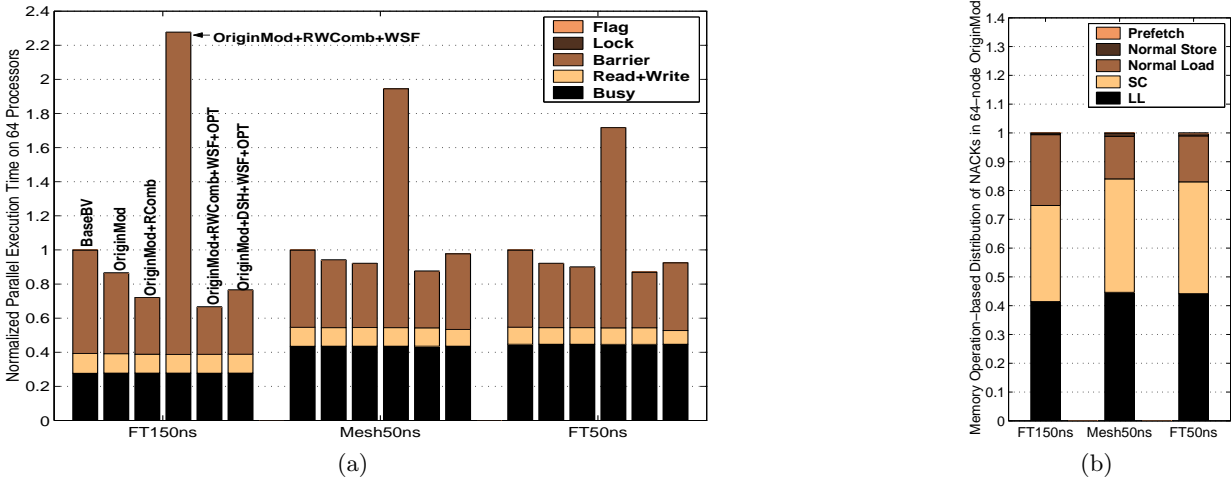


Figure 10. (a) Normalized execution time, (b) Distribution of NACKs for unoptimized LU

Figure 10(a) presents the execution time normalized to `BaseBV` for three network configurations while Figure 10(b) 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% in `FT150ns`, 84% in `Mesh50ns` and 83% 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% faster than `BaseBV` while the `OriginMod+RComb` protocol is 38.7% faster than `BaseBV` and 19.8% 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 8.5% 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% 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`.

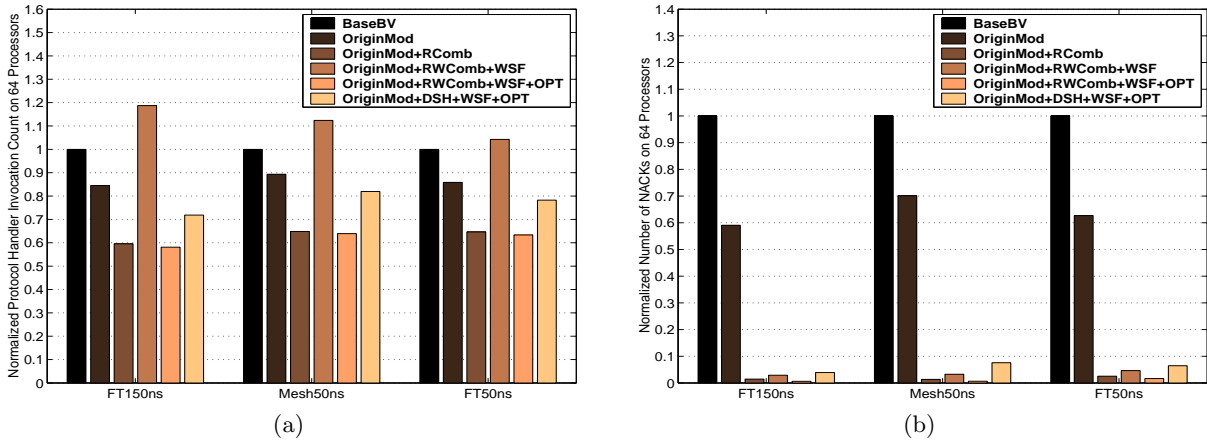


Figure 11. (a) Normalized protocol handler invocation count, (b) Normalized NACK count for unoptimized LU

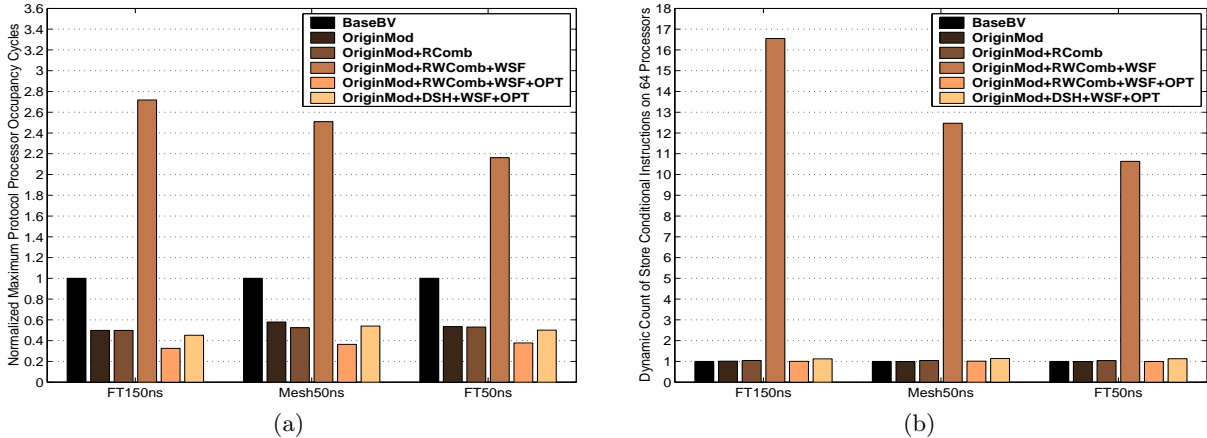


Figure 12. (a) Normalized protocol processor occupancy cycles, (b) Normalized dynamic count of SC instructions for unoptimized LU

Figure 11(a) 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. Figure 11(b) shows the normalized count of NACKs. Figure 12(a) 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, Figure 12(b) 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.

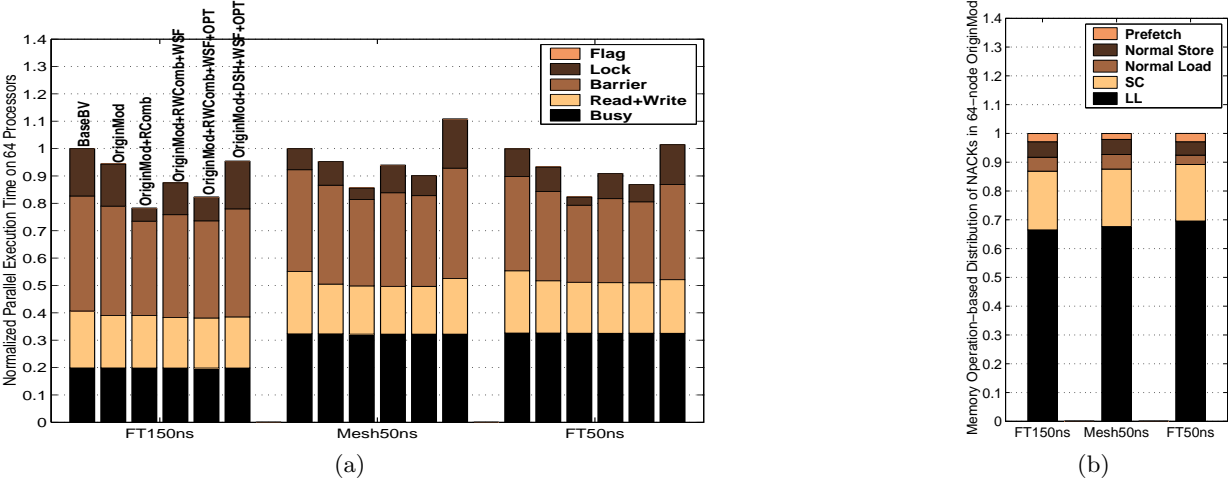


Figure 13. (a) Normalized execution time, (b) Distribution of NACKs for Ocean

Figure 13(a) presents the execution time normalized to `BaseBV` for three network configurations while Figure 13(b) presents the distribution of NACKs in the `OriginMod` protocol. From Figure 13(b) we note that for `FT150ns` 86.9% of the NACKs result from LL and SC instructions while for `Mesh50ns` and `FT50ns` the percentages are 87.6% and 89.1%, respectively. This is due to heavily contended lock acquires in Ocean.

From Figure 13(a) 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% faster than BaseBV and 20.6% 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% 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% faster than BaseBV and 11.4% 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% faster than BaseBV and 13.4% 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.

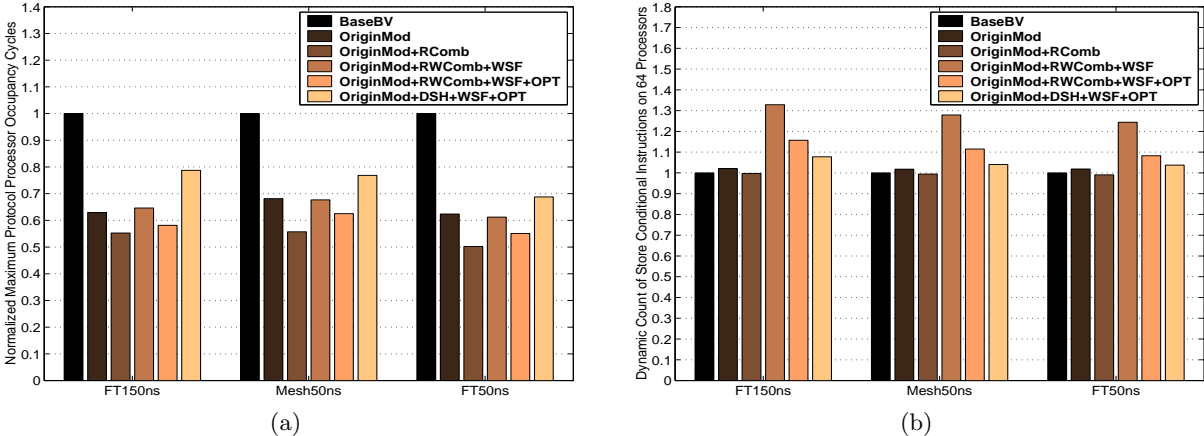


Figure 14. (a) Normalized protocol processor occupancy cycles, (b) Normalized dynamic count of SC instructions for Ocean

Figure 14(a) 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, Figure 14(b) 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

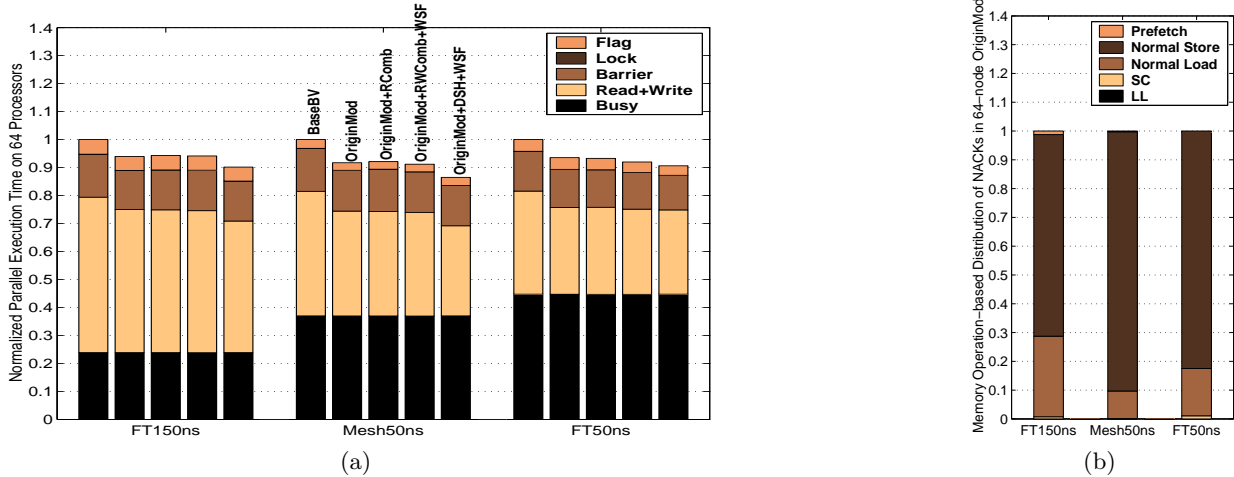


Figure 15. (a) Normalized execution time, (b) Distribution of NACKs for Radix-Sort

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 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.

Figure 15(a) presents the execution time normalized to `BaseBV` for the three network configurations for Radix-Sort, while Figure 15(b) 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 as the best for all three network configurations. For the `FT150ns` configuration this protocol executes 11% and 4.2% 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% 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 Figure 15(b). 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.

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

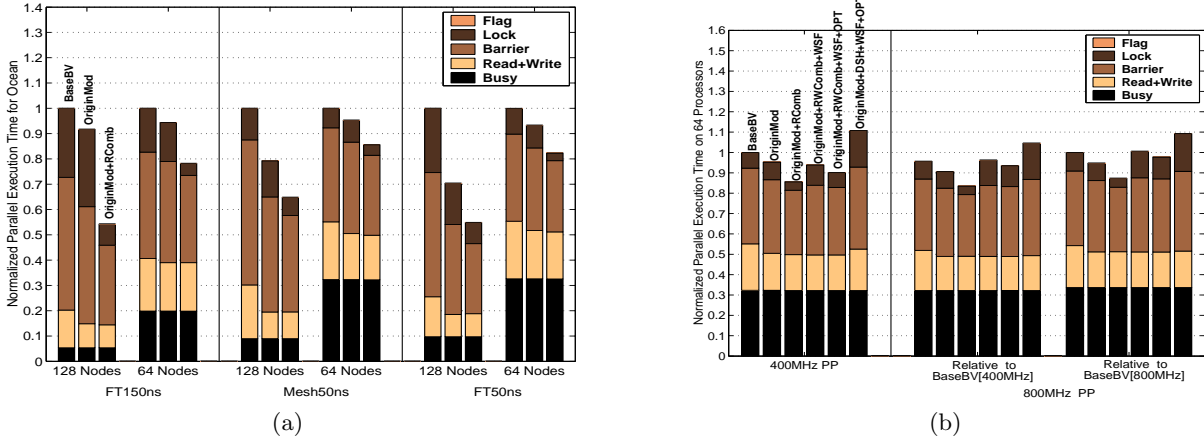
In Table 3, other than four cases, protocols with some form of request combining emerge as the best, and the `OriginMod+RComb` protocol is the best in 12 out of 18 cases. So, in Table 4 we summarize the speedup achieved by `OriginMod+RComb`, the best request combining protocol, with respect to `BaseBV` 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% to 93% relative to `BaseBV` and up to 41% relative to `OriginMod` across various network configurations. Other than a few cases of negligible slowdown (at most 2%), 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 Figure 16(a) 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

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

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

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% faster than OriginMod while on 128 nodes it executes 68.9% 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%, 2.0%, and 1.7% faster than OriginMod on FT150ns, FT50ns, and Mesh50ns configurations, respectively. For Ocean the numbers are 3.8%, 2.6%, and 2.6% 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 [15, 18], STiNG multiprocessor [22], S3.mp [24] 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 Figure 16(b) 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 micro-architectural 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.

References

- [1] D. Abts, D. J. Lilja, and S. Scott. Towards Complexity-Effective Verification: A Case Study of the Cray SV2 Cache Coherence Protocol. In *Proc. Workshop on Complexity-Effective Design*, held with the 27th Int'l Symp. Comp. Arch. (ISCA), June 2000.
- [2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proc. 25th Int'l Symp. Comp. Arch. (ISCA)*, pp. 3–14, June/July 1998.
- [3] L. A. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. 27th Int'l Symp. Comp. Arch. (ISCA)*, pp. 282–293, June 2000.
- [4] D. Chaiken, J. Kubiawicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. 4th ASPLOS*, pp. 224–234, April 1991.
- [5] M. Chaudhuri et al. “Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation”. *IEEE Trans. Comp.*, **52**(7):862–880, July 2003.
- [6] M. Galles. “Spider: A High-Speed Network Interconnect”. *IEEE Micro*, **17**(1):34–39, January-February 1997.
- [7] K. Gharachorloo et al. Architecture and design of AlphaServer GS320. In *Proc. 9th ASPLOS*, pp. 13–24, November 2000.
- [8] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proc. 9th ASPLOS*, pp. 49–58, November 2000.
- [9] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives

- for Large-Scale Cache-Coherent Multiprocessors. In *Proc. 3rd ASPLOS*, pp. 64–75, May 1989.
- [10] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. 1990 Int'l Conf. Par. Proc. (ICPP)*, pp. I.312–I.321, August 1990.
- [11] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. 5th Int'l Symp. High Perf. Comp. Arch. (HPCA)*, pp. 172–181, January 1999.
- [12] E. Hagersten, A. Landin, and S. Haridi. “DDM—A Cache-Only Memory Architecture”. In *IEEE Computer*, pp. 44–54, September 1992.
- [13] M. Heinrich. “The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols”. Ph.D. Dissertation, Stanford University, October 1998.
- [14] M. Heinrich et al. “A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols”. In *IEEE Trans. Comp.*, **48**(2):205–217, (Special Issue on Cache Memory and Related Problems), February 1999.
- [15] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. 6th ASPLOS*, pp. 274–285, October 1994.
- [16] M. Heinrich and M. Chaudhuri. “Ocean Warning: Avoid Drowning”. To appear in *ACM SIGARCH Computer Architecture News*, **31**(3), June 2003.
- [17] A. Kägi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proc. 24th Int'l Symp. Comp. Arch. (ISCA)*, pp. 170–180, June 1997.
- [18] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. 21st Int'l Symp. Comp. Arch. (ISCA)*, pp. 302–313, April 1994.
- [19] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proc. 24th Int'l Symp. Comp. Arch. (ISCA)*, pp. 241–251, June 1997.
- [20] D. Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. 17th Int'l Symp. Comp. Arch. (ISCA)*, pp. 148–159, May 1990.
- [21] D. Lenoski et al. The Stanford DASH Multiprocessor. In *IEEE Computer*, **25**(3):63–79, March 1992.
- [22] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. 23rd Int'l Symp. Comp. Arch. (ISCA)*, pp. 308–317, May 1996.
- [23] T. D. Lovett, R. M. Clapp, and R. J. Safranek. “NUMA-Q: An SCI-based Enterprise Server”. Sequent Computer Systems Inc., 1996.
- [24] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. 24th Int'l Conf. Par. Proc. (ICPP)*, pp. I1–I10, August 1995.
- [25] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proc. 6th Int'l Symp. High Perf. Comp. Arch. (HPCA)*, pp. 168–179, January 2000.
- [26] P. Ranganathan et al. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proc. 10th ASPLOS*, pp. 307–318, October, 1998.
- [27] Scalable Coherent Interface, ANSI/IEEE Standard 1596–1992, August 1993.
- [28] R. Simoni. “Cache Coherence Directories for Scalable Multiprocessors”. Ph.D. Dissertation, Stanford University, October 1992.
- [29] S.C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Int'l Symp. Comp. Arch. (ISCA)*, pp. 24–36, June 1995.

A Protocol Specifications

In this appendix we present the details of the coherence protocols including the structure of the directory entry, the message types and the protocol state machine.

A.1 BaseBV

In the following we present the details of the baseline bitvector protocol (**BaseBV**).

A.1.1 Directory Entry

P	D	UNUSED	INVAL	ACK COUNT	SHARER VECTOR
1 bit	1 bit	4 bits	1 bit	9 bits	48 bits

P = PENDING. D = DIRTY

Figure 17. Directory entry of BaseBV

Figure 17 shows the directory entry of the **BaseBV** protocol. The “ACK COUNT” field stores the number of invalidation acknowledgments to receive on a write. The “INVAL” field is set if the home node failed to send out all the invalidations due to shortage of space in the outgoing forwarded request queue (third virtual lane) in NI.

A.1.2 Message Types

Table 5 shows the messages from the processor to the memory controller. Table 6 shows the messages from the memory controller to the processor. Table 7 shows the network messages.

A.1.3 Protocol State Machine

Table 8 presents the protocol state machine for messages from the processor with local addresses. This state machine runs at the home node. Note that, if necessary, a memory data read (in cases of read and read-exclusive requests) or a memory data write (in case of a writeback) is speculatively initiated by the memory controller before a protocol handler starts executing. Table 9 presents the protocol state machine for messages from the processor with remote addresses. Note that, the directory state is not available in these cases. Tables 10 and 11 present the protocol state machine for messages from the network with local addresses. This state machine runs at the home node. Table 12 presents the protocol state machine for messages from the network with remote addresses. In addition to these state machines the coherence protocol has one more state machine to handle software queue messages. These

messages cause the home node to send out remaining invalidations. If all the invalidations are sent out successfully, the INVALID bit is reset and the owner is marked in the directory; otherwise the message is rescheduled on the software queue for future dispatch. The latter case arises when the home node again runs out of NI outgoing queue space while sending out the remaining invalidations.

Finally, we would like to mention that for coarseness greater than 1 (i.e. for more than 48 processors) we turn off upgrades. This is necessary to avoid upgrade-invalidate races, since this protocol is not equipped to handle it. We resolve this issue in the `OriginMod` protocol by making the memory controller capable of re-issuing an invalidated upgrade.

A.2 OriginMod

In the following we present the details of the modified SGI Origin 2000 protocol (`OriginMod`).

A.2.1 Directory Entry

PSH	PDEX	D	L	UNUSED	SHARER VECTOR
1 bit	1 bit	1 bit	1 bit	28 bits	32 bits

PSH=PENDING SHARED. PDEX=PENDING DIRTY EXCLUSIVE. D=DIRTY. L=LOCAL

Figure 18. Directory entry of OriginMod

Figure 18 shows the directory entry of the `OriginMod` protocol.

A.2.2 Message Types

Table 13 shows the messages from the processor to the memory controller. The first four messages are the same as in `BaseBV`. Table 14 shows the messages from the memory controller to the processor. The first seven messages are the same as in `BaseBV`. Table 15 shows the network messages. Although most of the network messages are the same as in `BaseBV`, some of the encodings are different. Also there are a few new messages.

A.2.3 Protocol State Machine

Tables 16 and 17 present the protocol state machine for messages from the processor with local addresses. This state machine runs at the home node. Table 18 presents the protocol state machine for messages from the processor with remote addresses. Tables 19, 20 and 21 present the protocol state machine for messages from the network with local addresses. This

state machine runs at the home node. Tables 22 and 23 present the protocol state machine for messages from the network with remote addresses. In addition to these state machines the coherence protocol has one more state machine to handle software queue messages. These messages cause the home node to send out remaining invalidations. If the home node again runs out of NI outgoing queue space while sending out the remaining invalidations, the message is rescheduled on the software queue for future dispatch.

A.3 OriginMod+RComb

In the following we present the details of our first NACK-free protocol, OriginMod+RComb.

A.3.1 Directory Entry

PSH	PDEX	D	L	SCH	PRV	PWV	PRID	PWID	R0	ROV	OFF
1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	7 bits	7 bits	7 bits	1 bit	3 bits

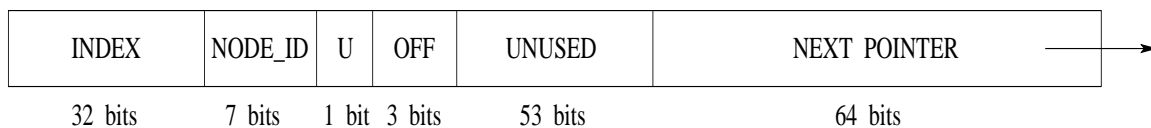
PSH=PENDING SHARED, PDEX=PENDING DIRTY EXCLUSIVE, D=DIRTY, L=LOCAL, SCH=SCHEDULED, PRV=PENDING READ LIST VALID, PWV=PENDING WRITE LIST VALID, PRID=INDEX OF THE FIRST ENTRY IN PENDING READ LIST, PWID=INDEX OF THE FIRST ENTRY IN PENDING WRITE LIST, R0=FIRST PENDING READER, ROV=FIRST PENDING READER VALID, OFF=QUAD-WORD OFFSET OF THE FIRST PENDING READ ADDRESS

Figure 19. Upper 32 bits of the directory entry of OriginMod+RComb

Figure 19 shows the upper 32 bits of the directory entry of the OriginMod+RComb protocol. The lower 32 bits form the sharer vector. Our protocol currently supports upto 128 processors, and so seven bits are sufficient to store a requesting node number (this dictates the width of the “R0” field). Since the current implementation supports pending lists of length 128 each, seven bits are sufficient to store a pending list entry index (this dictates the widths of the fields “PRID” and “PWID”). Section 3.2.2 discusses the use of all the states other than “OFF”. This field stores the quad-word offset of the address requested by R0 within a cache line. This is needed to carry out a critical double-word refill in the cache controller. Further, the memory controller sends the cache line in a subblock ordered fashion to satisfy the refill requirements of the L2 cache interface.

A.3.2 Pending List Entry

Figure 20 shows one pending list entry. The size of a pending list entry is 20 bytes. It is divided into three parts. An index is assigned to every list entry when the pending list



U=UPGRADE, OFF=OUAD-WORD OFFSET

Figure 20. Pending list entry

storage (5KB per node) is reserved at boot time. This index is 4 bytes long (only 7 bits are used) and does not change once it is assigned. The next pointer is 8 bytes long. The remaining 8 bytes are organized into a structure that holds the requesting node number (7 bits), an upgrade bit (relevant to the write pending list only), and the quad-word offset of the requested address within a cache line (3 bits). 53 bits are left unused in this structure, but they help the protocol processor to load the 64 bits containing the requesting node number, upgrade bit and the quad-word offset via a single load instruction.

A.3.3 Message Types

Message types are exactly the same as in `OriginMod`. No new messages were added. `MSG_NAK` is still kept in the system in case this is needed to avoid deadlock in presence of resource shortage. However, this situation never arises in our experiments.

A.3.4 Protocol State Machine

In this section we present the protocol state machine for `OriginMod+RComb`. Instead of presenting the full machine description, we will only discuss the changes that were made to the `OriginMod` state machine.

Table 24 shows the changes in the `OriginMod` state machine for handling messages from the processor with local addresses. Also, `PLPROC_PUTX_REQ`, `PLDP_GET_REQ`, `PLDP_GETX_REQ` and `PLPROC_GETWB_REQ` are the messages that clear the pending states in the directory. In these cases depending on the pending list states the protocol gives priority to the pending reads and sends out the replies to either the first two pending reads or the first pending write (if there is no pending read). If the pending lists are not empty at this point and the `Scheduled` bit in the directory is reset, a message is scheduled on the software queue for future dispatch. At this time the `Scheduled` bit in the directory is also set. This software queue message handler, when dispatched, walks through the pending lists and sends out the replies as described in Section 3.2.1. The `Scheduled` bit gets reset only when the software queue handler retires either because the pending lists are empty or because the directory entry transitions to a pending state.

In case of handling network messages with local addresses, the state machine behaves similarly as in Table 24 for message types MSG_GET, MSG_GETX and MSG_UPGRADE if the directory is in the pending state. The network message types that clear the pending states of the directory (as can be seen from Tables 19, 20 and 21) include MSG_PUT, MSG_WB, MSG_SWB, MSG_RWB, MSG_FORWARD_ACK, MSG_PUTX, and MSG_PUT_WB. The state machine in case of these messages behaves similarly as described in the previous paragraph.

Finally, we take note of a problem specific to supporting subblock ordered cache line delivery. While sending out pending replies, the cache lines are sent starting at the aligned address instead of the requested address. However, the message type used is MSG_PUT_FORWARD for pending read replies or MSG_PUTX_FORWARD for pending write replies. This makes sure that the processor interface at the requesting node recognizes these replies as cache line aligned and re-orders the data correctly by reading the requested address from the OTT. At this time the message type is also changed to PLDP_PUT_RPLY or PLDP_PUTX_RPLY, which are recognized by the L2 cache controller.

A.4 OriginMod+RWComb+WSF

In the following we present the details of our second NACK-free protocol, **OriginMod+RWComb+WSF**.

A.4.1 Directory Entry

The directory entry is exactly the same as in **OriginMod+RComb**, except that this protocol does not have a PDEX state. So, this bit in the directory entry is left unused.

A.4.2 Message Types

The message types are the same as in **OriginMod**, except that this protocol does not have MSG_FORWARD_ACK.

A.4.3 Protocol State Machine

The state machine is very similar to that in the **OriginMod+RComb** protocol. The differences specific to messages from the processor are mentioned below.

- The forwarded read-exclusive interventions (MSG_GETX) does not mark the directory entry pending. This affects the state machines of PLPROC_GETX_REQ and PLPROC_UPGRADE_REQ at the home node if the directory state is Dirty.
- PLPROC_PUTX_REQ at the home node does not generate any late intervention replies because these are not necessary any more. All late interventions are satisfied at the third

party nodes from the writeback buffer, which now holds the written back cache line in addition to the address until it is acknowledged by the home node. Also, a cache line is written back to home memory only if the directory state is dirty and the source of the writeback request matches the owner in the directory. However, PLPROC_PUTX_REQ always turns off the Local bit in the directory entry and acknowledges the writeback in the local writeback buffer.

- PLDP_GETX_REQ does not modify the directory state at all.

The differences specific to network messages are mentioned below.

- The forwarded read-exclusive interventions (MSG_GETX) do not mark the directory entry pending. This affects the state machines of MSG_GETX and MSG_UPGRADE at the home node if the directory state is Dirty.
- MSG_PUTX arriving at the home node does not modify the directory state at all.
- MSG_WB at the home node does not generate any late intervention replies because these are not necessary any more. All late interventions are satisfied at the third party nodes from the writeback buffer, which now holds the written back cache line in addition to the address until it is acknowledged by the home node. Also, a cache line is written back to home memory only if the directory state is dirty and the source of the writeback request matches the owner in the directory. In this case a MSG_WB_ACK is sent to the source node. In all other cases the cache line is not written back to memory and a MSG_WB_ACK_INT is sent to the source node.
- The part of the state machine for handling MSG_FORWARD_ACK is no longer necessary.

Finally, the software queue handler that handles the pending requests does not mark the directory entry pending after sending a read-exclusive intervention (MSG_GETX) while servicing the pending writes.

A.5 OriginMod+RWComb+WSF+OPT

This improved version of the previous protocol does not require any change at all in the protocol. It only requires additional micro-architectural supports in the primary cache controller of the processor as detailed in the paper.

A.6 OriginMod+DSH+WSF+OPT

In the following we present the details of the OriginMod+DSH+WSF+OPT protocol. This protocol is very similar to the Piranha protocol as far as elimination of NACKs is concerned. Again, we would like to mention that the optimization hardware aimed at accelerating LL/SC sequences does not require any changes to the protocol. So, in the following we will essentially present a formal description of the protocol OriginMod+DSH+WSF.

A.6.1 Directory Entry

D	DSH	L	UNUSED	OWNER	SHARER VECTOR
1 bit	1 bit	1 bit	21 bits	8 bits	32 bits

D=DIRTY. DSH=DIRTY SHARED. L=LOCAL

Figure 21. Directory entry of OriginMod+DSH+WSF

Figure 21 shows the directory entry of the OriginMod+DSH+WSF protocol supporting upto 256 processors (this dictates the width of the “OWNER” field).

A.6.2 Message Types

Three new message types are introduced from the memory controller to the processor. Table 25 describes them. Also, two messages are removed from the memory controller to the processor. These are PLDP_PUT_FORWARD and PLDP_PUTX_FORWARD.

Three new network messages are introduced. Table 26 describes them. Also, six messages are removed. These are MSG_PUT_FORWARD, MSG_SWB, MSG_RWB, MSG_FORWARD_ACK, MSG_PUTX_FORWARD, and MSG_PUT_WB. MSG_NAK is still kept in the system in case this is needed to avoid deadlock in presence of resource shortage. However, this situation never arises in our experiments.

A.6.3 Protocol State Machine

Tables 27 and 28 present the protocol state machine for messages from the processor with local addresses. This state machine runs at the home node. Table 29 presents the protocol state machine for messages from the processor with remote addresses. Tables 30 to 34 present the protocol state machine for messages from the network with local addresses. This state machine runs at the home node. Tables 35 and 36 present the protocol state machine for messages from the network with remote addresses. In addition to these state machines the coherence protocol

has one more state machine to handle software queue messages. These messages cause the home node to send out remaining invalidations. If the home node again runs out of NI outgoing queue space while sending out the remaining invalidations, the message is rescheduled on the software queue for future dispatch. Readers should note the subtle differences between this protocol and the `OriginMod` protocol.

Table 5. Processor to Memory Controller Messages in BaseBV

Message type (encoding)	Description
PLPROC_GET_REQ (0xd)	Read request
PLPROC_GETX_REQ (0xf)	Read-exclusive request
PLPROC_PUTX_REQ (0x5)	Writeback request
PLPROC_UPGRADE_REQ (0x8)	Upgrade request

Table 6. Memory Controller to Processor Messages in BaseBV

Message type (encoding)	Description
PI_DP_PUT_RPLY (0xd)	Read reply
PI_DP_PUTX_RPLY (0xf)	Read-exclusive reply
PI_DP_ACK_RPLY (0x8)	Upgrade acknowledgment
PI_DP_NAK_RPLY (0xa)	Negative acknowledgment
PI_DP_INVALID_REQ (0x4)	External invalidation request
PI_DP_GET_REQ (0x6)	External read intervention
PI_DP_GETX_REQ (0x7)	External read-exclusive intervention

Table 7. Network Messages in BaseBV

Message type (encoding)	Description
MSG_NAK (0x0a)	Negative acknowledgment to requester
MSG_NAK_CLEAR (0x22)	Pending clear request to home
MSG_GET (0x06)	Read request to home or third party node
MSG_GETX (0x47)	Read-exclusive request to home or third party node
MSG_PUT (0x0d)	Read reply to requester
MSG_PUTX_ACKS_DONE (0x0f)	Read-exclusive reply to requester with global completion
MSG_INVALID_ACK (0x34)	Invalidation acknowledgment from sharer to home
MSG_WB (0x00)	Writeback request to home
MSG_FORWARD_ACK (0x05)	Ownership transfer from previous owner to home
MSG_SWB (0x14)	Sharing writeback to home
MSG_UPGRADE (0x43)	Upgrade request to home
MSG_INVALID (0x34)	Invalidation from home to sharer
MSG_UPGRADE_ACK (0x48)	Upgrade acknowledgment to requester without global completion
MSG_PUTX (0x2f)	Read-exclusive reply to requester without global completion
MSG_ACKS_DONE (0x11)	Global completion of write request

Table 8. Handling Messages from the Processor for Local Addresses in BaseBV

Message type	Directory state	Action
PLPROC_GET_REQ	Sharers	Send PLDP_PUT_RPLY; mark sharer vector.
PLPROC_GET_REQ	Dirty	Send MSG_GET to owner; mark directory Pending.
PLPROC_GET_REQ	Pending	Send PLDP_NAK_RPLY.
PLPROC_GETX_REQ	Clean, no sharer	Send PLDP_PUTX_RPLY with completion; mark owner in directory; Dirty=1.
PLPROC_GETX_REQ	Sharers	Send PLDP_PUTX_RPLY without completion; mark Dirty=1 in directory; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more inval left, mark INVALID in directory and go to software queue; mark directory Pending.
PLPROC_GETX_REQ	Dirty	Send MSG_GETX to owner; mark directory Pending.
PLPROC_GETX_REQ	Pending	Send PLDP_NAK_RPLY.
PLPROC_PUTX_REQ	Dirty	Mark sharer vector=0, Dirty=0.
PLPROC_PUTX_REQ	Pending, Dirty	Mark Dirty=0.
PLPROC_UPGRADE_REQ	Requester is only sharer	Send PLDP_ACK_RPLY with completion; mark owner in directory; Dirty=1.
PLPROC_UPGRADE_REQ	Clean, no sharer	Read memory; send PLDP_PUTX_RPLY with completion; mark owner in directory; Dirty=1.
PLPROC_UPGRADE_REQ	Sharers with requester	Send PLDP_ACK_RPLY without completion; mark Dirty=1 in directory; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more inval left, mark INVALID in directory and go to software queue; mark directory Pending.
PLPROC_UPGRADE_REQ	Sharers without requester	Read memory; send PLDP_PUTX_RPLY without completion; mark Dirty=1 in directory; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more inval left, mark INVALID in directory and go to software queue; mark directory Pending.
PLPROC_UPGRADE_REQ	Dirty	Send MSG_GETX to owner; mark directory Pending.
PLPROC_UPGRADE_REQ	Pending	Send PLDP_NAK_RPLY.

Table 9. Handling Messages from the Processor for Remote Addresses in BaseBV

Message type	Action
PLPROC_GET_REQ	Send MSG_GET to home.
PLPROC_GETX_REQ	Send MSG_GETX to home.
PLPROC_PUTX_REQ	Send MSG_WB to home.
PLPROC_UPGRADE_REQ	Send MSG_UPGRADE to home.

Table 10. Handling Messages from the Network for Local Addresses in BaseBV

Message type	Directory state	Action
MSG_NAK	X	Send PLDP_NAK_RPLY to local processor.
MSG_NAK_CLEAR	Pending	Pending=0.
MSG_GET	Sharers	Send MSG_PUT to requester; mark sharer vector.
MSG_GET	Dirty	If local owner, send PLDP_GET_REQ to local processor and send either MSG_PUT (if L2 state reply is Dirty) or MSG_NAK to requester; mark sharer vector and Dirty=0 in former case. If remote owner, send MSG_GET to owner; mark directory Pending.
MSG_GET	Pending	Send MSG_NAK to requester.
MSG_GETX	Clean, no sharer	Send MSG_PUTX_ACKS_DONE to requester; mark owner in directory; Dirty=1.
MSG_GETX	Sharers	Send MSG_PUTX to requester; mark Dirty=1 in directory; send PLDP_INVALID_REQ to local sharer; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more inval left, mark INVALID in directory and go to software queue; mark directory Pending.
MSG_GETX	Dirty	If local owner, send PLDP_GETX_REQ to local processor and send either MSG_PUTX_ACKS_DONE (if L2 state reply is Dirty) or MSG_NAK to requester; mark owner and Dirty=1 in former case. If remote owner, send MSG_GETX to owner; mark directory Pending.
MSG_GETX	Pending	Send MSG_NAK to requester.
MSG_PUT	Pending, Dirty	Send PLDP_PUT_RPLY to local processor; mark sharers (two) in directory; mark Dirty=Pending=0.
MSG_PUTX_ACKS_DONE	Pending, Dirty	Send PLDP_PUTX_RPLY to local processor with completion; mark owner in directory; mark Pending=0.
MSG_INVALID_ACK	ACK COUNT=1, INVALID=0, Dirty	If local owner, send write completion; otherwise send MSG_ACKS_DONE to owner; mark Pending=ACK COUNT=0 in directory.
MSG_INVALID_ACK	ACK COUNT=1, INVALID=0, Dirty=0	If local owner, send write completion; otherwise send MSG_ACKS_DONE to owner; mark Pending=ACK COUNT=0 in directory; mark sharer vector=0.
MSG_INVALID_ACK	ACK COUNT≠1 or INVALID≠0	Decrement ACK COUNT.

Table 11. Handling Messages from the Network for Local Addresses in BaseBV (Cont.)

Message type	Directory state	Action
MSG_WB	Dirty	Dirty=vector=0.
MSG_WB	Dirty, Pending, ACK COUNT=0, INVAL=0	Dirty=vector=0.
MSG_WB	Dirty, Pending, ACK COUNT≠0 or INVAL≠0	Dirty=0
MSG_FORWARD_ACK	Pending, Dirty	Pending=0; mark owner.
MSG_FORWARD_ACK	Pending, Dirty=0	Pending=vector=0.
MSG_SWB	Pending, Dirty	Pending=Dirty=0; mark sharers (two).
MSG_UPGRADE	Requester is only sharer	Send MSG_UPGRADE_ACK and MSG_ACKS_DONE to requester; mark owner in directory; Dirty=1.
MSG_UPGRADE	Clean, no sharer	Read memory; send MSG_PUTX_ACKS_DONE to requester; mark owner in directory; Dirty=1.
MSG_UPGRADE	Sharers with requester	Send MSG_UPGRADE_ACK to requester; mark Dirty=1 in directory; send PLDP_INVALID_REQ to local sharer; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more invals left, mark INVAL in directory and go to software queue; mark directory Pending.
MSG_UPGRADE	Sharers without requester	Read memory; send MSG_PUTX to requester; mark Dirty=1 in directory; send PLDP_INVALID_REQ to local sharer; send invalidations (MSG_INVALID) until NI queue fills and increment ACK COUNT; if more invals left, mark INVAL in directory and go to software queue; mark directory Pending.
MSG_UPGRADE	Dirty	If local owner, send PLDP_GETX_REQ to local processor and send either MSG_PUTX_ACKS_DONE (if L2 state reply is Dirty) or MSG_NAK to requester; mark owner and Dirty=1 in former case. If remote owner, send MSG_GETX to owner; mark directory Pending.
MSG_UPGRADE	Pending	Send MSG_NAK to requester.

Table 12. Handling Messages from the Network for Remote Addresses in BaseBV

Message type	Action
MSG_NAK	Send PLDP_NAK_RPLY to local processor.
MSG_GET	Send PLDP_GET_REQ to local processor; If L2 state reply is Dirty, send MSG_PUT to requester; if requester is not home, send MSG_SWB to home. If L2 state reply is not Dirty, send MSG_NAK to requester; send MSG_NAK_CLEAR to home.
MSG_GETX	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX_ACKS_DONE to requester; if requester is not home, send MSG_FORWARD_ACK to home. If L2 state reply is not Dirty, send MSG_NAK to requester; send MSG_NAK_CLEAR to home.
MSG_PUT	Send PLDP_PUT_RPLY to local processor; if OTT entry is invalidated change reply to PLDP_NAK_RPLY.
MSG_PUTX_ACKS_DONE	Send PLDP_PUTX_RPLY to local processor with completion.
MSG_PUTX	Send PLDP_PUTX_RPLY to local processor without completion.
MSG_INVALID	Send MSG_INVALID_ACK to home; send PLDP_INVALID_REQ to local processor.
MSG_ACKS_DONE	Mark write completion.
MSG_UPGRADE_ACK	Send PLDP_ACK_RPLY to local processor without completion.

Table 13. Processor to Memory Controller Messages in OriginMod

Message type (encoding)	Description
PLPROC_GET_REQ (0xd)	Read request
PLPROC_GETX_REQ (0xf)	Read-exclusive request
PLPROC_PUTX_REQ (0x5)	Writeback request
PLPROC_UPGRADE_REQ (0x8)	Upgrade request
PLDP_GET_REQ (0x6)	Reply to blocked early read intervention
PLDP_GETX_REQ (0x7)	Reply to blocked early read-exclusive intervention
PLPROC_GETWB_REQ (0x0)	Combined writeback request and blocked read intervention reply
PLPROC_UP_RACE_REQ (0x2)	Re-issue for upgrade-invalidate race

Table 14. Memory Controller to Processor Messages in OriginMod

Message type (encoding)	Description
PI_DP_PUT_RPLY (0xd)	Read reply
PI_DP_PUTX_RPLY (0xf)	Read-exclusive reply
PI_DP_ACK_RPLY (0x8)	Upgrade acknowledgment
PI_DP_NAK_RPLY (0xa)	Negative acknowledgment
PI_DP_INVALID_REQ (0x4)	External invalidation request
PI_DP_GET_REQ (0x6)	External read intervention
PI_DP_GETX_REQ (0x7)	External read-exclusive intervention
PI_DP_PUT_FORWARD (0x1)	Forwarded read reply from home in case of a late intervention race
PI_DP_PUTX_FORWARD (0x2)	Forwarded read-exclusive reply from home in case of a late intervention race
PI_DP_INVALID_ACK (0xb)	Invalidation acknowledgment to OTT; this message does not go all the way to the processor

Table 15. Network Messages in OriginMod

Message type (encoding)	Description
MSG_GET (0x06)	Read request to home or third party node
MSG_GETX (0x07)	Read-exclusive request to home or third party node
MSG_PUT (0x0d)	Read reply to requester
MSG_PUT_FORWARD (0x01)	Forwarded read reply to requester from home
MSG_WB (0x00)	Writeback request to home
MSG_SWB (0x0b)	Sharing writeback to home
MSG_RWB (0x0c)	Combined sharing writeback and replacement hint to home
MSG_UPGRADE (0x03)	Upgrade request to home
MSG_FORWARD_ACK (0x05)	Ownership transfer from previous owner to home
MSG_INVALID_ACK (0x09)	Invalidation acknowledgment from sharer to writer
MSG_PUTX (0x0f)	Read-exclusive reply to requester
MSG_PUTX_FORWARD (0x02)	Forwarded read-exclusive reply to requester from home
MSG_UP_RACE (0x02)	Re-issue to home for upgrade-invalidate race
MSG_INVALID (0x04)	Invalidation from home to sharer
MSG_UPGRADE_ACK (0x08)	Upgrade acknowledgment to requester
MSG_NAK (0x0a)	Negative acknowledgment to requester
MSG_PUT_WB (0x10)	Read reply to home with replacement hint
MSG_WB_ACK (0x11)	Writeback acknowledgment with no intervention pending
MSG_WB_ACK_INT (0x14)	Writeback acknowledgment with intervention pending

Table 16. Handling Messages from the Processor for Local Addresses in OriginMod

Message type	Directory state	Action
PLPROC_GETX_REQ	Sharers (includes no sharer case)	Send PLDP_PUTX_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more invals left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, Local=1, and vector=requester in directory; if ack count is zero mark write completion; otherwise register ack count in OTT entry.
PLPROC_GETX_REQ	Dirty	Send MSG_GETX to owner; mark directory with PDEX=1, Local=1, and vector=requester.
PLPROC_GETX_REQ	PSH=1 or PDEX=1	Send PLDP_NAK_RPLY to processor.
PLPROC_UPGRADE_REQ	Sharers (without requester)	Read memory; send PLDP_PUTX_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more invals left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, Local=1, and vector=requester in directory; if ack count is zero mark write completion; otherwise register ack count in OTT entry.
PLPROC_UPGRADE_REQ	Sharers (with requester)	Send PLDP_ACK_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more invals left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, Local=1, and vector=requester in directory; if ack count is zero mark write completion; otherwise register ack count in OTT entry.
PLPROC_UPGRADE_REQ	Dirty	Send MSG_GETX to owner; mark directory with PDEX=1, Local=1, and vector=requester.
PLPROC_UPGRADE_REQ	PSH=1 or PDEX=1	Send PLDP_NAK_RPLY to processor.

Table 17. Handling Messages from the Processor for Local Addresses in OriginMod (Cont.)

Message type	Directory state	Action
PLPROC_GET_REQ	Sharers	Send PLDP_PUT_RPLY to processor; mark sharer in directory and set Local=1.
PLPROC_GET_REQ	Dirty	Send MSG_GET to owner; set PSH=1, Dirty=0, and vector=requester in directory.
PLPROC_GET_REQ	PSH=1 or PDEX=1	Send PLDP_NAK_RPLY to processor.
PLPROC_PUTX_REQ	PSH=PDEX=0	Mark acknowledgment in writeback buffer entry; zero out directory entry.
PLPROC_PUTX_REQ	PSH=1	Mark acknowledgment in writeback buffer entry; send MSG_PUT_FORWARD to processor marked in directory vector; set PSH=0 and mark sharer in directory.
PLPROC_PUTX_REQ	PDEX=1	Mark acknowledgment in writeback buffer entry; send MSG_PUTX_FORWARD to processor marked in the directory vector; set PDEX=0 in directory.
PLPROC_UP_RACE_REQ	X	Send PLDP_PUTX_RPLY to processor; ignore ack count in reply header; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to processor; block the intervention reply until ack count is zero.
PLDP_GET_REQ	PSH=1	Send MSG_PUT to requester; mark two sharers in directory and set Local=1, PSH=0.
PLDP_GETX_REQ	PDEX=Dirty=1	Send MSG_PUTX to requester with ack count set to zero in header; set PDEX=0 in directory entry.
PLPROC_GETWB_REQ	PSH=1	Send MSG_PUT to requester; mark requester as the only sharer in directory and set Local=PSH=0.

Table 18. Handling Messages from the Processor for Remote Addresses in OriginMod

Message type	Action
PLPROC_GET_REQ	Send MSG_GET to home.
PLPROC_GETX_REQ	Send MSG_GETX to home.
PLPROC_PUTX_REQ	Send MSG_WB to home.
PLPROC_UPGRADE_REQ	Send MSG_UPGRADE to home.
PLPROC_UP_RACE_REQ	Send MSG_UP_RACE to home.
PLDP_GET_REQ	Send MSG_PUT to requester; if requester is not home, send MSG_SWB to home.
PLDP_GETX_REQ	Send MSG_PUTX to requester with ack count set to zero in header; if requester is not home, send MSG_FORWARD_ACK to home.
PLPROC_GETWB_REQ	Send MSG_PUT_WB to requester; if requester is not home, send MSG_RWB to home.

Table 19. Handling Messages from the Network for Local Addresses in OriginMod

Message type	Directory state	Action
MSG_GET	Sharers	Send MSG_PUT to requester; mark sharer in directory.
MSG_GET	Dirty, Local	Send PLDP_GET_REQ to local processor; If L2 state reply is Dirty, send MSG_PUT to requester. Else if intervention is early, register it in OTT entry. Else if intervention is late, look up writeback buffer; from the type of writeback acknowledgment (if already received) decide whether to drop the intervention; if writeback acknowledgment is received, retire writeback buffer entry. In all cases, mark sharers (two) in directory and set Dirty=0. In case of early or late intervention, mark PSH=1.
MSG_GET	Dirty, not Local	Send MSG_GET to owner; mark directory with PSH=1, Dirty=0, vector=requester.
MSG_GET	PSH=1 or PDEX=1	Send MSG_NAK to requester.
MSG_GETX	Sharers (includes no sharer case)	Send MSG_PUTX to requester with ack count (same as remote sharer count) set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1 and vector=owner in directory.
MSG_GETX	Dirty, Local	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header. Else if intervention is early, register it in OTT entry. Else if intervention is late, look up writeback buffer; from the type of writeback acknowledgment (if already received) decide whether to drop the intervention; if writeback acknowledgment is received, retire writeback buffer entry. In all cases, set vector=requester and Local=0 in directory. In case of early or late intervention, mark PDEX=1.
MSG_GETX	Dirty, not Local	Send MSG_GETX to owner; mark directory with PDEX=1 and vector=requester.
MSG_GETX	PSH=1 or PDEX=1	Send MSG_NAK to requester.
MSG_PUT	PSH=1	Send PLDP_PUT_RPLY to local processor; mark sharers (two) in directory, set PSH=0, Local=1.
MSG_PUT_WB	PSH=1	Send PLDP_PUT_RPLY to local processor; mark local processor as the only sharer in directory; set PSH=0, Local=1.

Table 20. Handling Messages from the Network for Local Addresses in OriginMod (Cont.)

Message type	Directory state	Action
MSG_PUTX	PDEX=1, Dirty, Local	Send PLDP_PUTX_RPLY to local processor; set PDEX=0 in directory; ack count must be zero, so mark write completion; send any blocked early intervention to local processor; when intervention reply arrives send PLDP_GET_REQ or PLDP_GETX_REQ to protocol processor (these are intervention replies).
MSG_WB	PSH=PDEX=0	Send MSG_WB_ACK to source; zero out directory entry.
MSG_WB	PSH=1	Send MSG_WB_ACK_INT to source; If vector=local processor send PLDP_PUT_FORWARD to local processor, set PSH=0, Local=1, and mark sharer in directory. Otherwise send MSG_PUT_FORWARD to processor marked in vector, set PSH=0 and mark sharer in directory.
MSG_WB	PDEX=1	If source≠vector, send MSG_WB_ACK_INT to source; if vector=local processor send PLDP_PUTX_FORWARD to local processor with ack count set to zero in the header; mark write completion; otherwise if vector≠local processor send MSG_PUTX_FORWARD to processor marked in the directory vector. In both cases set PDEX=0 in directory. If source=vector, send MSG_WB_ACK to source; set directory to zero, but keep PDEX=1.
MSG_INVALID_ACK	X	Send PLDP_INVALID_ACK to processor interface; decrement ack count in OTT entry; if ack count is zero in OTT entry, do the following; mark write completion; if early intervention reply is pending, send PLDP_GET_REQ or PLDP_GETX_REQ protocol processor (these are intervention replies); if writeback is pending, send PLPROC_PUTX_REQ to protocol processor; if both intervention reply and writeback are pending, send PLPROC_GETWB_REQ to protocol processor.
MSG_FORWARD_ACK	PDEX=1	Set PDEX=0 in directory.

Table 21. Handling Messages from the Network for Local Addresses in OriginMod (Cont.)

Message type	Directory state	Action
MSG_SWB	PSH=1	Mark both sharers in directory; set PSH=0.
MSG_RWB	PSH=1	Mark the only sharer in directory; set PSH=0.
MSG_UPGRADE	Sharers (without requester)	Read memory; send MSG_PUTX to requester with ack count set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1 and vector=owner in directory.
MSG_UPGRADE	Sharers (with requester)	Send MSG_UPGRADE_ACK to requester with ack count set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1 and vector=owner in directory.
MSG_UPGRADE	Dirty, Local	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header. Else if intervention is early, register it in OTT entry. Else if intervention is late, look up writeback buffer; from the type of writeback acknowledgment (if already received) decide whether to drop the intervention; if writeback acknowledgment is received, retire writeback buffer entry. In all cases, set vector=requester and Local=0 in directory. In case of early or late intervention, mark PDEX=1.
MSG_UPGRADE	Dirty, not Local	Send MSG_GETX to owner; mark directory with PDEX=1 and vector=requester.
MSG_UPGRADE	PSH=1 or PDEX=1	Send MSG_NAK to requester.
MSG_UP_RACE	X	Send MSG_PUTX to requester.

Table 22. Handling Messages from the Network for Remote Addresses in OriginMod

Message type	Action
MSG_GET	Send PLDP_GET_REQ to local processor; If L2 state reply is Dirty, send MSG_PUT to requester; if requester is not home, send MSG_SWB to home. Else if intervention is early, register it in OTT entry. Else if intervention is late, look up writeback buffer; from the type of writeback acknowledgment (if already received) decide whether to drop the intervention; if writeback acknowledgment is received retire writeback buffer entry.
MSG_GETX	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header; if requester is not home, send MSG_FORWARD_ACK to home. For late or early intervention do the same as above in MSG_GET.
MSG_PUT	Send PLDP_PUT_RPLY to local processor; if OTT entry is invalidated change reply to PLDP_NAK_RPLY.
MSG_PUT_FORWARD	Send PLDP_PUT_FORWARD to processor interface where the message type is changed to PLDP_PUT_RPLY after data re-ordering.
MSG_PUTX	Send PLDP_PUTX_RPLY to local processor; add ack count in header to that in OTT entry; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to local processor; block the intervention reply until ack count is zero.
MSG_PUTX_FORWARD	Send PLDP_PUTX_FORWARD to processor interface where the message type is changed to PLDP_PUTX_RPLY after data re-ordering; do the same as above in MSG_PUTX.
MSG_INVALID	Send MSG_INVALID_ACK to writer; send PLDP_INVALID_REQ to local processor.
MSG_UPGRADE_ACK	Send PLDP_ACK_RPLY to local processor. If OTT entry is invalidated, issue PLPROC_UP_RACE_REQ to protocol processor; add ack count in header to that in OTT entry; If OTT entry is not invalidated do the following; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to local processor; block the intervention reply until ack count is zero.
MSG_NAK	Send PLDP_NAK_RPLY to local processor.
MSG_PUT_WB	Send PLDP_PUT_RPLY to local processor; if OTT entry is invalidated change reply to PLDP_NAK_RPLY.

Table 23. Handling Messages from the Network for Remote Addresses in OriginMod (Cont.)

Message type	Action
MSG_INVALID_ACK	Send PLDP_INVALID_ACK to processor interface; decrement ack count in OTT entry; if ack count is zero in OTT entry, do the following; mark write completion; if early intervention reply is pending, send PLDP_GET_REQ or PLDP_GETX_REQ to protocol processor (these are intervention replies); if writeback is pending, send PLPROC_PUTX_REQ to protocol processor; if both intervention reply and writeback are pending, merge them, and send PLPROC_GETWB_REQ to protocol processor.
MSG_WB_ACK	Look up writeback buffer; if there is any pending intervention, identify it as early and register it with the correct OTT entry; retire writeback buffer entry.
MSG_WB_ACK_INT	Look up writeback buffer; if pending intervention is received, retire writeback buffer entry; otherwise mark writeback buffer entry as acknowledged with pending intervention.

Table 24. Avoiding Local NACKs in OriginMod+RComb

Message type	Directory state	Action
PLPROC_GET_REQ	PSH=1 or PDEX=1, R0V=0	Set R0V=1, R0=requester, OFF=quad-word offset of address.
PLPROC_GET_REQ	PSH=1 or PDEX=1, R0V=1, PRV=0	Allocate a read pending list entry; set PRV=1, set PRID=allocated entry index; set requesting node number and quad-word offset in the allocated entry; set the next pointer of allocated entry to NULL.
PLPROC_GET_REQ	PSH=1 or PDEX=1, R0V=1, PRV=1	Allocate a read pending list entry; set the next pointer of allocated entry to the entry with index PRID; set requesting node number and quad-word offset in the allocated entry; set PRID=allocated entry index.
PLPROC_GETX_REQ	PSH=1 or PDEX=1, PWV=0	Allocate a write pending list entry; set PWV=1, set PWID=allocated entry index; set requesting node number and quad-word offset in the allocated entry; mark UPGRADE=0 in the allocated entry; set the next pointer of allocated entry to NULL.
PLPROC_GETX_REQ	PSH=1 or PDEX=1, PWV=1	Allocate a write pending list entry; set the next pointer of allocated entry to the entry with index PWID; set requesting node number and quad-word offset in the allocated entry; mark UPGRADE=0 in the allocated entry; set PWID=allocated entry index.
PLPROC_UPGRADE_REQ	PSH=1 or PDEX=1, PWV=0	Allocate a write pending list entry; set PWV=1, set PWID=allocated entry index; set requesting node number and quad-word offset in the allocated entry; mark UPGRADE=1 in the allocated entry; set the next pointer of allocated entry to NULL.
PLPROC_UPGRADE_REQ	PSH=1 or PDEX=1, PWV=1	Allocate a write pending list entry; set the next pointer of allocated entry to the entry with index PWID; set requesting node number and quad-word offset in the allocated entry; mark UPGRADE=1 in the allocated entry; set PWID=allocated entry index.

Table 25. New Memory Controller to Processor Messages in OriginMod+DSH+WSF

Message type (encoding)	Description
PLDP_GET_SHARED_REQ (0x1)	External read intervention for cache line in shared owned state
PLDP_GETX_SHARED_REQ (0x2)	External read-exclusive intervention for cache line in shared owned state
PLDP_PUT_OWNER_RPLY (0xe)	Read reply with shared ownership

Table 26. New Network Messages in OriginMod+DSH+WSF

Message type (encoding)	Description
MSG_GET_SHARED (0x01)	Read intervention for cache line in shared owned state
MSG_GETX_SHARED (0x02)	Read-exclusive intervention for cache line in shared owned state
MSG_PUT_OWNER (0x0e)	Read reply with shared ownership

Table 27. Handling Messages from the Processor for Local Addresses in OriginMod+DSH+WSF

Message type	Directory state	Action
PLPROC_GET_REQ	DirtySH=Dirty=0	Send PLDP_PUT_RPLY to processor; mark sharer in vector and set Local=1.
PLPROC_GET_REQ	Dirty	Send MSG_GET to owner; mark Dirty=0, DirtySH=Local=1, and OWNER=requester; mark the previous owner in sharer vector.
PLPROC_GET_REQ	DirtySH	Send MSG_GET_SHARED to owner; mark Local=1 and OWNER=requester; mark the previous owner in sharer vector.
PLPROC_GETX_REQ	Dirty=DirtySH=0	Send PLDP_PUTX_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, OWNER=requester, and vector=0 in directory.
PLPROC_GETX_REQ	Dirty	Send MSG_GETX to owner; mark Local=1 and OWNER=requester.
PLPROC_GETX_REQ	DirtySH	Send MSG_GETX_SHARED to owner with ack count=number of sharers in vector (this will be copied over at the owner when the reply is sent); send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, DirtySH=0, OWNER=requester, and vector=0 in directory.
PLPROC_PUTX_REQ	Dirty=1 or DirtySH=1	If OWNER=local processor writeback cache line; mark Dirty=DirtySH=0 in directory; acknowledge writeback in local writeback buffer.
PLPROC_PUTX_REQ	Dirty=DirtySH=0	Acknowledge writeback in local writeback buffer.
PLDP_GET_REQ	X	Send MSG_PUT_OWNER to requester.
PLDP_GETX_REQ	X	Send MSG_PUTX to requester.
PLPROC_GETWB_REQ	X	Send MSG_PUT_OWNER to requester; mark Local=0 in directory entry.

Table 28. Handling Messages from the Processor for Local Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Directory state	Action
PLPROC_UPGRADE_REQ	Dirty=DirtySH=0, requester is a sharer	Send PLDP_ACK_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, OWNER=requester, and vector=0 in directory.
PLPROC_UPGRADE_REQ	Dirty=DirtySH=0, requester is not a sharer	Read memory; Send PLDP_PUTX_RPLY to processor with ack count set in the header; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, OWNER=requester, and vector=0 in directory.
PLPROC_UPGRADE_REQ	Dirty	Send MSG_GETX to owner; mark Local=1 and OWNER=requester.
PLPROC_UPGRADE_REQ	DirtySH, OWNER=requester	Send PLDP_ACK_RPLY to processor with ack count=number of sharers in vector; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, OWNER=requester, DirtySH=0, and vector=0 in directory.
PLPROC_UPGRADE_REQ	DirtySH, OWNER≠requester	Send MSG_GETX_SHARED to owner with ack count=number of sharers in vector (this will be copied over at the owner when the reply is sent); send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=Local=1, OWNER=requester, DirtySH=0, and vector=0 in directory.
PLPROC_UP_RACE_REQ	X	Send PLDP_PUTX_RPLY to processor; ignore ack count in reply header; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to processor; block the intervention reply until ack count is zero.

Table 29. Handling Messages from the Processor for Remote Addresses in OriginMod+DSH+WSF

Message type	Action
PI_PROC_GET_REQ	Send MSG_GET to home.
PI_PROC_GETX_REQ	Send MSG_GETX to home.
PI_PROC_UPGRADE_REQ	Send MSG_UPGRADE to home.
PI_PROC_UP_RACE_REQ	Send MSG_UP_RACE to home.
PI_PROC_PUTX_REQ	Send MSG_WB to home.
PI_DP_GET_REQ	Send MSG_PUT_OWNER to requester.
PI_DP_GETX_REQ	Send MSG_PUTX to requester.
PI_PROC_GETWB_REQ	Send MSG_PUT_OWNER to requester.

Table 30. Handling Messages from the Network for Local Addresses in OriginMod+DSH+WSF

Message type	Directory state	Action
MSG_GET	DirtySH=Dirty=0	Send MSG_PUT to requester; mark sharer in directory entry.
MSG_GET	Dirty, Local	Send PLDP_GET_REQ to local processor; If L2 state reply is Dirty send MSG_PUT to requester; writeback cache line; set Dirty=0, mark two sharers in vector. If L2 state reply is not Dirty If intervention is early, register it in OTT entry. If intervention is late, fill data from writeback buffer and send PLDP_GET_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry; mark Dirty=0, DirtySH=1; mark OWNER=requester; mark old owner in the sharer vector.
MSG_GET	Dirty, not Local	Send MSG_GET to owner; set Dirty=0, DirtySH=1, and OWNER=requester; mark the old owner as a sharer in the vector.
MSG_GET	DirtySH	If owner is local send PLDP_GET_SHARED_REQ to local processor; If L2 state reply is Owned send MSG_PUT to requester; writeback cache line; set DirtySH=0, mark two new sharers in vector. If L2 state reply is not Owned If intervention is early, register it in OTT entry. If intervention is late, fill data from writeback buffer and send PLDP_GET_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry; mark OWNER=requester; mark old owner as a sharer in vector. If owner is remote send MSG_GET_SHARED to owner; mark OWNER=requester; mark old owner as a sharer in vector.

Table 31. Handling Messages from the Network for Local Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Directory state	Action
MSG_GETX	DirtySH=Dirty=0	Send MSG_PUTX to requester with ack count (same as remote sharer count) set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval's left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, OWNER=requester, and vector=0 in directory.
MSG_GETX	Dirty, Local	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send PLDP_GETX_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry. In all cases, set OWNER=requester and Local=0 in directory.
MSG_GETX	Dirty, not Local	Send MSG_GETX to owner; mark directory with OWNER=requester.
MSG_GETX	DirtySH	If owner is local send PLDP_GETX_SHARED_REQ to local processor; If L2 state reply is Owned send MSG_PUTX to requester with ack count=number of sharers in directory vector; set DirtySH=Local=0, Dirty=1, OWNER=requester. If L2 state reply is not Owned If intervention is early, register it in OTT entry. If intervention is late, fill data from writeback buffer and send PLDP_GETX_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry; mark DirtySH=Local=0, Dirty=1, OWNER=requester. If owner is remote send MSG_GETX_SHARED to owner with ack count=number of sharers in directory vector (this will be copied over at the owner when the reply is sent); mark DirtySH=0, Dirty=1, OWNER=requester. In all cases invalidate the sharers (other than the requester) marked in vector and set Local=0 if local processor was a sharer.

Table 32. Handling Messages from the Network for Local Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Directory state	Action
MSG_WB	Dirty=1 or DirtySH=1	If OWNER=source writeback cache line; send MSG_WB_ACK to source; mark Dirty=DirtySH=0. If OWNER≠source send MSG_WB_ACK_INT to source.
MSG_WB	Dirty=DirtySH=0	Send MSG_WB_ACK_INT to source.
MSG_UPGRADE	DirtySH=Dirty=0, requester is a sharer	Send MSG_UPGRADE_ACK to requester with ack count (same as remote sharer count) set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, OWNER=requester, and vector=0.
MSG_UPGRADE	DirtySH=Dirty=0, requester is not a sharer	Read memory; send MSG_PUTX to requester with ack count (same as remote sharer count) set in the header; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark Dirty=1, OWNER=requester, and vector=0.
MSG_UPGRADE	Dirty, Local	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send PLDP_GETX_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry. In all cases, set OWNER=requester and Local=0.
MSG_UPGRADE	Dirty, not Local	Send MSG_GETX to owner; mark directory with OWNER=requester.
MSG_UPGRADE	DirtySH, OWNER=requester	Send MSG_UPGRADE_ACK to requester with ack count=number of sharers in vector; send PLDP_INVALID_REQ to local processor (if Local=1) and set Local=0 in directory; send invalidations (MSG_INVALID) until NI queue fills; if more inval left, save the remaining sharer vector in software queue for future dispatch; mark DirtySH=0, Dirty=1, vector=0.

Table 33. Handling Messages from the Network for Local Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Directory state	Action
MSG_UPGRADE	DirtySH, requester is not OWNER	<p>If owner is local send PLDP_GETX_SHARED_REQ to local processor; If L2 state reply is Owned send MSG_PUTX to requester with ack count=number of sharers in directory vector; set DirtySH=Local=0, Dirty=1, OWNER=requester. If L2 state reply is not Owned If intervention is early, register it in OTT entry. If intervention is late, fill data from writeback buffer and send PLDP_GETX_REQ to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received, retire writeback buffer entry; mark DirtySH=Local=0, Dirty=1, and OWNER=requester.</p> <p>If owner is remote send MSG_GETX_SHARED to owner with ack count=number of sharers in directory vector (this will be copied over at the owner when the reply is sent); mark DirtySH=0, Dirty=1, OWNER=requester.</p> <p>In all cases invalidate the sharers (other than the requester) marked in vector and set Local=0 if local processor was a sharer.</p>
MSG_UP_RACE	X	Send MSG_PUTX to requester.
MSG_PUT	X	Send PLDP_PUT_RPLY to local processor; if OTT entry is invalidated change reply to PLDP_NAK_RPLY.
MSG_PUT_OWNER	X	<p>Send PLDP_PUT_OWNER_RPLY to local processor; If PLDP_GET_SHARED_REQ is pending in OTT fill data from reply buffer; send intervention reply (PLDP_GET_REQ) to protocol processor; change local processor reply to PLDP_PUT_RPLY; if OTT entry is invalidated, send PLDP_INVALID_REQ to local processor.</p> <p>If PLDP_GETX_SHARED_REQ is pending in OTT fill data from reply buffer; send intervention reply (PLDP_GETX_REQ) to protocol processor; change local processor reply to PLDP_PUT_RPLY; send PLDP_INVALID_REQ to local processor.</p>

Table 34. Handling Messages from the Network for Local Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Directory state	Action
MSG.PUTX	X	Send PLDP_PUTX_REPLY to local processor; add ack count in header to that in OTT entry; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to local processor; block the intervention reply until ack count is zero; if there is blocked early intervention and OTT entry is invalidated, send PLDP_INVALID_REQ to local processor.
MSG.INVALID_ACK	X	Send PLDP_INVALID_ACK to processor interface; decrement ack count in OTT entry; if ack count is zero in OTT entry, do the following; mark write completion; if early intervention reply is pending, send PLDP_GET_REQ or PLDP_GETX_REQ to protocol processor (these are intervention replies); if writeback is pending, send PLPROC_PUTX_REQ to protocol processor; if both intervention reply and writeback are pending, send PLPROC_GETWB_REQ to protocol processor.

Table 35. Handling Messages from the Network for Remote Addresses in OriginMod+DSH+WSF

Message type	Action
MSG_GET	Send PLDP_GET_REQ to local processor; If L2 state reply is Dirty, send MSG_PUT_OWNER to requester. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send intervention reply (PLDP_GET_REQ) to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received retire writeback buffer entry.
MSG_GET_SHARED	Send PLDP_GET_SHARED_REQ to local processor; If L2 state reply is Owned, send MSG_PUT_OWNER to requester. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send intervention reply (PLDP_GET_REQ) to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received retire writeback buffer entry.
MSG_GETX	Send PLDP_GETX_REQ to local processor; If L2 state reply is Dirty, send MSG_PUTX to requester with ack count set to zero in header. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send intervention reply (PLDP_GETX_REQ) to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received retire writeback buffer entry.
MSG_GETX_SHARED	Send PLDP_GETX_SHARED_REQ to local processor; If L2 state reply is Owned, send MSG_PUTX to requester. Else if intervention is early, register it in OTT entry. Else if intervention is late, fill data from writeback buffer and send intervention reply (PLDP_GETX_REQ) to protocol processor; mark writeback buffer entry as intervention received; if writeback acknowledgment is received retire writeback buffer entry.
MSG_PUT	Send PLDP_PUT_RPLY to local processor; if OTT entry is invalidated change reply to PLDP_NAK_RPLY.
MSG_PUT_OWNER	Send PLDP_PUT_OWNER_RPLY to local processor; If PLDP_GET_SHARED_REQ is pending in OTT fill data from reply buffer; send intervention reply (PLDP_GET_REQ) to protocol processor; change local processor reply to PLDP_PUT_RPLY; if OTT entry is invalidated, send PLDP_INVALID_REQ to local processor. If PLDP_GETX_SHARED_REQ is pending in OTT fill data from reply buffer; send intervention reply (PLDP_GETX_REQ) to protocol processor; change local processor reply to PLDP_PUT_RPLY; send PLDP_INVALID_REQ to local processor.

Table 36. Handling Messages from the Network for Remote Addresses in OriginMod+DSH+WSF (Cont.)

Message type	Action
MSG_PUTX	Send PLDP_PUTX_RPLY to local processor; add ack count in header to that in OTT entry; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to local processor; block the intervention reply until ack count is zero; if there is blocked early intervention and OTT entry is invalidated, send PLDP_INVALID_REQ to local processor.
MSG_UPGRADE_ACK	Send PLDP_ACK_RPLY to local processor. If OTT entry is invalidated, issue PLPROC_UP_RACE_REQ to protocol processor; add ack count in header to that in OTT entry; If OTT entry is not invalidated do the following; if ack count is zero in OTT entry, mark write completion; send any blocked early intervention to local processor; block the intervention reply until ack count is zero.
MSG_INVALID	Send MSG_INVALID_ACK to writer; send PLDP_INVALID_REQ to local processor.
MSG_INVALID_ACK	Send PLDP_INVALID_ACK to processor interface; decrement ack count in OTT entry; if ack count is zero in OTT entry, do the following; mark write completion; if early intervention reply is pending, send PLDP_GET_REQ or PLDP_GETX_REQ to protocol processor (these are intervention replies); if writeback is pending, send PLPROC_PUTX_REQ to protocol processor; if both intervention reply and writeback are pending, merge them, and send PLPROC_GETWB_REQ to protocol processor.