Exploring Virtual Network Selection Algorithms in DSM Cache Coherence Protocols

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

Abstract—Distributed shared memory (DSM) multiprocessors typically require disjoint networks for deadlock-free execution of cache coherence protocols. This is normally achieved by implementing virtual networks with the help of virtual channels or virtual lanes multiplexed on a single physical network. To keep the coherence protocol simple, messages are usually assigned to virtual lanes in a predefined static manner based on a cycle-free lane assignment dependence graph. However, this static split of virtual networks (such as request and reply networks) may lead to underutilization of certain virtual networks while saturating the other networks. In this paper, we explore different static and dynamic schemes to select the virtual lanes for outgoing messages and mix the load among them without restricting any particular type of message to be carried only by a particular virtual network. We achieve this by exposing the selection algorithms to the coherence protocol itself, so that it can inject messages into selected virtual lanes based on some local information, and still enjoy deadlock-freedom. Our execution-driven simulation on five applications from the SPLASH-2 suite shows that as the system scales, the virtual network selection algorithms play an important role. For 128-node systems, our dynamic selection algorithm speeds up parallel execution by as much as 22 percent over an optimized baseline system running a modified SGI Origin 2000 protocol. We also explore how network latency, the number of message buffers per virtual lane, and the depth of network interface output queues affect the relative performance of various virtual lane selection algorithms.

Index Terms—Distributed shared memory, cache coherence protocol, virtual network, deadlock-freedom.

1 Introduction

Virtual channels in routers allow messages to share a common physical link [4] and are often used to implement separate virtual networks in DSM multiprocessors. Usually, each input port in the router is equipped with a number of virtual channels or virtual lanes (we will interchangeably use the terms virtual channel, virtual lane, and virtual network). Further, each virtual lane of a router port usually maps to an input queue in the network interface (NI) of the memory controller, meaning that a message in a particular virtual lane will be drained into the corresponding NI input queue at the destination node, while a message in an NI output queue will get injected into the corresponding virtual network and arrive on the corresponding incoming virtual lane at the destination node. For example, if every router port has four virtual lanes, the NI will have four input and output queues. These virtual lanes usually serve two purposes. First, they improve physical link utilization by allowing messages to overtake blocked ones on a shared physical link. Second, cache coherence protocols normally rely on the availability of separate virtual networks for deadlock-free execution. Every directory-based cache coherence protocol has certain types of messages. Read, upgrade, read-exclusive, and writeback are the four main types of requests generated by the processor. These are generated, respectively, by load misses, store misses to shared cache lines, store misses to memory lines that are not in the cache, and evicted cache lines in the modified state. In a home-based directory protocol, every cache line is assigned a home node that is responsible for maintaining the state of the cache line in a structure called a directory. On inspection of the directory, the home node may initiate a series of coherence transactions possibly terminating in a reply to the requesting node. In all the protocols considered in this paper, the reply to an intervention (i.e., a request forwarded by the home node) is directly sent to the requesting node. To service a coherent store miss, the home node sends invalidation requests to the current sharers. Acknowledgments to these messages are sent either to the home node or to the writer. To guarantee deadlock-freedom, cache coherence protocols statically assign different virtual networks to carry disjoint sets of messages. However, this may lead to underutilization of certain virtual networks. In this paper, we focus on this aspect of virtual networks and explore how cache coherence protocols can improve the utilization of the output queues in the NI and, hence, the virtual lanes in the router. We assume a deadlock-free deterministic routing algorithm. Evaluation of our virtual network selection algorithms in conjunction with adaptive routing remains a topic of future research.

Complicated virtual network selection policies have direct implication on deadlock avoidance in the cache coherence protocol. This is the reason why most DSM multiprocessors adhere to a predefined static selection policy based on a cycle-free virtual lane dependence graph. Coherence protocols usually support either two disjoint virtual networks (namely, request and reply), or sometimes three, as we shall see. The Stanford DASH multiprocessor [17], [18], the SGI Origin 2000 [16], and the Piranha chip-multiprocessor [1] belong to the

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

Manuscript received 31 Mar. 2003; revised 25 Jan. 2004; accepted 29 Jan. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0045-0303.

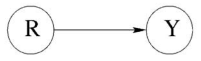


Fig. 1. Virtual lane dependence graph in a 2-lane protocol.

first category that execute two-lane protocols. In most cases, a request generates a reply and these two message types travel in separate virtual networks. Therefore, to avoid deadlock, an incoming message on the request network can be serviced only if there is space in the outgoing NI reply queue and an incoming reply message may always be serviced since it does not generate any further messages. This leads to a trivial lane dependence graph as shown in Fig. 1.

In this graph, "R" stands for the request network while "Y" stands for the reply network. The arc shows that an incoming message on the request network may require space in the reply network and, hence, cannot be safely scheduled by the memory controller until space is available. In these protocols, a problem arises when a request generates another (or several) request(s) which, in turn, generates a reply. For example, a read-exclusive request may generate several invalidation requests which, in turn, will generate invalidation acknowledgment replies. To drain the incoming request network, the memory controller cannot rely on the availability of space in the outgoing request network since this would lead to a cycle ($R \rightarrow R$) in the lane dependence graph. The DASH multiprocessor, other than having fairly large input and output queues, resorts to a time-out mechanism to avoid deadlock in such situations. The home node waits for a predetermined number of cycles trying to get space in the outgoing request network and, if it fails, it rejects the request and sends a negative acknowledgment (NACK) on the reply network to the requester. This preserves a strict request-reply protocol, but relies on the fact that the retry from the requester will eventually succeed. Instead of NACKing, the SGI Origin 2000 resorts to a back-off invalidation or intervention mechanism and sends a reply to the requester with the remaining sharer identities to invalidate or with the identity of the exclusive owner in the case of an intervention. The Piranha internode protocol implements two virtual networks, namely, a low priority network (L) and a high priority network (H). The L network is used for sending requests other than writebacks to the home node, while the H network is used for interventions, invalidations, writebacks, and all replies. The $H \rightarrow H$ deadlock cycles are broken by providing sufficient buffering in the network.

To solve the problems of the two-lane protocols, current DSM multiprocessors normally support more than two virtual networks. The Stanford FLASH multiprocessor [10], [11], [12], [15] runs four-lane protocols. The lane dependence graph is shown in Fig. 2. In addition to having request (R) and reply (Y) lanes, the "RR" lane carries the

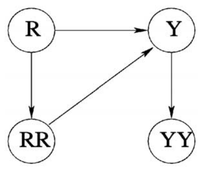


Fig. 2. Virtual lane dependence graph in a 4-lane Stanford FLASH protocol.

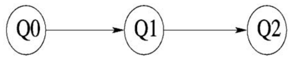


Fig. 3. Virtual lane dependence graph in AlphaServer GS320 protocol.

requests generated by request messages, i.e., invalidations and interventions. These messages are normally called forwarded requests. The other lane (YY) is responsible for consuming replies generated by the reply messages. The last invalidation acknowledgment (a reply) arriving at the home node uses this lane for sending a write completion reply message to the writer. Note that this is necessary only because the Stanford FLASH protocol collects invalidation acknowledgments at the home node. The AlphaServer GS320 [7] uses a three-lane protocol. The virtual lanes are named Q0, Q1, and Q2. The Q0 lane carries requests from a processor to a home. The replies and the forwarded requests (e.g., invalidations and interventions) from the home node travel on the Q1 network. The Q2 lane carries the replies to the forwarded requests. These replies are directly sent to the requester. This leads to a lane dependence graph, as shown in Fig. 3. In all these cases, the processor interface typically injects requests into the request network, thereby initiating a chain of transactions possibly terminating in a reply back to the requester.

Supporting adaptive routing requires an even larger number of virtual networks. The minimally adaptive two-dimensional torus router in the Alpha 21364 [21], [22] implements 19 virtual channels. Three virtual channels form a virtual network which is responsible for routing exactly one of the six types of coherence protocol messages. This accounts for 18 virtual channels. The remaining virtual channel is responsible for routing a seventh special type of coherence protocol message. Out of the three virtual channels within a nonspecial virtual network, one forms the adaptive routing network, while the other two form the deadlock-free dimension-order routing network.

From the above discussion, it is clear that the trend in DSM multiprocessors is to statically assign message types to the virtual networks. For example, a request can only travel on the request network, etc. The AlphaServer GS320 provides some flexibility by mixing replies and forwarded requests in the Q1 network. But, still, the processor interface can inject requests only in the Q0 network. In addition, there always exists a static sink—the reply network for two-lane protocols, the YY lane in the Stanford FLASH protocol, or the Q2 lane in the AlphaServer GS320 protocol. This guarantees that the network will drain in steady state. These decisions are taken so that proving a coherence protocol deadlock-free becomes easier. However, static assignment of messages to virtual networks (a static virtual network selection policy) may lead to underutilization of certain virtual networks while the other networks approach saturation. In our experience, this case is quite common for two reasons. First, for applications with long sharing lists, the invalidation message load is normally very high and bursty when writes take place, since all the invalidation messages are injected into the same virtual network. Second, a forwarded request normally generates two replies, namely, a direct reply to the requester and a notification reply to the home node. This leads to a relatively

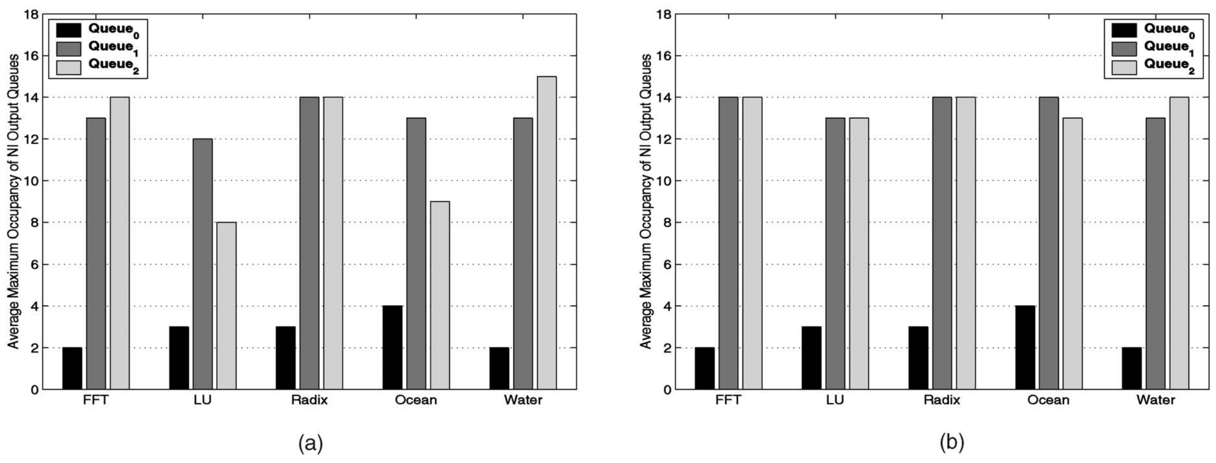


Fig. 4. Average maximum occupancy of NI output queues for (a) 64 nodes and (b) 128 nodes.

high load on the outgoing reply network as compared to the request network. This is shown in Figs. 4a and 4b.

Fig. 4a shows the maximum occupancy for each of three NI output queues averaged across all the nodes for five applications running our baseline 3-lane protocol (Section 2.2) on a 64-node system (uniprocessor nodes) built out of 2-way bristled fat hypercubes [16]. Similar results for 128 nodes are shown in Fig. 4b. The Queue_0 implements the request network, the Queue_1 implements the reply network, and the Queue_2 carries all invalidation and intervention requests. Each NI output queue has 16 slots. In both the plots, it is clear that the reply network and the forwarded request network are heavily loaded, while the request network remains mostly idle. On average, at most, four slots in the output request queue are ever occupied. In this paper, we focus on the problem of this load imbalance among the virtual networks arising from simple static selection algorithms. Following the SGI Spider router [6] used in the SGI Origin 2000, we fix the number of virtual lanes at four. We propose one fixed priority and one round robin selection algorithm for balancing the invalidation load. Next, we augment the low-overhead fixed priority algorithm with a static and a dynamic selection algorithm to more evenly distribute the network load. We carry out execution-driven simulations on five applications from the SPLASH-2 suite [26] assuming a deterministic routing algorithm [5], which is commonly used in DSM multiprocessors. Note that, although this study does not consider adaptive routing as used in Piranha [1], Cray T3E [24], and Alpha 21364 [21], [22], our techniques can be directly applied to such systems. More specifically, our algorithms do not assume anything about the routing algorithms. Also, our virtual network selection algorithms are different from adaptive routing algorithms because adaptive routing, instead of switching between virtual networks, only executes an adaptive routing algorithm within each virtual network (which may have multiple virtual channels). Our dynamic selection algorithm achieves up to 22 percent speedup for 128 nodes over the baseline. We further explore the relative performance of the algorithms as the network latency, the number of message buffers per virtual lane, and the depths of the NI output queues are varied.

In the next section, we briefly present related work in the field of evaluating network parameters using execution-driven simulation of DSM multiprocessors. Section 2 presents our baseline system architecture, Section 3 introduces our virtual network selection algorithms, Sections 4 and 5 discuss our simulation results, and Section 6 concludes the paper.

1.1 Related Work

Much has been done to investigate the effects of network parameters in DSM multiprocessor systems with synthetic workloads. However, execution-driven simulation for studying the effects of architectural changes on DSM multiprocessors is of paramount importance. Kumar and Bhuyan [14] use execution-driven simulation to investigate the impact of the number of virtual lanes and the number of buffers per virtual lane in a DSM multiprocessor built out of a 2D torus. The study does not consider the coherence protocol aspects and is orthogonal to our proposal. It concludes that both parameters are important for parallel performance and that four virtual lanes per port and two to four buffers per lane are sufficient. Vaidya et al. [25] explore the impact of virtual lanes and adaptivity on parallel performance using a 2D mesh, and conclude that none of them contribute significantly to end-performance. Martínez et al. [20] carry out execution-driven simulations to explore the impact of virtual lanes and adaptivity in bristled hypercubes, and conclude that in these systems parallel performance can be greatly improved by introducing more virtual lanes and adaptivity while the nonbristled systems remain largely insensitive to these effects. The study uses a conventional two-lane coherence protocol. In contrast to all these studies, we focus on virtual network selection algorithms in coherence protocols and explore the impact of the network parameters on the relative performance of these algorithms.

The Spider router [6] used with the SGI Origin 2000 provides minimal support for congestion control within the request and the reply networks. The router has four virtual lanes and two lanes can be assigned to each virtual network. One bit in the header is used to switch messages between two virtual lanes within the same virtual network. This helps in

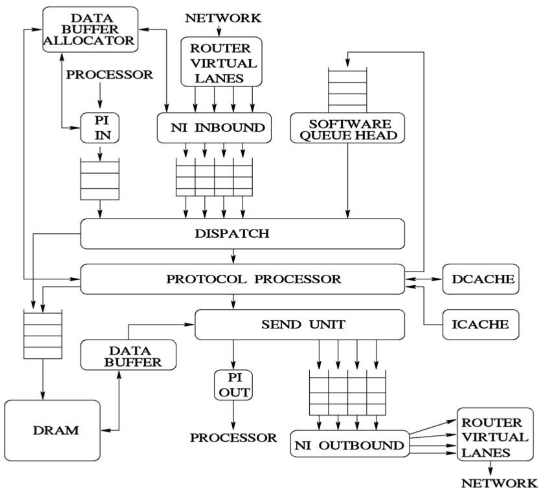


Fig. 5. Node controller architecture.

selecting the least used virtual lane within a virtual network. We are not aware whether this feature is actually used in the SGI Origin 2000. However, this mechanism still does not allow complete mixing of the virtual networks, i.e., a virtual lane belonging to the reply network cannot be used to inject requests. Our algorithms try to achieve this mix as much as possible, thereby improving utilization of the NI output queues, while still keeping the protocol deadlock-free.

2 System Architecture

This section describes our system architecture including the node controller architecture and the cache coherence protocol.

2.1 Node Controller Architecture

Our node controller architecture shown in Fig. 5 is directly derived from the Memory And General Interconnect Controller (MAGIC) of the Stanford FLASH multiprocessor.

It is very similar to the hub of the SGI Origin 2000 with the exception that our node controller is programmable and can execute any cache coherence protocol. Such flexible protocol processors, used in the Piranha system [1], STiNG multiprocessor [19], S3.mp [23], etc., allow late binding of the protocol, flexibility in the choice of protocol, and a relatively easy and fast protocol verification phase. Coherence messages arrive at the processor interface (PI inbound) or the network interface (NI inbound) 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 has 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 scheduling among the PI queues (there are two PI input queues; see Section 2.2 for details), four NI input queues, and the software queue (see below). After selecting a message, a table lookup decides which protocol message handler is invoked to service the scheduled message. A message arriving on a certain virtual network may require space in a number of outgoing networks. This is specified in the form of a dependence graph and is programmed in the dispatch unit at boot time. Before scheduling a message, the dispatch unit guarantees that the availability of virtual lane

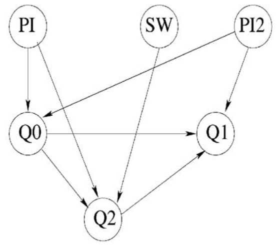

$$\begin{tikzcd}
PI \arrow[d] \arrow[drr] & & SW \arrow[dd] & & PI2 \arrow[dllll] \arrow[d] \\
Q0 \arrow[rrrr] \arrow[drr] & & & & Q1 \\
& & Q2 \arrow[urr] & &
\end{tikzcd}$$

Fig. 6. Cycle-free virtual lane dependence graph in baseline protocol.

space in required outgoing networks exceeds a minimum threshold (2 in this study). Still, it may happen that, while running a handler, the protocol processor finds that it needs more space in an outgoing network queue—for example, this situation may arise while sending out invalidation messages. At this point, the incomplete message is stored in the software queue, which is a reserved space in main memory. At some later point, this message will get scheduled from the head of the software queue by the dispatch unit and will continue sending the remaining invalidations. The protocol processor has dedicated protocol instruction and data caches backed by main memory. During the handler execution, the protocol processor may instruct the send unit to send out certain types of messages (such as data requests/replies or other coherence messages) to either the local processor (via PI outbound) or remote nodes (via NI outbound).

2.2 Cache Coherence Protocol

This section describes the baseline cache coherence protocol on top of which all the virtual network selection algorithms run. Our protocol is a simplified MSI version of the SGI Origin 2000 protocol. This protocol differs from the actual Origin protocol in four notable respects. First, our protocol is MSI as opposed to MESI and, consequently, the home node does not send speculative replies to the requester when the line is dirty at a third node. Second, our protocol supports eager-exclusive replies (as opposed to strict sequential consistency) where upgrade acknowledgments and read-exclusive data replies are immediately sent to the requester even before all the invalidation messages are sent and all the acknowledgments are collected. Our relaxed consistency model guarantees "global completion" of all writes on release boundaries, thereby preserving the semantics of flags, locks, and barriers. Third, our protocol sends an exclusive data reply (versus a negative acknowledgment) if an upgrade request comes from a node that is not marked as a sharer in the directory. A write followed by a writeback to the same cache line from a different node reaching the home node before this upgrade can lead to such a race. Fourth, our protocol uses three virtual lanes, namely, Q0, Q1, and Q2, as opposed to a two-lane strict request-reply mechanism. The Q0 lane is the request lane, the Q1 lane is the reply lane and also serves as the sink, and the Q2 lane is used to send out forwarded requests, i.e., invalidations and interventions. This eliminates the necessity of implementing any back-off mechanism. The lane dependence graph for our protocol is shown in Fig. 6. Note that the lane dependence graph is exactly the same as that in the Stanford FLASH protocol, except that the fourth virtual lane is not used. The processor interface (PI) uses the Q0 network for sending requests to the home node and the Q2 network for sending forwarded intervention and invalidation requests. The second input queue (PI2) in the processor interface requires space in the Q0 network for sending blocked writebacks and space in the Q1 network for sending blocked intervention replies (see below for detail). The software queue (SW) uses only the Q2 network since it only sends out pending invalidation messages. Note that the protocol does not rely on any message ordering in the network. As a result, we can safely ignore this issue while designing our virtual network selection algorithms.

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 pending shared bit is set when a read request is forwarded by the home node to the current exclusive owner. This bit gets cleared when the home node receives a sharing writeback message from the previous owner. Similarly, the pending dirty exclusive bit is set when a read-exclusive request is forwarded by the home node to the current exclusive owner. This bit gets cleared when the home node receives an ownership transfer message from the previous owner. In both of the forwarding cases, the intervention reply is directly sent to the requester by the owner if it can satisfy the request (i.e., if the owner still caches the memory line). The local bit in the directory entry is set when the local processor caches the line. This bit is used to quickly decide whether an invalidation or intervention needs to go over the network interface or not. The sharer vector is 32 bits wide and, depending on the directory state, it is used to store either the sharer identities in the form of a bit vector or the identity of the dirty exclusive owner. The remaining 28 bits are left unused for future extensions of the protocol. Whenever the number of nodes in the system is larger than 32, the protocol scales by becoming a coarsevector protocol [9] with a coarseness of 2 and 4 for 64 and 128-node systems, respectively, so that each bit in the sharer vector represents a cluster of two or four consecutive nodes. For multiprocessor nodes, a single bit in the directory entry keeps track of the sharing information for each node (information is not maintained per processor).

As in the Origin protocol, our protocol collects the invalidation acknowledgments at the requester. The write reply from the home node carries the expected number of acknowledgments. The requester node remembers this count in the outstanding transaction table (OTT) and decrements the count as it receives the acknowledgments. However, the write reply is immediately sent to the processor so that it can continue with the write. Writebacks for that line and any intervention to that line are blocked in the OTT until all the acknowledgments are collected. These will be referred to as blocked writebacks and blocked interventions. In addition to the actual PI input queue, there is a 4-entry bypass input queue (named PI2) in the processor interface that is used by the OTT control hardware to put blocked writebacks and intervention replies after the last invalidation acknowledgment is collected. We also merge any blocked writeback with blocked intervention replies in case both a writeback and an intervention to the same memory line are blocked at the OTT, thereby saving some network messages. Our protocol correctly handles the early and late intervention races. An early intervention race occurs when a forwarded intervention races past the write reply and reaches the owner first. Early interventions are buffered in the OTT in the processor interface of the designated owner. A late intervention race occurs when a forwarded intervention reaches the owner after it has issued a writeback message to the home node. Late interventions are replied to by the home node when it receives the writeback. At this time, it also clears the appropriate pending bit in the corresponding directory entry. The late interventions are ignored at the third party nodes. To properly decide which interventions to ignore, the node controller requires a writeback buffer that stores the addresses of the outstanding writeback messages until they are acknowledged by the home node. Two types of writeback acknowledgment messages are required to properly decide whether a writeback buffer entry should wait for an upcoming intervention before being recycled.

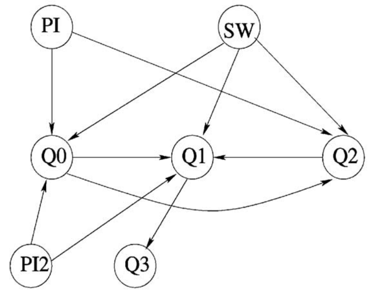


Fig. 7. Virtual lane dependence graph for balancing invalidations.

3 Virtual Network Selection Algorithms

This section describes our virtual network selection algorithms. We present two algorithms to accelerate invalidations, combine one of them with a static reply message balancing algorithm and, finally, present a completely dynamic algorithm that chooses the outgoing virtual network based upon past lane usage while maintaining deadlock-freedom.

3.1 Accelerating Invalidation Messages

In the baseline protocol presented in Section 2.2, the invalidation messages are carried only by the Q2 network. This fills up the corresponding NI output queue in bursts. When the Q2 queue fills, the protocol processor stores the remaining sharer identities in the software queue. The dispatch unit eventually schedules a handler that starts sending the pending invalidations. Quickly sending invalidation messages automatically accelerates the corresponding invalidation acknowledgments leading to earlier completion of writes. This can be done by using all the virtual networks to send out invalidations. However, these messages generate replies and, therefore, we need to leave out at least one virtual lane to carry the invalidation acknowledgments. Our algorithms use Q0, Q1, and Q2 networks to send invalidations. The invalidations sent on Q0 and Q2 generate invalidation acknowledgments on the Q1 network, while those sent on the Q1 network inject the corresponding acknowledgments into the Q3 network. This, when augmented with the graph in Fig. 6, results in the lane dependence graph, as shown in Fig. 7. The software queue now needs space in Q0, Q1, and Q2 for sending out pending invalidations. Note that Q3 acts as the static sink in this protocol.

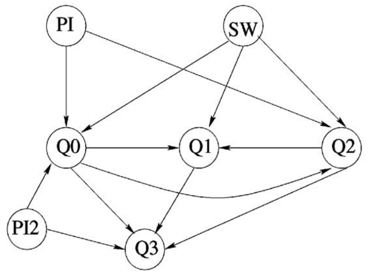


Fig. 8. Virtual lane dependence graph for balancing replies.

Having presented the central idea for distributing the invalidation load, we turn to the distribution algorithm. We have experimented with two algorithms. The first algorithm assigns a fixed priority to the three queues in the order Q2, Q0, and Q1, giving the reply network (Q1) the least priority. This is done to keep the reply network as free as possible to minimize the delay of reply messages. Therefore, a protocol message handler always sends invalidations along the Q2 network first. Once that queue fills up, the handler injects the invalidations into the Q0 network and then into the Q1 network.

The second algorithm picks the Q0, Q1, and Q2 networks in a round robin fashion. The protocol processor maintains state about the next round robin candidate and updates this state at the end of the handler. The next time the handler executes, invalidations are sent first along the round robin candidate instead of Q2 as in the fixed priority scheme. When that queue fills up, the next candidate is chosen and the process continues until all the invalidations are sent or all three queues fill.

As the results in Section 5 will show, the fixed priority algorithm performs equally well or better than the more complicated round robin algorithm. This is largely due to assigning the least priority to the reply queue. The round robin algorithm also consumes extra cycles in the protocol processor to maintain state. Therefore, in the following discussion, we distribute the invalidations according to the fixed priority algorithm only.

3.2 Balancing Reply Message Load

This section augments the fixed priority invalidation algorithm with a static algorithm to distribute the load of reply messages. While the baseline protocol only uses the Q1 network to send replies, this algorithm statically distributes the reply messages among the Q1 and Q3 networks based on message type. All the replies to forwarded requests travel along the Q3 network, including both the reply to the requester and the notification message to the home node. The home node sends read replies on the Q1 network, and write replies (i.e., read-exclusive replies and upgrade acknowledgments) and writeback acknowledgments on the Q3 network. The Q1 network is almost completely dedicated to read replies, and all other replies are grouped into the Q3 network (because of the underlying fixed priority invalidation algorithm, some invalidations and acknowledgments are sent on the Q1 network). The resulting lane dependence graph is shown in Fig. 8. Note the distinctions between this graph and the one shown in Fig. 7. The PI2 queue now needs space in the Q3 network

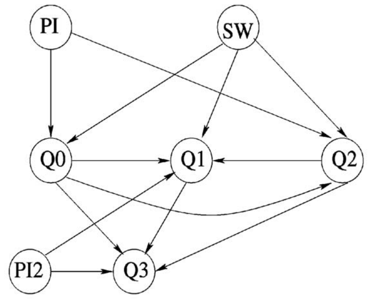


Fig. 9. Virtual lane dependence graph for the dynamic algorithm.

instead of Q1 to inject the blocked intervention replies (these are replies to forwarded requests). The incoming Q0 network (the request network) needs space in the outgoing Q3 network so that the home node can inject write replies and the writeback acknowledgments. Finally, the incoming Q2 network (that carries the forwarded requests) needs space in the Q3 network to send the replies generated by forwarded requests.

3.3 A Dynamic Selection Algorithm

In this section, we augment the fixed priority invalidation algorithm with a completely dynamic algorithm to distribute all non-invalidation messages (not only replies) among the four virtual networks. The central idea is to provide as many choices as possible in the outgoing networks to every source of incoming messages and to use all the NI output queues as equally as possible. The main constraint that must be satisfied while opening up outgoing virtual network space for a particular incoming network is that the protocol must remain deadlock-free. This is guaranteed by designing a cycle-free lane dependence graph and maintaining a static sink. The chosen lane dependence graph is shown in Fig. 9. The only difference between this graph and the one shown in Fig. 8 is that we open up space in all the virtual networks for the processor interface and assign them in a non-overlapping fashion to the PI and PI2 input queues. The PI input queue uses the Q0 and Q2 networks, while the PI2 queue uses the Q1 and Q3 networks. This is the best that can be achieved with four virtual lanes—adding an arc between any two Qi's will introduce a cycle in the graph.

This algorithm does not attach any special meaning to any virtual network, i.e., Q1 can carry any message, not only replies, etc. The memory controller is augmented with one round robin counter for each of PI, PI2, Q0, and Q2 incoming queues for selecting the corresponding outgoing queues. Q3 does not need a counter since it is the sink. Q1 does not need a counter since it has no choice other than sending all messages into Q3 as dictated by the lane dependence graph. The counter for a particular incoming queue records the next round robin outgoing network queue (i.e., one of Q0, Q1, Q2, and Q3) to be used for sending a message that gets generated by a message arriving at that incoming queue. Therefore, with four virtual lanes, each counter is 2 bits wide. Recall that every incoming queue cannot use all the outgoing queues since some arcs are absent in the lane dependence graph.

TABLE 1
Applications and Problem Sizes

Applications	Problem Sizes
FFT	1M points, blocked for L1 data cache
LU	512×512 matrix, 16×16 blocks
Radix-Sort	2M keys, radix=32
Ocean	514×514 grid, tolerance 1e-5
Water	1024 molecules, 3 time steps

The algorithm works as follows: In response to processor requests arriving from the PI, the coherence protocol forwards requests to the home node or third party nodes alternately on Q0 and Q2. The blocked intervention replies (these arrive on the PI2 queue) are sent alternately on Q1 and Q3. The blocked writebacks arriving on the PI2 queue are sent only on the Q1 network since these messages generate replies. The messages arriving on Q0 and Q2 may generate either forwarded requests or replies. The algorithm handles them as follows: The forwarded requests generated by the messages arriving on Q0 are sent alternately on Q1 and Q2, while the replies generated by these Q0 messages are sent on Q1, Q2, and Q3 in a round robin manner. Note that, since Q3 is the sink, it can only carry replies. The forwarded requests generated by the messages arriving on Q2 are sent on Q1 while the replies generated by these Q2 messages are sent on Q1 and Q3, alternately. Considering the algorithm description, it should be clear that any message arriving on the incoming Q1 network does not generate any forwarded request, but may generate replies. These replies are always sent on Q3. Q3 acts as the static sink, i.e., any message arriving on Q3 does not generate any further messages. Note that the fixed priority invalidation algorithm runs together with this algorithm.

4 Evaluation Methodology

This section discusses the applications and the simulation environment we use to evaluate our virtual lane selection algorithms.

4.1 Applications

We have chosen five programs from the SPLASH-2 benchmark suite [26]. These are shown in Table 1. There are two complete applications (Ocean and Water) and three computational kernels (FFT, LU, and Radix-Sort). The programs were chosen because they represent a variety of important scientific computations with different communication patterns and synchronization requirements. As a simple optimization, in Ocean the global error lock in the multigrid phase has been changed from a lock-test-set-unlock sequence to a more efficient test-lock-test-set-unlock sequence [13]. All five applications use page placement. FFT, LU, Radix-Sort, and Ocean use software prefetch to hide remote latency as much as possible.

4.2 Simulation Environment

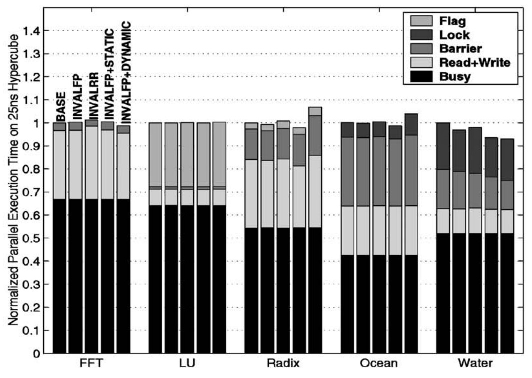
The main processor runs at 1GHz and is equipped with separate 32KB primary instruction and data caches that are two-way set associative and have line sizes of 64 bytes and 32 bytes, respectively. The secondary cache is unified, 2MB, two-way set associative and has a line size of 128 bytes. The processor ISA includes prefetch and prefetch exclusive instructions and the cache controller uses a critical double-word refill scheme. The processor model also contains a fully-associative 8-entry instruction TLB and a 64-entry data TLB with 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 [2], [8].

The embedded protocol processor in the memory controller is a dual-issue core running at 400MHz system clock frequency. The instruction and data cache behavior of the protocol processor is modeled precisely via a cycle-accurate simulator similar to that for the protocol processor in [8]. We simulate a 32KB direct-mapped protocol instruction cache and a 512KB direct-mapped protocol data cache. Our execution driven simulator models contention in detail within the memory controller, between the controller and its external interfaces, at main memory, and for the system bus. The access time of main memory SDRAM is fixed at 90ns (36 system cycles). The memory queue is 16 entries deep. The input and output queue sizes in the node controller's processor interface (PI) are set at 16 and 2 entries, respectively. The corresponding queues in the network interface (NI) are 2 and 16 entries deep. We will also explore the effects of making the NI output queues 8 and 32 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. For dual-processor nodes, sizes of the outstanding transaction table and the writeback buffer are doubled to be able to hold more outstanding requests and writebacks. Each node controller has 32 cache line-sized data buffers used for holding data as a protocol message passes through various stages of processing.

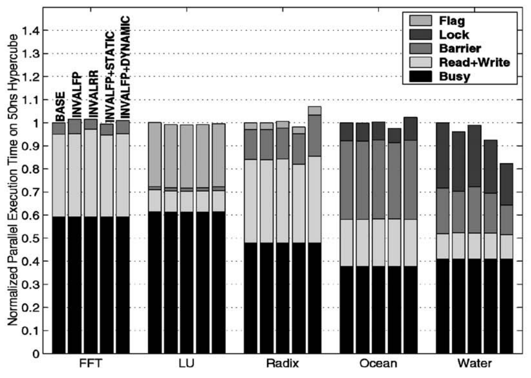
The router architecture is very similar to that used in the SGI Spider chip [6]. There are six full-duplex ports and each port has four incoming virtual lanes. We experiment with one, two, and four message buffers per virtual lane. One message is composed of a 64-bit header, a 64-bit address, and optional data payload up to 128 bytes, which is the coherence granularity. We assume that a message buffer is big enough to hold an entire message. In this context, we would like to mention that the Spider router provides 256 bytes of message buffering at each virtual lane. Our router model (like Spider) carries out a look-ahead routing algorithm that determines the output port at the next router ahead of time. We simulate a 2-way bristled fat hypercube for more than 32 nodes as done in the SGI Origin 2000 with the conventional e-cube routing. We experiment with 25ns and 50ns hop times. The simulated node-to-network link bandwidth is kept constant at 1GB/s.

5 Simulation Results

This section presents the simulation results for the virtual network selection algorithms. We evaluate the following algorithms.



(a)



(b)

Fig. 10. Normalized execution time for 64 nodes with two message buffers per virtual lane, 16 slots in NI output queues, and (a) 25ns and (b) 50ns hop time.

1. Base. The base protocol discussed in Section 2.2.
2. InvalFP. The fixed priority invalidation algorithm presented in Section 3.1.
3. InvalRR. The round robin invalidation algorithm presented in Section 3.1.
4. InvalFP+Static. The static reply balancing algorithm presented in Section 3.2.
5. InvalFP+Dynamic. The dynamic algorithm presented in Section 3.3. This algorithm requires nominal hardware support in the NI for selecting the round robin candidate from the legal outgoing virtual networks as dictated by the lane dependence graph.

We present results for both 64-processor and 128-processor DSM systems. For 64 processors, we evaluate uniprocessor nodes (64 nodes) and dual-processor nodes (32 nodes). For 128 processors, we only evaluate uniprocessor node systems. We also evaluate how the relative performance of these algorithms changes as the network latency, number of message buffers per virtual lane, and the depths of the NI output queues are varied.

5.1 64-Processor Results

This section presents results for a 64-processor system. First, we present results for systems with uniprocessor nodes, i.e., 64 nodes. Next, we present results for systems with dual-processor nodes (32 nodes).

5.1.1 Uniprocessor Nodes

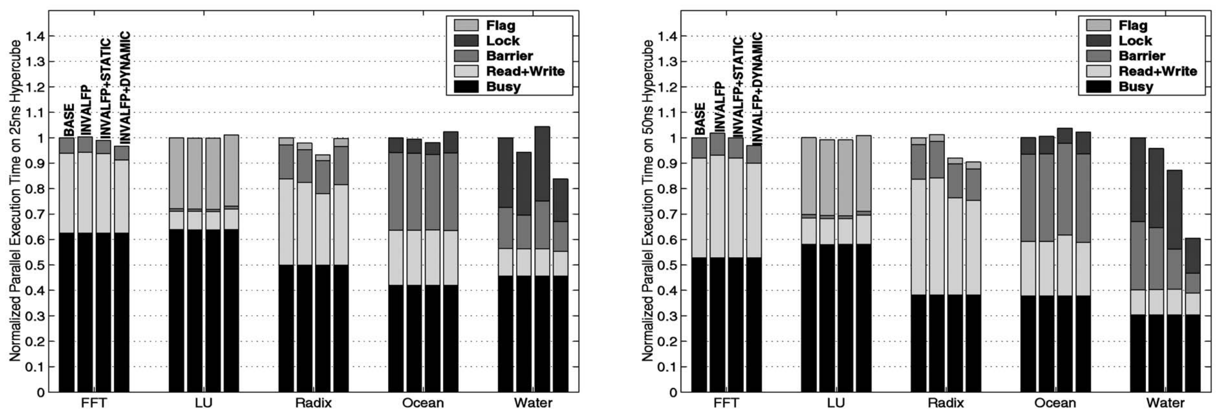
Figs. 10a and 10b present results for 64 nodes with 25ns and 50ns hop times, respectively. The execution times are normalized to Base. We divide the execution time into busy cycles, read and write stall cycles, and synchronization cycles. We further break down the synchronization time into time spent on lock acquires, barriers, and flags.

For all the applications other than Water, the algorithms perform almost equally. For Radix-Sort and Ocean, the static algorithm performs slightly better than the other algorithms. For Water, InvalFP is always better than InvalRR. For both latencies, the dynamic algorithm outperforms all others in Water. For 25ns hop time, it is 7.5 percent faster than Base, while for 50ns hop time, it is 22 percent faster. All the gains come from reduction in lock and barrier times. The lock acquire time decreases due to accelerated invalidations leading to early write completion (i.e., completion of store conditionals used to implement lock acquires), while reduction in barrier time results from improved load balance. Water is the most lock-intensive application among the five selected ones and, hence, with 64 nodes, many invalidations get generated during the lock acquire phase. In general, the relative performance of all the algorithms compared to Base improves for the slower network meaning that virtual network selection algorithms grow in importance as network latency increases.

For Radix-Sort and Ocean, the dynamic algorithm is the worst among all the algorithms. This is due to an increased read/write stall time in Radix-Sort and an increased lock acquire time in Ocean. The reasons are different for Radix-Sort and Ocean. We will explain these with the data from the 25ns hop time simulation. The reasons are the same with 50ns hop time.

In Radix-Sort, the slowdown results from a data buffer shortage in the memory controller. In the Base protocol, the dispatch unit fails to schedule a message 38.46 percent of the time due to a lack of available space in the outgoing Q1 and Q2 networks. In the dynamic algorithm, this goes down to 8.22 percent due to better utilization of the virtual networks. However, this causes more data buffers to be held in the NI output queues at the same time. This is because an outgoing message may hold the data buffer allocated to it until the cache line is injected into one of the message buffers in the corresponding virtual lane of the router. This, in turn, delays new, possibly critical, messages from being scheduled on the protocol processor. Radix-Sort is most sensitive to this problem due to its irregular bursty communication in the permutation phase. However, we will see that the benefit of the dynamic algorithm starts to outweigh this problem as the system scales to 128 nodes.

In Ocean, the increased lock acquire time results from a totally different phenomenon. Due to better load balance in the dynamic algorithm, the lock accesses from different processors are found to occur almost at the same time. As a result, the load-linked instructions (LL) are negatively acknowledged 52 percent more often compared to the Base protocol. Further, the store-conditional instructions (SC) are negatively acknowledged 36 percent more often.



(a) (b)

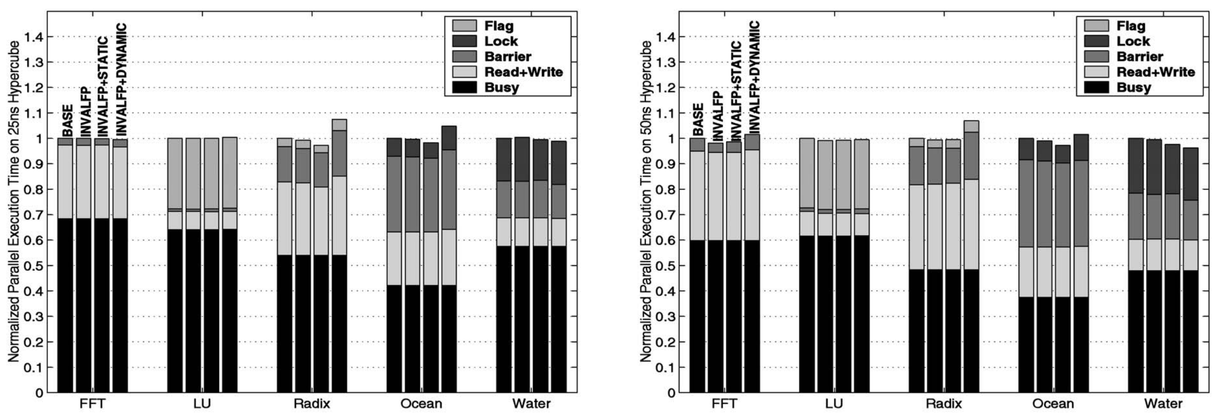
Fig. 11. Normalized execution time for 64 nodes with one message buffer per virtual lane, 16 slots in NI output queues, and (a) 25ns and (b) 50ns hop time.

This leads to an overall 44 percent increased negative acknowledgment (NACK) count in the dynamic algorithm. This is one instance where NACKs actually make a noticeable difference in the end-performance.

Next, we turn to see the effect of the number of message buffers per virtual lane in the router. This effectively changes the amount of buffering in the network. Figs. 11a and 11b show the results for one message buffer per virtual lane on 25ns and 50ns networks, respectively. The results do not include the `InvalRR` algorithm since the `InvalFP` algorithm has been found superior. Now, along with Water, FFT and Radix-Sort show some interesting variation in performance across the algorithms. For 25ns hop time the dynamic algorithm is the best for FFT executing 3 percent faster than `Base`. The static algorithm emerges the best for Radix-Sort, executing 7.5 percent faster than `Base`. For Water, the dynamic algorithm continues to be the best, executing 19 percent faster than `Base`. The static algorithm in this application increases the execution time by 4.3 percent over `Base` due to a 16.4 percent increase in NACK count. This happens mostly because of the timings of the lock accesses in this protocol. As the network slows down to 50ns, the dynamic algorithm benefits for Radix-Sort, executing 11.1 percent faster than `Base`. For Water, this algorithm is able to reduce the lock and barrier times by a surprisingly large amount and executes 63.9 percent faster than `Base`. In summary, with a smaller number of message buffers in the network virtual lanes, the contention in the NI increases and the output queue selection algorithms become even more important.

Figs. 12a and 12b present the results for four message buffers per virtual lane in the router with 25ns and 50ns hop times, respectively. Clearly, the algorithms lose significance as the message buffers get less contended. Radix-Sort and Ocean continue to see similar effects as in the two message buffers per lane case. For a 50ns hop time, the dynamic algorithm runs 4.2 percent faster than `Base` for Water.

Next, we explore the effect of varying the depths of the NI output queues. The results fix the number of message buffers per lane in the router to two. Figs. 13a and 13b present the results for eight slots (Fig. 10 had 16) in each NI output queue with 25ns and 50ns hop times, respectively. A comparison with Fig. 10 shows that reduced NI output queue size has little performance effect for LU, Radix-Sort, and Ocean. For FFT, the dynamic algorithm runs 3 to 4 percent faster than `Base` for both hop times. This



(a) (b)

Fig. 12. Normalized execution time for 64 nodes with four message buffers per virtual lane, 16 slots in NI output queues, and (a) 25ns and (b) 50ns hop time.

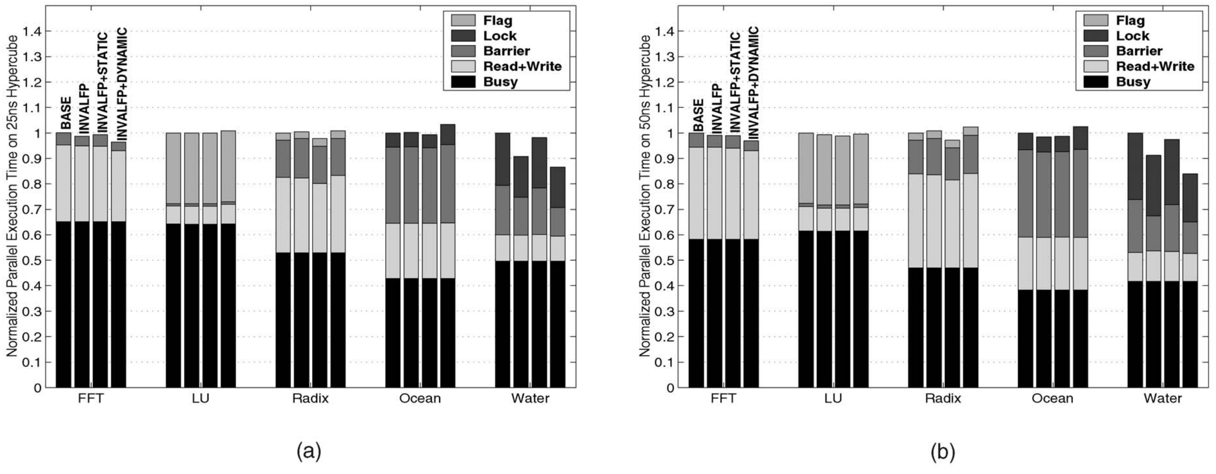


Fig. 13. Normalized execution time for 64 nodes with two message buffers per virtual lane, eight slots in NI output queues, and (a) 25ns and (b) 50ns hop time.

algorithm runs 14.9 percent and 19 percent faster than `Base` in Water for 25ns and 50ns hop times, respectively.

Figs. 14a and 14b present the results with 32 slots (as opposed to 16 in Fig. 10) in each NI output queue for 25ns and 50ns hop times, respectively. For 25ns, the algorithms perform almost equally. For Ocean, the static algorithm runs 4.2 percent faster than `Base`, while for Water, the dynamic algorithm executes 6.4 percent faster than `Base`. As the hop time increases to 50ns, Ocean attains an even larger reduction in the lock acquire time with the static algorithm. This algorithm executes 7.5 percent faster than `Base`. Water continues to observe good performance with the dynamic algorithm. It runs 19 percent faster than `Base`. Finally, we note that 32 entries in each NI output queue with 32 data buffers per node may not be a good design choice. As already mentioned, a message may hold a data buffer until it is pulled out of the NI output queue and injected into the router. This simply increases the lifetime of a data buffer preventing other possibly critical messages from being scheduled. But, implementing a large number of multiported data buffers in the memory controller is impractical.

From the above discussion, it is clear that the relative performance of the algorithms is more sensitive to the number of message buffers per virtual lane and the network latency than to the depths of NI output queues. Decreasing the number of message buffers per virtual lane increases the importance of the network selection algorithms much more compared to decreasing the depths of the NI output queues. Further, the relative performance of the algorithms always improves as the network hop time increases.

The dynamic algorithm greatly helps reduce the lock acquire time and barrier time in Water. LU remains completely insensitive to the algorithms. For FFT the static and the dynamic algorithms get some advantage in some of the cases. Ocean and Radix-Sort show strikingly similar behavior in most of the cases. For both of these applications, the static algorithm typically emerges the best, while the dynamic algorithm usually hurts performance.

5.1.2 Dual-Processor Nodes

This section discusses the results for 64-processor systems built out of dual-processor nodes. Figs. 15a and 15b present

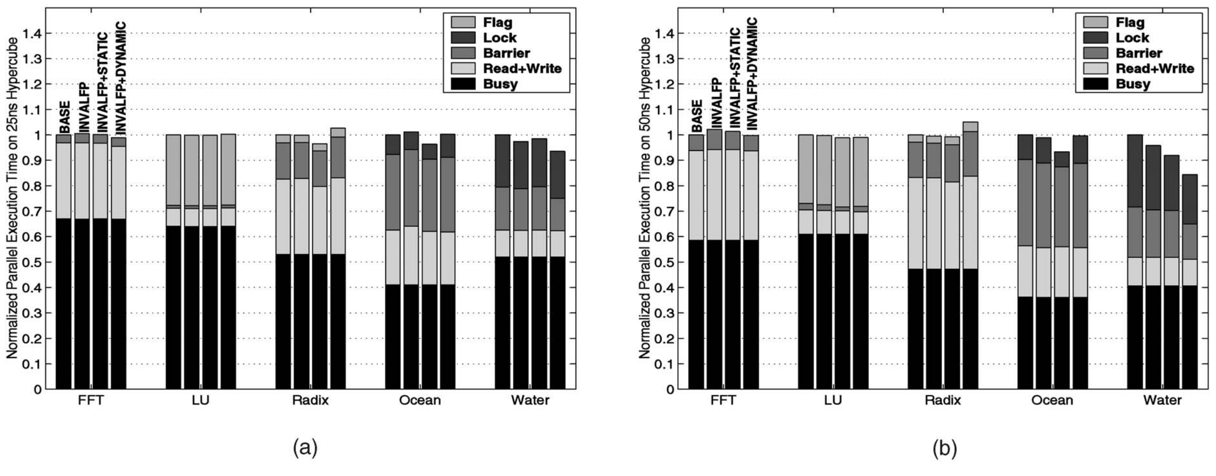
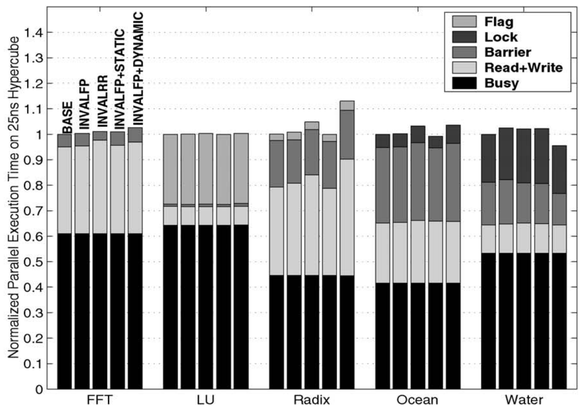
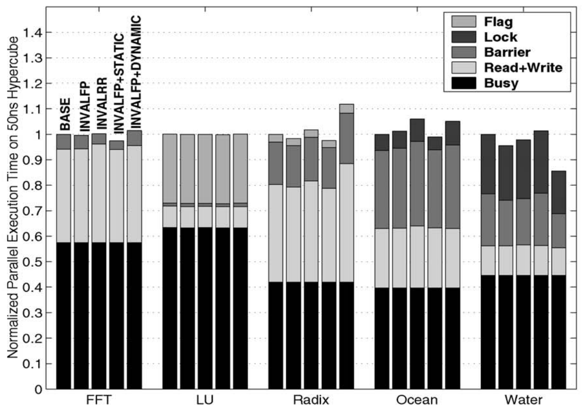


Fig. 14. Normalized execution time for 64 nodes with two message buffers per virtual lane, 32 slots in NI output queues, and (a) 25ns and (b) 50ns hop time.



(a)



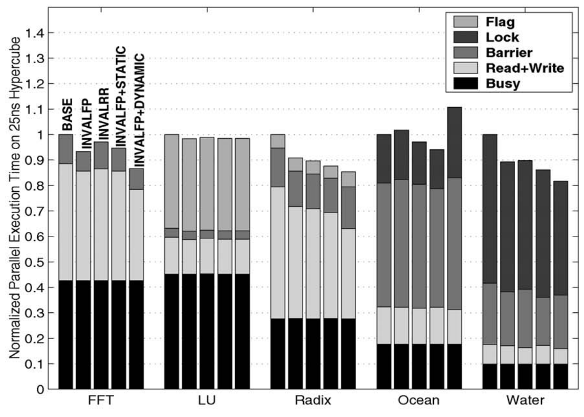
(b)

Fig. 15. Normalized execution time for 64 processors with dual-processor nodes, two message buffers per virtual lane, 16 slots in NI output queues, and (a) 25ns and (b) 50ns hop time.

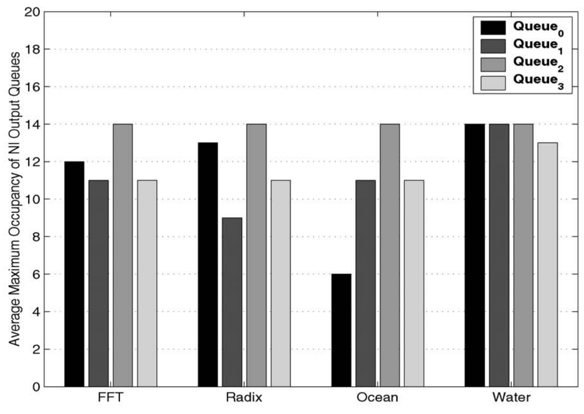
results for both 25ns and 50ns hop times fixing the number of message buffers to two and the depths of the NI output queues to 16. The `InvalRR` algorithm is worse compared to `InvalFP` in most of the cases. Water continues to benefit from the dynamic algorithm due to reduction in lock and barrier times. This algorithm is 4.2 percent and 16.3 percent faster than `Base` for 25ns and 50ns hop times, respectively. Further, some algorithms perform relatively better as the network slows down, e.g., the static algorithm in FFT, `InvalFP` and the static algorithm in Radix-Sort, and all the algorithms in Water.

5.2 Scaling to Larger Systems

This section presents the results for 128 nodes with uniprocessor nodes fixing the hop time to 25ns. Fig. 16a shows the normalized execution time while Fig. 16b plots the average maximum occupancy of the NI output queues for the dynamic algorithm in the same way as Fig. 4b does for the `Base` protocol. Both the results use two message buffers per virtual lane and 16 entries in each NI output queue. We have omitted the results for LU in the occupancy plot since this application is mostly insensitive to the algorithms. In all the applications, other than Radix-Sort and Ocean, `InvalFP` performs better than `InvalRR`. The trends are now quite different for FFT and Radix-Sort. The dynamic algorithm runs 14.9 percent faster than the `Base` protocol for FFT. This is due to reduced read/write stall time. For Radix-Sort, the algorithms perform progressively better. `InvalFP` executes 9.9 percent faster and `InvalRR` runs 11.1 percent faster compared to `Base`. The static algorithm executes 13.6 percent faster, while the dynamic algorithm speeds up execution by 17.6 percent over `Base` for Radix-Sort. In this application, all the gains arise from reduced read/write stall time. Also, LU executes 2 percent faster than `Base` for all the algorithms other than `InvalRR`. For Ocean, the static algorithm continues to excel, executing 6.4 percent faster than `Base`, while the dynamic algorithm continues to suffer from a 48 percent increase in NACK count over `Base`. Water continues to show similar trends as in 64 nodes. `InvalRR` and `InvalFP` are, respectively, 11.1 percent and 12.4 percent faster than `Base`. The static and the dynamic algorithms execute, respectively, 16.3 percent and 22 percent faster than `Base`. Again, most of

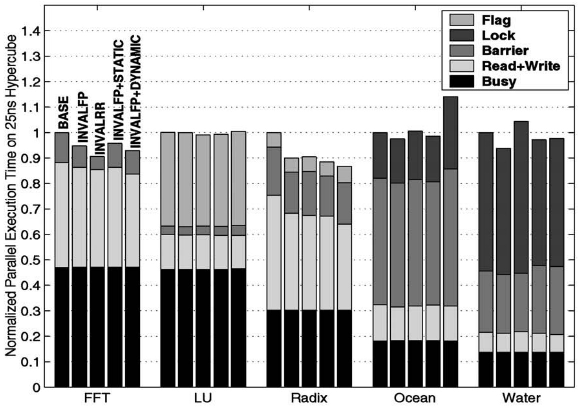


(a)

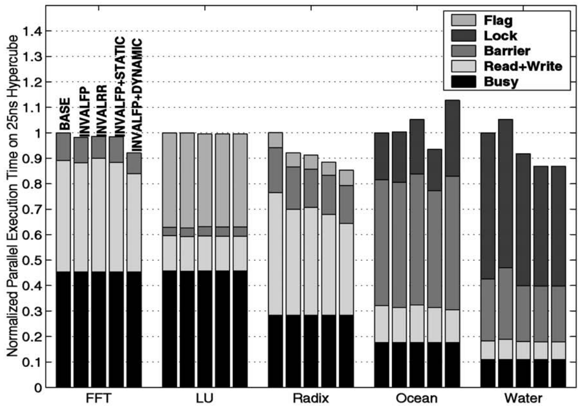


(b)

Fig. 16. (a) Normalized execution time, (b) average maximum occupancy of NI output queues in the dynamic algorithm for 128 nodes, two message buffers per virtual lane, 16 slots in NI output queues, and 25ns hop time.



(a)



(b)

Fig. 17. Normalized execution time for 128 nodes with 25ns hop time. (a) Four message buffers per virtual lane and (b) 32 entries in each NI output queue.

the gains come from greatly reduced lock acquire time made possible by accelerated invalidations leading to early completion of store conditionals. In general, there is more variation in performance across the algorithms for 128 nodes as compared to 64 nodes. The results establish the fact that the virtual network selection algorithms become more important as we move toward large-scale DSM multiprocessors.

Turning to Fig. 16b, we observe that the dynamic algorithm (the best performing algorithm for FFT, Radix-Sort, and Water) achieves a very good balance in the NI output queues for Water. Although, for FFT, Radix-Sort and Ocean the Q2 network is still overloaded due to the fixed priority invalidation algorithm, the utilization of the other three networks is good, especially for FFT and Radix-Sort. For Ocean the Q0 network still looks slightly underutilized. As we have already pointed out, it is impossible to execute a pure round robin algorithm because the coherence protocol must remain deadlock-free. Still, our dynamic algorithm achieves a much better balance of load among the NI output queues compared to the baseline static algorithm shown in Fig. 4b.

Next, we investigate the effects of reducing contention in the NI. We separately present the results with four message buffers per virtual lane and 32 slots in the NI output queues. Fig. 17a shows the results with four message buffers (as opposed to two in Fig. 16a) per virtual lane, while keeping the depths of the NI output queues fixed at 16. A comparison with Fig. 16a brings out some interesting differences. For FFT, the relative benefit of the dynamic algorithm goes down slightly while the `InvalRR` algorithm emerges the best, executing 9.9 percent faster than `Base`. Most of the gains achieved by `InvalRR` arise from a reduced barrier time resulting from better load balance. LU continues to remain insensitive to the algorithms. The relative performance of the algorithms is largely unchanged for Radix-Sort with a slight decrease in the gain achieved by the dynamic algorithm, executing 14.9 percent faster than `Base`. For Ocean, all the algorithms other than the dynamic one perform almost equally. For Water, the relative importance of the algorithms diminishes as the message buffers become less contended. For this application, the `InvalFP` algorithm emerges the best, executing 6.4 percent faster than `Base`. In summary, as we observed for 64 nodes, increasing the number of message buffers usually reduces the performance gap between the various virtual network selection algorithms.

Fig. 17b presents the results with 32 entries (as opposed to 16 in Fig. 16a) in each NI output queue, while fixing the number of message buffers per lane to two. Again, comparing these results against the ones presented in Fig. 16a, we find similar trends. For FFT, the performance gap between the algorithms is reduced, but still, the dynamic algorithm continues to deliver the best performance, executing 8.7 percent faster compared to `Base`. For Radix-Sort, the results are essentially the same as in Fig. 16a. For Ocean, the static algorithm continues to be the best running 6.4 percent faster compared to `Base`. For Water, the performance gap between the algorithms diminishes; the dynamic algorithm runs 14.9 percent faster than `Base`. In summary, the general observations remain unchanged as the system scales. The performance gap between the virtual network selection algorithms is more affected by the number of message buffers per lane in the router than the depths of the NI output queues in the memory controller.

6 Conclusions

We have presented an evaluation of four virtual network selection algorithms along with a conventional base algorithm for five shared-memory scientific applications running on 64 and 128-node DSM multiprocessors. The evaluated algorithms include one fixed priority and one round robin algorithm to distribute the load of invalidation messages among different virtual networks. One static algorithm aims at balancing the reply message load, while a completely dynamic algorithm picks the outgoing network for all types of messages based on a round robin policy. Our results show that the low overhead fixed priority algorithm appears to be sufficient for balancing invalidation load. In 64-node systems, except for one application (Water), the performance gap between the algorithms is not significant. However, as the system scales to support 128 nodes, the virtual network selection algorithms become important. Except for LU, all applications show different performance levels for different algorithms. In most cases, the dynamic algorithm emerges the best, speeding up parallel execution by as much as 22 percent over an optimized base system. But, there are applications (e.g., Ocean) for which the static algorithm consistently outperforms the dynamic one.

With the dynamic algorithm, Ocean suffers from an increased NACK count. To further explore this issue, we designed a number of NACK-free cache coherence protocols [3] and augmented the best one with our `InvalFP`, a static and a dynamic virtual network selection algorithm. Our evaluation on Ocean with a 25ns hop time and 64 nodes shows that, after eliminating the NACKs, virtual network selection algorithms continue to improve the end-performance of Ocean, with the static algorithm slightly outperforming the dynamic one.

To explore the performance sensitivity of the algorithms, we varied the number of message buffers per virtual lane, the depths of the NI output queues, and the network hop time. In general, the performance gap between the algorithms is more sensitive to the number of message buffers in the network routers than the NI output queue depth in the node controller. Even with four message buffers per virtual lane, the algorithms exhibit a wide range of performance levels for some of the applications (Radix-Sort) running on 128 nodes. Although this study has fixed the number of virtual networks to four, having more virtual networks will lead to better distribution of load and increased freedom of choice for the dynamic algorithm. Finally, with increasing network hop time, the relative performance gap between the algorithms increases, implying that having a good virtual network selection algorithm has significant benefits for slow or contended interconnection networks.

ACKNOWLEDGMENTS

This research was supported by National Science Foundation CAREER Award CCR-0340600.

REFERENCES

[1] L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 282-293, June 2000.
[2] M. Chaudhuri et al., "Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation," *IEEE Trans. Computers,* vol. 52, no. 7, pp. 862-880, July 2003.
[3] M. Chaudhuri and M. Heinrich, "The Impact of Negative Acknowledgments in Shared Memory Scientific Applications," *IEEE Trans. Parallel and Distributed Systems,* vol. 15, no. 2, pp. 134-150, Feb. 2004.
[4] W.J. Dally, "Virtual-Channel Flow Control," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 2, pp. 194-205, Feb. 1992.
[5] W.J. Dally and C. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers,* vol. 36, no. 5, pp. 547-553, May 1987.
[6] M. Galles, "Spider: A High-Speed Network Interconnect," *IEEE Micro,* vol. 17, no. 1, pp. 34-39, Jan.-Feb. 1997.
[7] K. Gharachorloo et al., "Architecture and Design of AlphaServer GS320," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 13-24, Nov. 2000.
[8] J. Gibson et al., "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 49-58, Nov. 2000.
[9] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. 1990 Int'l Conf. Parallel Processing,* pp. 312-321, Aug. 1990.
[10] M. Heinrich et al., "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 274-285, Oct. 1994.
[11] M. Heinrich, "The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," PhD Dissertation, Stanford Univ., Oct. 1998.
[12] M. Heinrich et al., "A Quantitatitve Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 205-217, (special issue on cache memory and related problems), Feb. 1999.
[13] M. Heinrich and M. Chaudhuri, "Ocean Warning: Avoid Drowning," *ACM SIGARCH Computer Architecture News,* vol. 31, no. 3, pp. 30-32, June 2003.
[14] A. Kumar and L.N. Bhuyan, "Evaluating Virtual Channels for Cache-Coherent Shared-Memory Multiprocessors," *Proc. 10th ACM Int'l Conf. Supercomputing,* pp. 253-260, May 1996.
[15] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.
[16] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 241-251, June 1997.
[17] D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 148-159, May 1990.
[18] D. Lenoski et al., "The Stanford DASH Multiprocessor," *Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.
[19] T.D. Lovett and R.M. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Int'l Symp. Computer Architecture,* pp. 308-317, May 1996.
[20] J.F. Martínez, J. Torrellas, and J. Duato, "Improving the Performance of Bristled CC-NUMA Systems Using Virtual Channels and Adaptivity," *Proc. 13th ACM Int'l Conf. Supercomputing,* pp. 202-209, June 1999.
[21] S.S. Mukherjee et al., "The Alpha 21364 Network Architecture," *IEEE Micro,* vol. 22, no. 1, pp. 26-35, Jan. 2002.
[22] S.S. Mukherjee et al., "A Comparative Study of Arbitration Algorithms for the Alpha 21364 Pipelined Router," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 223-234, Oct. 2002.
[23] A. Nowatzyk et al., "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. 24th Int'l Conf. Parallel Processing,* pp. 1-10, Aug. 1995.
[24] S.L. Scott and G.M. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus," *Proc. Conf. Hot Interconnects 4,* Aug. 1996.
[25] A.S. Vaidya, A. Sivasubramaniam, and C.R. Das, "Performance Benefits of Virtual Channels and Adaptive Routing: An Application-Driven Study," *Proc. 11th ACM Int'l Conf. Supercomputing,* pp. 140-147, July 1997.
[26] S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

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

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