Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact *

Rajeev Sivaram§ Craig B. Stunkel† Dhabaleswar K. Panda‡

§IBM Power Parallel Systems
522, South Road, MS P963
Poughkeepsie, NY 12601
rsivaram@us.ibm.com

†IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
stunkel@watson.ibm.com

‡Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
panda@cis.ohio-state.edu

Abstract

Multidestination message passing has been proposed as an attractive mechanism for efficiently implementing multicast and other collective operations on direct networks. However, applying this mechanism to switch-based parallel systems is non-trivial. In this paper we propose alternative switch architectures with differing buffer organizations to implement multidestination worms on switch-based parallel systems. First, we discuss issues related to such implementation (deadlock-freedom, replication mechanisms, header encoding, and routing). Next, we demonstrate how an existing central-buffer-based switch architecture supporting unicast message passing can be enhanced to accommodate multidestination message passing. Similarly, implementing multidestination worms on an input-buffer-based switch architecture is discussed, and two architectural alternatives are presented that reduce the wiring complexity in a practical switch implementation. The central-buffer-based and input-buffer-based implementations are evaluated against each other as well as against the corresponding software-based schemes. Simulation experiments under a range of traffic (multiple multicast, bimodal, varying degree of multicast, and message length) and system size are used for evaluation. The study demonstrates the superiority of the central-buffer-based switch architecture. It also indicates that under bimodal traffic the central-buffer-based hardware multicast implementation affects background unicast traffic less adversely compared to a software-based multicast implementation. These results show that multidestination message passing can be applied easily and effectively to switch-based parallel systems to deliver good multicast and collective communication performance.

*This research is supported in part by NSF Career Award MIP-9502294, NSF Grant CCR-9704512, and an IBM Cooperative Fellowship. A preliminary version of this paper has been presented at the 24th *Annual International Symposium on Computer Architecture (ISCA-24)* [43], June 1997. This research was performed when Rajeev Sivaram was a graduate student at The Ohio State University.

Contents

1 Introduction

The wormhole-routing switching technique is the current trend in building parallel systems due to inherent advantages such as low-latency communication and reduced communication hardware overhead [9, 23, 10]. This switching technique is being used with a wide variety of topologies. Examples include k-ary n-cube networks (Cray T3D [7], Cray T3E, Intel Paragon [12], Ncube [11]), fat-tree networks (CM-5 [17], Meiko CS2 [3]), and bidirectional multistage interconnection networks (IBM SP1/SP2[41, 42]). All of these systems are being used to support either the distributed-memory or distributed-shared memory programming paradigms. For efficient support of either paradigm, these systems require fast collective communication [4, 21] support, as defined by the Message Passing Interface (MPI) Standard, from the underlying communication subsystem. Among the set of collective communication operations, *broadcast* and *multicast* are fundamental and they are used in several other operations like barrier synchronization and reduction [26]. Thus, reducing the latency of broadcast and multicast operations on these systems is vital for achieving high performance parallel computation.

Many software schemes have been recently proposed in the literature to efficiently implement broadcast and multicast in wormhole-routed k-ary n-cube networks [2, 22], multistage interconnection networks [46], and irregular networks [15]. All of these schemes use point-to-point (*unicast*) message passing and require multiple *contention-free* phases to achieve fast broadcast and multicast. For multicasting to m destinations, these schemes typically require $\lceil \log_2(m+1) \rceil$ communication phases. Since the ratio of communication start-up time to propagation time is quite high on current parallel systems [7, 12, 41], such a software-based unicast message-passing approach leads to very high latency for broadcast and multicast operations. In addition, under hybrid unicast and multicast traffic, the software-based approach leads to low throughput from the network. This is due to increased network contention because of the multi-phase implementation of the multicast. Such performance degradation has forced researchers and designers of parallel systems to investigate hardware support for multicast [24].

A new concept, *multidestination* wormhole message passing, has been introduced recently [19, 28] for efficient implementation of collective communication operations using a fewer number of communication phases. Unlike a unicast message which has a single destination, such a mechanism allows a message to have multiple destinations. A multidestination worm provides the flexibility to distribute data to multiple nodes or to gather data from multiple nodes using a single message and a single communication start-up. These worms get routed on the data network together with the unicast messages. Using such worms, it has been shown that broadcast and multicast operations on k-ary n-cube networks with different routing schemes (e-cube, adaptive, Hamiltonian, etc.) can be implemented with significantly reduced latency [19, 28] compared to unicast-based schemes. This mechanism and the associated schemes were specifically developed for direct networks with k-ary n-cube topologies in mind and cannot be easily extended to deliver high-performance multicast in switch-based systems.

In recent work [36, 38], we have extended the multidestination message passing concept to multistage interconnection networks. This extension involves a new concept called *multiport encoding* and an asynchronous replication mechanism at the input buffers of a switch to implement deadlock-free multicast. Using this new encoding, a set of algorithms for multicast/broadcast have been developed and evaluated. In this work, it has been shown that the latency of a single multicast/broadcast operation can be reduced by up to a factor of 4 compared to the software-based scheme using unicast messages.

The results in [36] show the proof-of-concept of multidestination message passing in MINs. However, the paper does not explore other architectural alternatives and their impact. For example, the IBM SP2 [42] uses a central-buffer-based buffer organization in a switch for unicast message passing, and this potentially provides a powerful basis for worm replication. Similarly, the multiport-based encoding in [36] often requires a multicast to be implemented in multiple phases. Even though this approach requires fewer phases than the unicast-based software scheme, a challenging problem is to implement multicast in even fewer phases using better encoding schemes.

In this paper, we address these challenges and consider architectural alternatives and their impact on the implementation of multidestination worms for multicast/broadcast on switch-based parallel systems. We consider a *bit-string encoding* scheme to implement multidestination worms. This encoding allows a multicast/broadcast to be implemented using a single communication phase. Two alternative switch architectures, central-buffer-based and input-buffer-based, are considered. Detailed designs for implementing multidestination worms on these two switch architectures are derived, and methods for reducing the complexity of practical switch implementations are presented. It is shown that multidestination worms can be implemented on both switch architectures with little additional hardware. These designs also provide realistic timing parameters for routing and propagation of multidestination worms.

Next, the implementations on both switch architectures are evaluated under different traffic scenarios. Two performance measures (latency vs. throughput and received vs. applied load) are studied for multiple multicast traffic as well as hybrid traffic consisting of both multicasts and unicasts. The hardware-based implementations are also compared with the software-based multicast schemes using unicast messages. The results demonstrate the benefits of the hardware-based multicast schemes compared to the software-based schemes. Furthermore, with equal amounts of buffer space in a switch, the central-buffer-based implementation shows an improvement of up to a factor of 2 (in terms of saturation load) compared to the input-buffer-based implementation. These results clearly demonstrate the superiority of the central-buffer-based scheme to support multidestination worms for fast multicast/broadcast on switch-based parallel systems.

The remaining paper part of this paper is organized as follows. Section 2 provides an overview of switch-based parallel systems. Issues related to implementing multidestination worms are presented in Section 3. Detailed designs to implement multidestination worms on central-buffer-based and input-buffer-based switch architectures are presented in Sections 4 and 5, respectively. A comparison of these two implementations is carried out in Section 6 and Section 7 shows performance evaluation results. Related work is reviewed in Section 8 and our conclusions are presented in Section 9.

2 Switch-Based Parallel Systems

In this section we briefly describe the various categories of switch-based parallel systems and define the scope of this work. Switch-based parallel systems fall into two broad classes: (i) systems based on regular interconnects and (ii) systems based on irregular interconnects.

Communication and synchronization issues on parallel systems based on regular interconnects have been studied quite extensively. Architectures using unidirectional multistage interconnection networks (MINs) and bidirectional MINs (bidi-MINs) or fat-tree networks [17, 31] fall into this category. In such networks, the regular pattern of interconnection between switches allows for simple routing functions to be defined for routing messages between nodes. In unidirectional MINs, messages must traverse all switch stages to reach the destination processors. In bidirectional MINs on the other hand, *turnaround* routing can be used, which rewards local communication. Figures 1(a) and (b) show examples of unidirectional and bidirectional MIN networks respectively.

Another class of switch-based parallel systems that is becoming popular is a network/cluster of workstations (NOW/COW). These systems typically consist of workstations interconnected with switches in irregular topologies. Deadlock free routing of messages in such networks is a challenging problem [30, 32]. Figure 1(c) shows an example of an irregular network of workstations.

The schemes and designs proposed in this paper are applicable to all categories of switch-based parallel systems. However, for performance comparisons we restrict ourselves to bidirectional MIN topologies. In the next section, we introduce the concept of multidestination worms in the context of switch-based parallel systems and discuss some of the issues to be considered in implementing them.

3 Issues in Implementing Multidestination Worms

The path-based multicast proposed for direct networks [19, 28] has been shown to lead to deadlock in unidirectional MINs [6]. Even without this constraint, path-based multicast is highly inefficient because the network has to be traversed multiple times, and flits of the message have to be copied and forwarded by the network interface associated with the nodes.

An alternative method for implementing multidestination worms is to augment switches with replication capability. A switch replicates flits of an arriving multidestination worm to one or more of its output ports. In strict or pure wormhole routing, switches buffer only a single flit of an arriving worm and cannot accept the next flit until this buffer is freed. For a multidestination

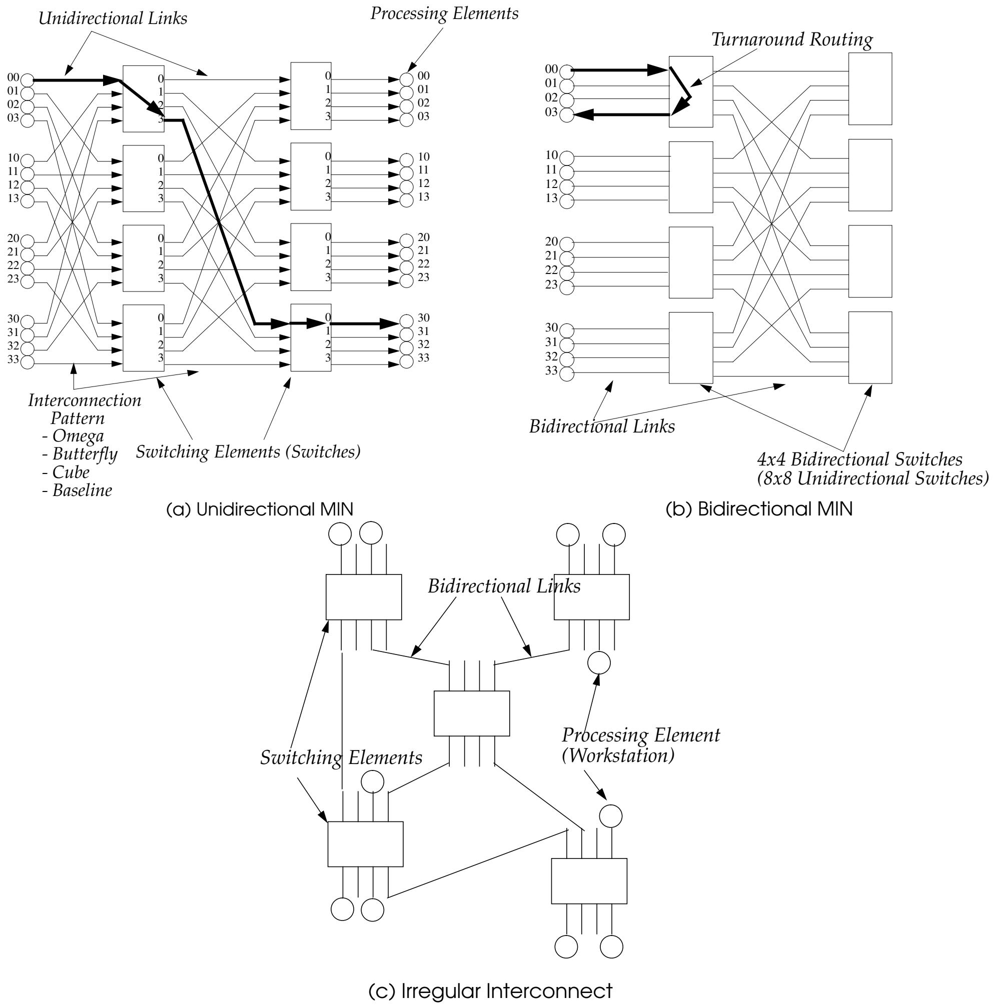

Figure 1: Examples of switch-based parallel systems.

worm that has to be forwarded to multiple output ports, this buffer cannot be freed until the switch can forward the flit to all (required) output ports. Thus deadlock can easily occur if two multidestination packets that must be forwarded to the same set of output ports arrive at a switch with one of the worms reserving a subset of the output ports and the other worm reserving the rest. This deadlock problem exists even if the switch can buffer more than one flit.

Figure 2 presents two examples of such deadlock scenarios. In the first example, deadlock occurs between two multidestination worms at a single switch because each worm reserves a subset of the output ports required by the other. If the buffers are smaller than the size of the packet, the buffers cannot be freed because the data in the buffer has not been forwarded to all the required ports. Furthermore, the ports cannot be released because only a part of the packet has been transmitted to them. Even if we can perform arbitration so that such a situation never occurs at a single switch (by, say, using some form of prioritized reservation of ports), deadlock can still occur over multiple switches as shown in the second example of Fig. 2. In this example, the numbers identify the flit numbers within the corresponding packets. Two multidestination worms entering at the left are each destined for the four output ports at the right. The upper multidestination worm in the figure reserves the output ports of the top right switch, whereas the bottom multidestination worm reserves the output ports of the bottom right switch. If the buffers at the switches are smaller than the packet size, deadlock can occur because the input ports of the right hand switches fill up with flits from each of the multidestination worms, and once again, neither the buffer space nor the output ports can be freed.

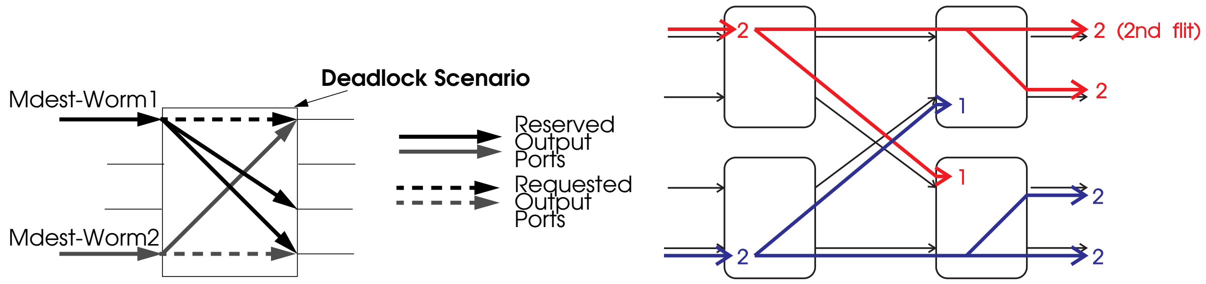

Figure 2: Typical cases of deadlock for multidestination worms that use replication at the switches.

In the next section we describe two replication mechanisms that have been proposed in the literature and the methods that they adopt to overcome this deadlock problem.

3.1 Deadlock-free Replication

Two replication methods have been proposed in the literature: *synchronous* and *asynchronous*. *Synchronous* replication requires that flits of a multidestination worm proceed in lock-step. Thus any *branch* of the multidestination worm that is blocked can block all other branches. This replication mechanism was originally proposed in the context of pure wormhole routing and is susceptible to deadlock. Furthermore it requires some kind of feedback architecture to ensure that the flits proceed in lock-step [6].

Asynchronous replication [6, 29] is an alternative mechanism that allows flits of a multidestination worm to be forwarded to the subset of requested output ports that the worm successfully reserves. If a switch is equipped with a buffer of size f flits, under asynchronous replication at most f flits can be forwarded to the output ports that have been reserved by the multidestination worm before the worm is blocked because of the required output ports which it could not reserve. This is because none of the f flits in the buffer can be freed as they haven't been forwarded to all of the required output ports, and more flits cannot enter the full buffer. *Bubbles* are therefore introduced if the remaining requested output ports cannot be obtained before the input buffer fills up. Even this method of replication is susceptible to deadlock.

To prevent deadlock under synchronous replication, deadlock avoidance schemes have been proposed that arbitrate between multidestination packets at a switch to prevent cyclic wait [6]. To prevent deadlock under asynchronous replication, switches must be equipped with buffers large enough to store the largest packet in the system. Deadlock is prevented if the switches can guarantee that a packet accepted for transmission can be *eventually* completely buffered at the switch [36], a requirement that is weaker than virtual cut-through [14].

Asynchronous replication may be preferred for a practical implementation due to the following reasons. Firstly, it does not require the costly feedback architecture required under synchronous replication. Secondly, it is more efficient because blocked branches don't block other branches (in fact, there is no dependence across branches). Finally, many current day switches already provide for relatively large buffers at the switches [3, 42] making the satisfaction of the deadlock-freedom requirement under asynchronous replication easier.

In this paper we present two alternative implementations of multidestination worms with asynchronous replication. The two implementations differ primarily in the way they guarantee that an arriving multidestination worm can eventually be completely buffered. One of the implementations uses a central-buffer-based switch architecture. This architecture uses an output queuing technique similar to the ones used in switches of the IBM SP2 [42] for unicast message passing. The other implementation uses an input-buffer-based switch architecture. Detailed designs to implement multidestination worms on these two switch architectures are discussed in Sections 4 and 5 respectively. In the remaining part of this section we discuss two additional issues related to implementing multidestination worms: header encoding and routing.

3.2 Header Encoding

A variety of header encoding mechanisms for routing multidestination headers have been proposed in the literature. These schemes differ in the following characteristics: size of the header required for encoding, knowledge required at the switches about the system topology, complexity and speed of the decoding logic required at the switches, and the multicast sets that can be covered using these schemes [5, 36]. It must be emphasized that the encoding and decoding mechanism used for routing a multidestination worm is independent of the scheme adopted for replicating a multidestination worm at a switch.

In [36] we have proposed a multiport encoding scheme for multidestination worms. This encoding mechanism allows extremely simple decoding logic at the switches. In addition, switches are not required to know the topology of the MIN network. Depending on the network topology, the source node encodes the header of a multidestination worm based on the path that needs to be followed to reach a set of destinations. However, all arbitrary multicast destination sets cannot be encoded in a single worm using multiport encoding. Instead, given an arbitrary multicast destination set, a set of multiport encoded worms are required to cover the destinations. A multi-phase multinomial tree based approach is adopted to perform multicast using this set of multiport encoded worms. Even though this scheme performs better than software-based multicast using unicast message passing, a multi-phase implementation may not be desirable for system level multicasts such as cache invalidation traffic in distributed shared memory systems. This is because with cache invalidation only the source node has information about the destination set [8]. Thus, a single phase implementation may be preferable in such systems, perhaps at a slight additional decoding logic cost.

One form of header encoding that accomplishes multicast to arbitrary multicast destination sets in a single communication phase is *bit-string encoding* [5, 29]. The encoding consists of N bits where N is the system size, with a '1' bit in the ith bit position indicating that processor i is a multicast destination.

To decode a bit-string encoded header a switch must possess knowledge about the processors that can be reached through each of its output ports. This *reachability information* [29] can be encoded using a similar N bit string for each output port with '1' bits denoting processors that are reachable via the output port. Such an N-bit string is associated with each output port. For some unidirectional MINs, the space required to store this information can be reduced by taking into account the fact that a processor may be reached from only one of the switch's output ports because of the unique path property possessed by some of these networks. The reachability information can be set up at the switches at the time of system startup or reconfiguration. Figure 3 shows a bidirectional MIN network along with the reachability information associated with some of the output ports. The figure also shows a bit string encoded header and the path followed by the corresponding multidestination worms.

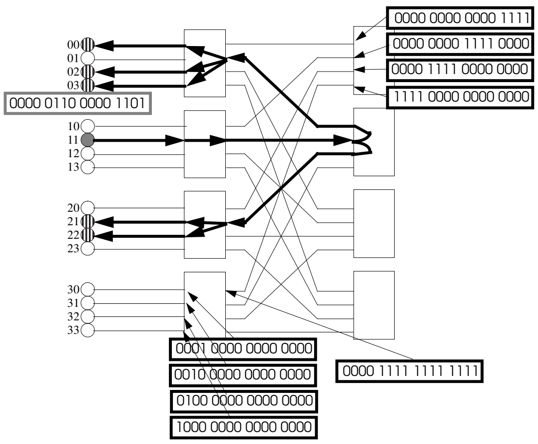

Figure 3: An example of a bidirectional MIN network with 16 processors, bit-string encoded header with the path of the corresponding multidestination worm, and the reachability strings associated with some of the output ports.

When a multidestination worm with a bit-string encoded header arrives at a switch, the switch compares the bit-string in the header with the reachability information associated with every one of its output ports. A bit-wise AND operation on the strings can be used to determine if the strings have one or more '1' bits at common positions, indicating that the multidestination worm should be forwarded to the corresponding output port. The logic required to compare these bit-strings is simple and the output ports required by the worm can thus be determined quite easily.

3.3 Routing

Although the bit-string encoding presented in the previous section allows a multidestination worm to be replicated along a fixed path from a source to its destinations, the presence of alternative paths between nodes in some switch-based networks may not be adequately exploited using the bit-string encoding alone. In networks like the bidi-MINs, there exists considerable choice in the way a multidestination message can be routed. A multidestination message could replicate downwards on its forward path while going to the *least common ancestor* (LCA) [31] stage of the source and destinations and then cover the remaining destinations by replicating on the way back from the LCA stage[1].

Alternatively, a multidestination worm could just travel to the LCA stage and then cover all the destinations by replicating on the way back [29]. Since the same set of destinations can be reached from any switch in the LCA stage, choice exists in both methods for determining the path taken upto the LCA stage—one can decide to deterministically route messages to the LCA stage or to make the choice adaptively.

In this paper, we assume that multidestination worms travel adaptively up to the LCA stage and that the destinations are covered by replicating on the downward path from the LCA stage. In addition to the bit-string encoded header, the worm may carry a count field which is initialized to the LCA stage. The count is decremented at each switch on the worm's forward path and the worm begins its backward journey when the count value becomes zero. It is to be noted that such choices of path do not exist for unidirectional MINs with the unique path property.

For irregular topologies, routing can be performed much like the routing described for bidi-MINs by assuming a tree structure superimposed on the irregular network. Such tree structures are typically used in irregular networks to provide deadlock-free routing of unicast messages. For example, in the up*/down* [32] routing algorithm, a tree structure is imposed on the networks and links are uniquely identified as 'up' links or 'down' links depending on whether they take you closer or farther away from the root. Such a tree can be used to determine reachability information for the 'down' links of the switches. Multidestination routing can then be performed by adaptively routing a message to a LCA switch by following 'up' links and then replicating the message on the 'down' path at the intermediate switches using the reachability information associated with each of the switches' 'down' links [37].

In this section, we have described some of the issues with respect to implementing multidestination worms in switch-based architectures. In the next two sections we discuss how asynchronous replication of multidestination worms can be implemented on two alternative switch architectures: central-buffer-based (output-queue-based) and input-buffer-based.

[1]Such a method allows replication to proceed in both directions within a switch. This introduces deadlock problems because of multidestination packets traveling in one direction using up the entire buffer space at a switch that uses a shared central buffer for replication. Such deadlock can be avoided by ensuring that at all times there exists at least one packet's space in the buffer for multidestination packets traveling in either direction.

4 Central Buffer (Output Queue) based Switch Architecture

We now describe a switch architecture with output queuing modified for implementing multidestination worms. We first describe an output queue-based switch architecture very similar to the high performance switch in the IBM SP2 [42]. We then present modifications that could be made to this architecture to implement multidestination worms.

4.1 Buffered Wormhole Routing

The *buffered wormhole routing* used in the IBM SP2 is a variation of wormhole routing wherein every switch in the network is equipped with a central buffer and a crossbar, as illustrated in Figure 4. When a packet arrives at an input port and encounters no contention for the chosen output

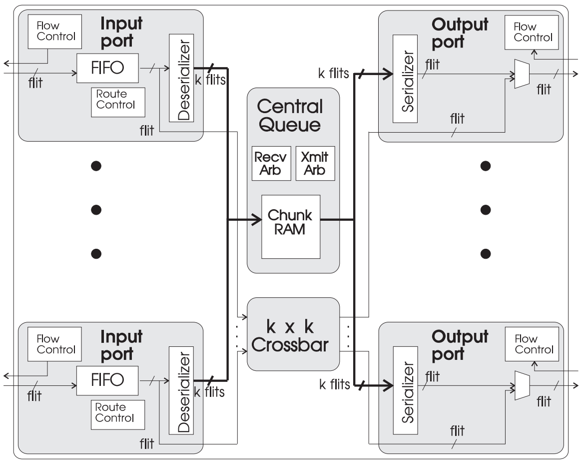

Figure 4: A switch equipped with a central buffer

port, the packet uses the crossbar path for minimal delay. However, when packets are blocked at an input port due to a busy output port, the switch attempts to store the packet in this central buffer, thus freeing the links held by the trailing packet flits. There may be enough space in the central buffer to store the entire packet. However, there is *no guarantee* that a packet arriving at a switch will find enough space in the central buffer to be completely stored. If the central buffer does not have adequate space to store the entire blocked packet, as many as possible of the packet flits are stored in the central buffer and the remainder of the packet is blocked in place. Note that in the absence of contention, packets may propagate through the network just as in a purely wormhole routed network, and the central buffers will remain empty.

An SP2-like central buffer is an extremely attractive resource for packet replication because multiple output ports can retrieve the identical packet flits from the same shared buffer. However, because there is no assurance that this central buffer can store an entire multidestination packet, the central buffer as described cannot guarantee to prevent multicast deadlock. To address this problem, we will describe minor modifications to the basic central buffer free-space logic that are similar to virtual cut-through operation. Specifically, these changes guarantee that any packet admitted to the central buffer can (eventually) be entirely stored. This guarantee effectively decouples the interdependence of the replicated output packets at a switch, eliminating the cause of multicast wormhole routing deadlock.

In the SP2 buffered wormhole implementation, the central buffer effectively forms a separate FIFO queue of packets for each output port. Each input port can write flits into the buffer, and each output port can read flits. The central buffer space is dynamically allocated to requesting input ports on a least-recently-served basis.

For a k-port switch, k flits are buffered (deserialized) by an input port into a k-flit *chunk* before being written into the central buffer. These k-flit chunks are read from the central buffer and are disassembled (serialized) into k flits again by the intended output port. This reduces the number of central buffer RAM read and write ports required. As an example, in the 8-port SP2 switches, up to 1 flit is received (transmitted) at each of the 8 input (output) ports every cycle. An SP2 chunk is therefore 8 flits, and the central buffer only requires 1 RAM write port and 1 RAM read port to match the input and output bandwidth of the switch. In general, for a switch with k input and k output ports, a chunk size of k flits is required for a 2-port RAM to match the input and output bandwidth of the switch.

The central buffer logic maintains a list of free chunk locations. A central buffer write allocates a free chunk, and a read returns a free chunk. There must be a mechanism—the next-packet list—to order the packets within each packet queue. Each packet is divided into chunks, and thus there is also a mechanism—the next-chunk list—to order the chunks within a packet. To record these two types of linking information, two pointer fields are associated with each chunk of data: the next-packet (NP) field and the next-chunk (NC) field (see Figure 5). In addition, each output port

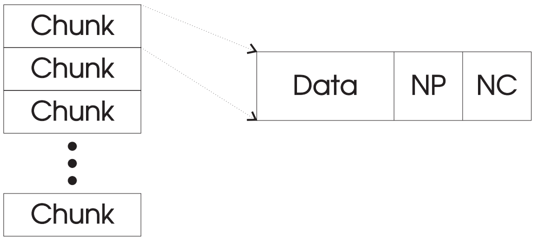

Figure 5: Organization of a central buffer

o maintains first-packet ($firstP[o]$) and last-packet ($lastP[o]$) pointers into its packet queue, and a first-chunk ($firstC[o]$) field that points to the next chunk to be read if output port o has already read the header chunk of the current packet. Each input port i maintains a last-chunk ($lastC[i]$) field that points to the last chunk written by input port i. All pointers are assumed to be *nil* when invalid.

In the following discussion, we shall assume input port i is writing chunks to be read by output port o. The next-packet list is updated each time the first chunk (the *header chunk*) of a packet is written. The updation of the list is done as follows. If no packets are currently on the destination output port's packet queue ($firstP[o] \equiv nil$), then $firstP[o] \leftarrow writeloc$, where $writeloc$ is the address where the header is written. Otherwise, $NP[lastP[o]] \leftarrow writeloc$. The last-packet pointer is updated ($lastP[o] \leftarrow writeloc$), and the packet-list is terminated ($NP[writeloc] \leftarrow nil$).

The next-chunk fields provide a similar linking function between packet chunks. On a write, when a valid last-chunk pointer exists, the central buffer next-chunk location pointed to by last-chunk is updated with the location of the currently written chunk (if $lastC[i] \neq nil$, then $NC[lastC[i]] \leftarrow writeloc$). When an input port writes a chunk, it also updates its last-chunk pointer with the write location ($lastC[i] \leftarrow writeloc$).

The logical structure of a typical output port queue within the central buffer is shown in Figure 6. There are two packets shown, each with its associated chunks displayed in a column. Pointer

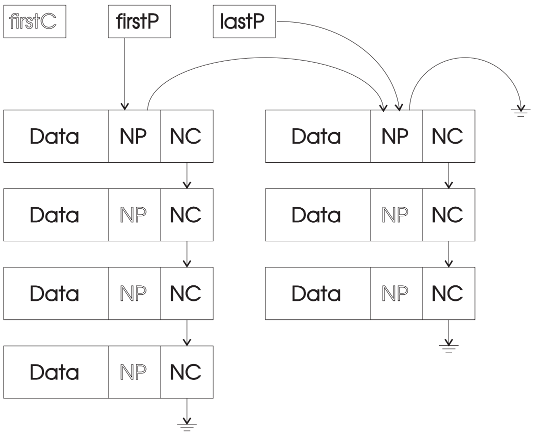

Figure 6: Structure of an output port queue within the central buffer

fields with no arrows emanating from them are currently invalid (e.g., next-packet fields are not used except for header chunks). It should be evident that the order of packets in a queue is entirely determined by the order of header chunk writes.

On the output port side, except for when a header chunk is being read, the output port first-chunk field is used to determine the location of the next central buffer chunk read ($readloc \leftarrow firstC[o]$). For header chunk reads, the first-packet pointer is used to determine the location to read from ($readloc \leftarrow firstP[o]$). When a header chunk is read from the central buffer, the next-packet list must be updated ($firstP[o] \leftarrow NP[readloc]$). Furthermore, on every chunk read the output port's first-chunk pointer is updated with the associated central buffer next-chunk pointer ($firstC[o] \leftarrow NC[readloc]$). Figure 7 shows the structure of the queue from Figure 6 after the first two chunks of the first packet in the queue have been read by the output port. Note that $firstP$ and $firstC$ have been updated, and $firstC$ is now a valid pointer field required for retrieving the next chunk from the queue.

Having examined the architecture of a typical central-buffered switch, we now describe how such an architecture can be enhanced to support asynchronous replication of multidestination worms.

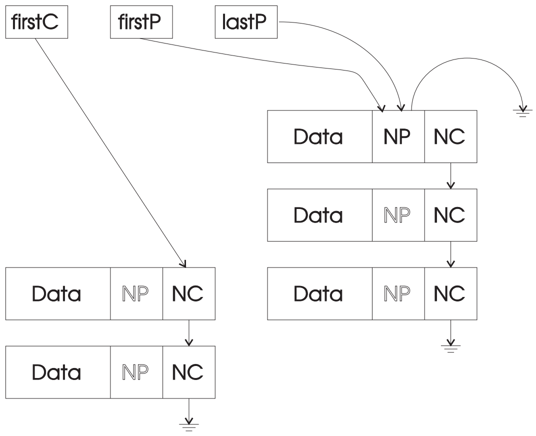

Figure 7: Structure of the same output port queue after two chunk reads

4.2 The Shared Central Buffer Replication Method

We desire a replication method that requires only one copy of the packet to be transmitted from the input port, and a method which acquires the deadlock-avoidance advantages of virtual cut-through operation. This goal can be achieved via the central buffer, with replication occurring during the *read* of the chunk out of the central buffer by the output port.

In this single-copy replication method, the input port writes each chunk into the central buffer once, but also initializes an associated counter c to k_s, where k_s is the degree of replication required at that switch. When an output port reads the chunk from the central buffer, it checks c. If $c \equiv 1$, then the chunk is thrown away. Otherwise, c is decremented.[2] The relatively large size of a chunk minimizes the impact of adding a counter to the storage space for each central buffer chunk.

The multiple-flit size of a chunk provides another striking advantage for this method: an input port can write a multidestination packet into the central buffer at full bandwidth, while simultaneously all k_s output ports are reading the packet at full bandwidth. Thus latency *and* required buffer space are minimized. We have established the motivation for the single-copy replication method; in the next section we examine the implementation issues.

4.2.1 Replication Implementation

In this section we describe modifications to the basic buffered wormhole strategy that provide efficient single-copy replication. The basic change is to provide a counter, c, with each chunk that indicates the number of output ports that have not yet read the chunk from the central buffer, as introduced in the preceding section. However, there remains a problem with the next-packet lists.

If a single header chunk is written to the central buffer, then there exists only a single next-packet pointer associated with this header chunk. Therefore, even if the last-packet and appropriate next-

[2]An implementation need not strictly follow this convention. For instance, c could initially be set to $k_s - 1$, in which case if $c \equiv 0$ then the last output port is reading the chunk. This choice might allow c to be implemented with 1 less bit.

packet pointers are updated for every destination output port, we will be faced with an unacceptable situation in which the next-packet lists are converged. i

One solution is to write k_s copies of the header chunks, one for each destination output port. This immediately provides a distinct next-packet pointer for each queue, maintaining their disjoint nature. All other chunks of the multidestination packet are written once. This method requires $k_s - 1$ more chunks of central buffer space. In addition, because it requires several cycles for the header write, this also temporarily reduces the achievable total bandwidth into (but not out of) the central buffer. Figure 8 illustrates this solution. In this figure, output ports x and y share the same

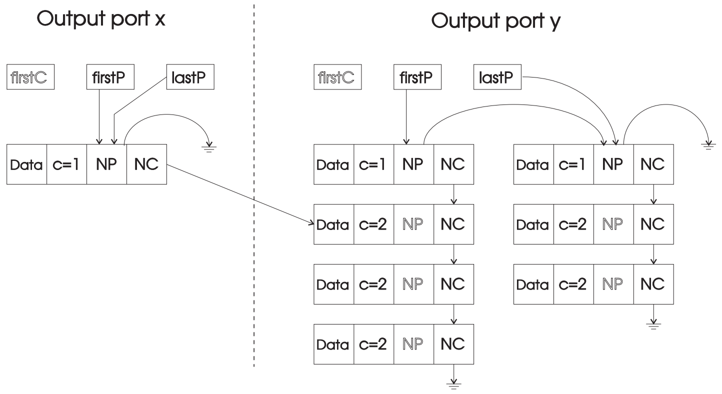

Figure 8: Replicated header chunks pointing to the same second chunk

(multidestination) packet at the start of their queues, yet their next-packet lists remain separate.

If hardware cost is not a constraining factor, there is another solution which does not require $k_s - 1$ extra cycles for writing the header chunk. Completely separate next-packet RAMs can be implemented for each of the k output ports (as opposed to the one shared next-packet RAM implemented by the NP fields shown in Figure 5). With separate next-packet RAMs, all k_s next-packet lists can be updated during the same cycle that the header chunk is written. Due to the replication of hardware, the next-packet lists are prevented from converging.

For the simulations studies in this paper, we have assumed the first option since hardware cost is an important constraint.

4.2.2 Emulating Virtual Cut-Through

As explained before, the equivalent of virtual cut-through (VCT) operation is required to avoid the dependence between output ports which can lead to multicast deadlock. VCT designs perform flow-control on a packet basis, allowing transmission of a packet from a switch only when the entire packet can be buffered at the downstream switch or node. Wormhole flow-control designs can be augmented to provide the aspects of VCT that are essential for multicasting, given sufficient buffering capability within each switch. To use a central buffer for emulating VCT, the total central buffer size must be as large or larger than the largest packet to be buffered.

When a multicast packet is replicated at a switch, each chunk of this packet must be stored in the central buffer before being read by all destination output ports. The packet header chunk is not allowed to enter the central buffer until there is a *guarantee* that the entire packet will *eventually* obtain space within the central buffer. This does not necessarily require that space for the entire packet exists *prior* to writing the header chunk.[3] When the header chunk is written (at least before any other chunk from any input port is subsequently written) the required number of chunks are reserved for use by that multicast packet only. In designs that maintain a "free chunk counter," this is most easily accomplished by decrementing the free space count by the total number of multicast packet chunks to be written.

To summarize, although the normal wormhole switch-to-switch flow-control is used between switches, the multicast packet header chunk is prevented from entering the central buffer until the entire multicast is guaranteed to fit within this buffer. This is a weaker requirement than VCT, and is only applied to multicast packets which are to be replicated within that switch. Thus, multidestination worms can be implemented in central buffer-based switch architectures with very little additional hardware support.

If some or all of the destination output ports are idle, it is possible to forward flits directly to these idle ports via the crossbar to minimize latency. However, because the idle ports could soon block from downstream traffic congestion, these ports might eventually require use of the central buffer as well. Conversely, a full central buffer could prevent the multicast packet from being written into the buffer (for the "busy" output ports) at full bandwidth, and this would slow the progress of succeeding flits destined for the idle ports. To simplify the accompanying design issues, we did not consider forwarding multicasts through the crossbar.

5 Input Buffer based Switch Architecture

In this section we describe an input-buffer-based switch architecture for implementing multidestination worms. We first describe the basic idea behind the switch architecture and then provide two alternative implementations that can reduce the wiring complexity in a switch implementation. The important change from an architecture that supports only unicast packets, is that the input buffers must be larger than the largest multicast packet allowed in the system. This ensures that any multicast packet that arrives at an input port is guaranteed to be completely buffered eventually. (The packet at the head of the buffer will be eventually freed. This implies that the packets behind it will definitely progress one by one to the head of the buffer. Since the buffers are assumed to be larger than the largest packet, at this point the packet will be completely buffered at the switch.)

[3]If the designer can identify cases in which chunks currently within the central buffer are guaranteed to be read and freed, then these chunks may be able to be counted toward the available buffer space. These cases will vary from design to design and may also be topology-dependent.

5.1 Enhancing an Input-Buffered Architecture for Multicast

We now describe the basic idea behind enhancing an input buffered architecture to support multicasting using multidestination worms.

Just as with the central-buffered architecture, flits of an arriving multidestination worm are assembled into chunks and stored in a chunk-wide RAM that constitutes the input buffer. When a multidestination worm arrives at the switch, a count value is associated with every one of its chunks. The switch decodes the multidestination worm header to determine the output ports to which the worm should be forwarded. A request is then enqueued for each of these output ports and the idle output ports begin reading chunks of the message from the appropriate input ports. As a chunk is read, the count value associated with it is decremented and the chunk is discarded once all output ports have read it. The chunk that is read is stored in an output buffer from where it is transmitted onto the next link on a flit by flit basis. Note that the header chunk is also read by each of the output ports as part of the message—every branch of the multidestination worm carries the same bit string header implying that no header manipulation needs to be done at the intermediate switches. As discussed in the previous section, chunking allows output ports to maintain a flit per cycle rate while requiring only a 2 port (1 read, 1 write) RAM. Figure 9 shows a snapshot of a multidestination worm with asynchronous replication making progress in an input-buffer-based switch architecture. In the figure, the multidestination worm enqueues a request for a busy port, while the ports that are available read chunks of the worm from the input buffer, simultaneously decreasing the associated count value. The ports take turns reading from the input buffer.

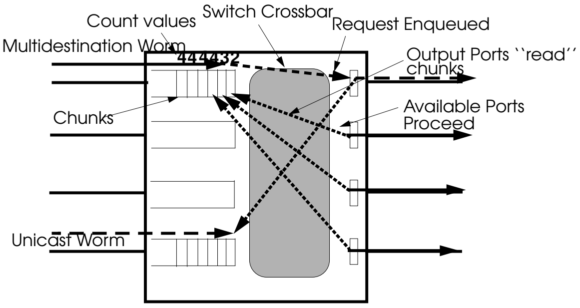

Figure 9: Snapshot of a multidestination worm making progress through a switch with an input buffer based architecture.

An implication from the above description is that a k-flit chunk is transmitted in a single cycle across the switch crossbar. This would require a chunk-wide crossbar in a naive implementation, which would increase the wiring complexity within the switch chip. Assuming a MUX implementation of a crossbar, Fig. 10 shows an organization of the input and output buffers as well as the connectivity between them. Let us define a flit-MUX as a flit-wide k to 1 MUX.[4] The suggested organization uses k large MUXs where each large MUX has k k-flit wide inputs and one k-flit

[4] A flit-MUX may actually be implemented as *flitsize* smaller MUXs, each of which has k 1-bit inputs and one 1-bit output.

wide output. Each of these larger MUXs is actually implemented using k flit-MUXs. Such an implementation incurs high wiring complexity.

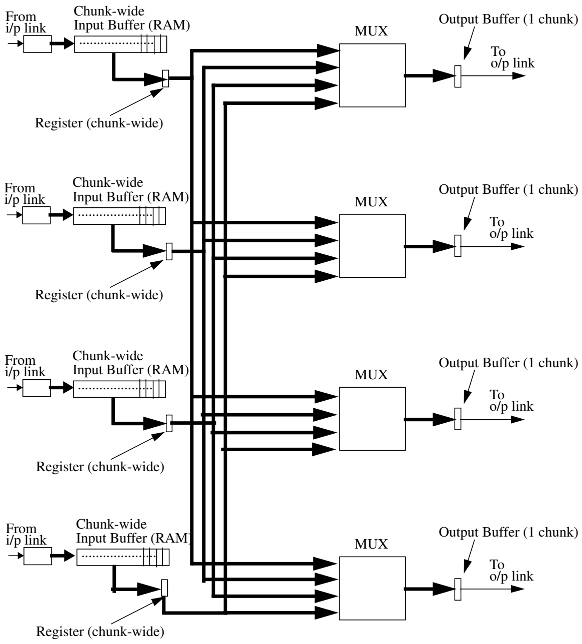

Figure 10: Switch organization to implement multidestination worms using larger MUXs in a 4×4 switch. The thick lines represent chunk-wide ($k \times flitsize$) lines; the thin lines represent flit-wide lines.

To alleviate this problem, we propose alternative implementations which perform as well as an implementation with a chunk-wide crossbar in the next section. These implementations pipeline the transfer of flits in a chunk from the input to output by either (i) using a series of registers to pipeline the transfer of the flits from the chunk-wide RAM, or (ii) using k separate flit-wide RAMs instead of a single chunk-wide RAM (where k is the number of input and output ports in the switch, which defines the chunk size as described before).

Another important design decision in this architecture is whether packets at different points in the input buffer should be allowed to make progress or whether progress should be restricted to the packet at the 'head' of the input buffer [40]. The former option implies that a fairly complex next packet list must be associated with every output port, much like the next packet list described in the context of the central buffer in the previous section. While this list is associated with the single central buffer in the previous section, the presence of multiple input buffers complicates the structure of this list. This complexity is eliminated if only packets at the head of the input buffer are allowed to make progress—an output port need only know which input port it must serve next, the location to read from to begin transfer is implicit. We therefore decided to adopt the approach with packets being processed in FIFO order from the head of the input buffer for our study.

5.2 Reducing Switch Complexity

We now propose two modified switch architectures that can reduce the cost and wiring complexity when compared to the design outlined in the previous subsection, while retaining all of the functionality and performance. Both methods reduce the width of the MUXs in the multicasting circuit.

5.2.1 Register-Staircase Approach

This method uses a set of k registers associated with every input buffer in an effort to reduce the MUX width. A schematic of the proposed architecture for a 4×4 switch is shown in Fig. 11. This approach makes use of the following observations to achieve performance similar to that achieved by the model specified in the previous section: (i) all output ports transmit the chunk that they read from an input port on a flit by flit basis to their corresponding output link, and (ii) output ports can proceed in parallel if they read from different registers.

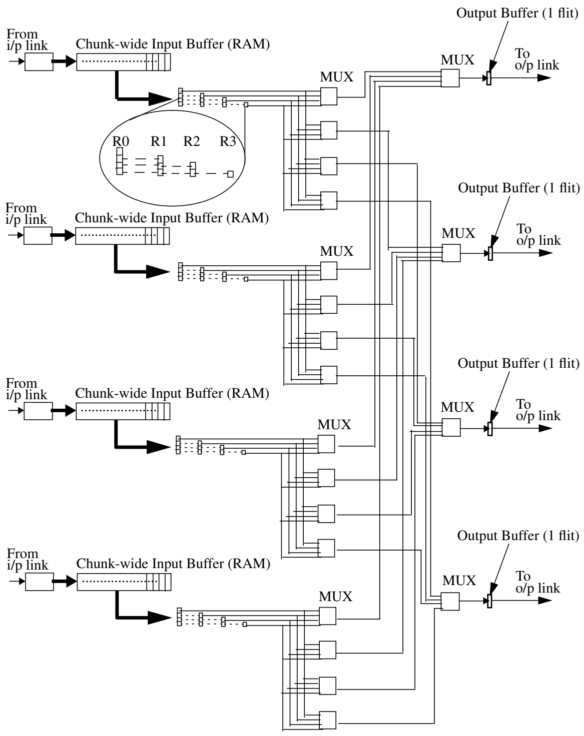

Figure 11: The switch organization under the Register-Staircase Approach.

The basic idea is as follows. Let us assume that all k output ports want to read from the same input buffer. The output ports that are ready take turns reading from the input buffer. Note that there is no restriction that all output ports have to be ready at the same time. An output port may begin contending with other output ports for the input buffer whenever it is ready. This flexibility allows output ports to be reading chunks from possibly different points in the input buffer. Flits are transferred in a *pipelined* fashion as depicted in Fig. 12. In a given cycle, one of the contending output ports (say O1) succeeds in reading a chunk from the input buffer. The chunk that is read is transferred to the first of the k registers (R0) in this cycle. In the next cycle, the next available output port (say O2) reads a chunk from the input buffer and transfers it to R0. In the same cycle, O1 reads the first flit from R0 and transfers the *remaining* flits of R0 to R1. In the third cycle, an output port O3 may read from the input buffer and transfer a chunk to R0 and O2 reads the first flit from R0 and transfers the remaining flits of R0 to R1. In the same cycle, O1 reads the first flit from R1 and transfers the remaining flits of R1 to R2 while putting the flit it read in the previous cycle onto the output link. This pipelined transfer continues with each output port transferring a flit to its corresponding output link once every cycle and is shown in Fig. 12.

Output Ports	CYCLE #1	CYCLE #2	CYCLE #3	CYCLE #4	CYCLE #5
O1	O1 reads chunk into R0	O1 reads first flit from R0 R1 <- Remaining flits from R0	O1 puts 1st flit on output link O1 reads next flit from R1 R2 <- Remaining flits from R1	O1 puts 2nd flit on output link O1 reads next flit from R2 R3<- Remaining flits from R2	O1 puts 3rd flit on output link O1 reads next flit from R3 and next chunk into R0
O2		O2 reads chunk into R0	O2 reads first flit from R0 R1 <- Remaining flits from R0	O2 puts 1st flit on output link O2 reads next flit from R1 R2 <- Remaining flits from R1	O2 puts 2nd flit on output link O2 reads next flit from R2 R3<- Remaining flits from R2
O3			O3 reads chunk into R0	O3 reads first flit from R0 R1 <- Remaining flits from R0	O3 puts 1st flit on output link O3 reads next flit from R1 R2 <- Remaining flits from R1
04				O4 reads chunk into R0	O4 reads first flit from R0 R1 <- Remaining flits from R0

Figure 12: The pipelined transfer of flits in a 4×4 switch under the register staircase approach in a scenario where all output ports read concurrently from the same input buffer.

It is to be observed that such an implementation reduces the size of the k output port MUXs in the circuit of Fig. 10 although the total number of flit-MUXs has increased (a total of $(k^2 + k)$ flit-MUXs are used in a $k \times k$ switch). Furthermore, there is no loss in performance when compared to the previous approach. Most importantly, we have moved much of the dense wiring complexity into the separate input port modules, avoiding huge multi-drop buses from crossing the chip to the output ports.

5.2.2 Split-RAM Approach

We now present an alternative organization for reducing the complexity of the wiring used in the switches. The basic idea behind this organization is to use k *flit-wide RAMs* (FWRs) instead of a single chunk-wide RAM. The k flits of the chunk are written into the FWRs with the ith flit of the chunk being written to the ith FWR $(0 \leq i \leq k-1)$. Figure 13 gives a schematic of the proposed design.

The output ports read flits from each of the RAMs in sequence. To set up the pipeline, an

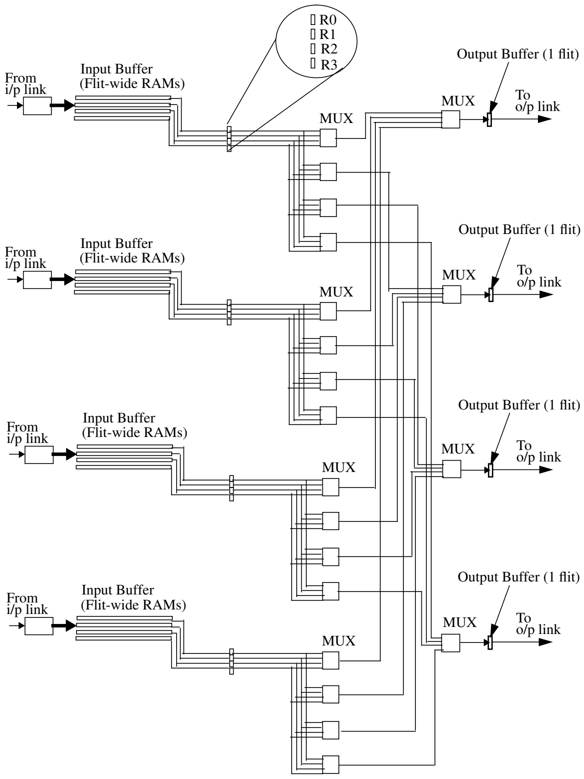

Figure 13: The switch organization under the Split-RAM Approach.

output port reads a flit from one of the FWRs and stores the flit in the corresponding register. In the next cycle, the output port reads the flit from this register and stores it in the output buffer while reading the next flit from the next FWR into the corresponding register. In every succeeding cycle, the output port transfers the flit in its output buffer to the output link, reads in the next flit from an appropriate register into the output buffer, and reads in the next flit from a FWR into a corresponding register. The destination output ports of the multicast packet that are ready, read flits from the k FWRs in this pipelined fashion. Once the pipeline is established, each of the ready destination output ports transmits one flit every cycle on their associated output links.

The number and size of MUXs used under this approach is exactly the same as with the register staircase approach. However, there is no need for the register staircase logic. Also, k flit-wide RAMs are needed instead of a single chunk-wide RAM. This implies that for the flits to be read from the k FWRs in parallel, k separate copies of the read control logic are required. It is to be noted that this method performs as well as the method using the register staircase. Again, the complexity of the wiring problem between the input ports and output ports is much improved. The Split-RAM method is similar to the pipelined memory approach proposed in [13].

5.3 Summary

We have presented an internal switch architecture capable of providing dynamic multicast via a crossbar with chunk-wide data paths. The switch crossbar requires k of these flit-MUXs for each of the k output ports, for a total of k^2 flit-MUXs. Each input port sends a k-flit output to all output ports, creating a difficult wiring problem aggravated by multi-drop signals.

We suggested two alternatives to address this wiring problem while maintaining the same performance. In each alternative, $(k^2 + k)$ flit-MUXs are used, but each input port sends a separate single flit to each output port. We anticipate that the dramatic improvement in the input port to output port wiring congestion will easily offset the extra MUX cost and the additional logic required to set up these 1-flit outputs. In addition, each of the suggested alternatives requires only a 1-flit register at each output port, compared to a k-flit register used in the original approach. The Register-Staircase approach will be advantageous for technologies in which latches are relatively cheap, while the Split-RAM approach will be advantageous when the penalty for dividing RAMs into multiple thinner RAMs is not excessive. These considerations vary with technology, and we therefore refrain from supporting one approach over the other.

In summary, the primary modifications we propose to an input-buffered switch supporting only unicast messages so that it supports multidestination messages are: (a) an input buffer per input port which is larger than the largest worm allowed in the system, (b) an arbitration mechanism to select among competing output ports, and (c) counters associated with packet chunks. The changes proposed for supporting the multidestination mechanism are therefore few and can be easily incorporated into existing switch architectures.

6 Comparing the two architectures

In this section we qualitatively examine the two proposed architectures before comparing them to software multicasting in the next section. Both schemes will benefit from using fewer links overall for a multicast, compared to the links used by the m messages required for the software multicast. The central buffer approach is significantly more complex, thus it is not worth considering unless it is expected to provide better performance. It is well-known that shared central buffers provide superior performance for most unicast traffic [42, 44].

The central buffer based architecture provides a dynamically-shared resource for input and output ports. In contrast, the input buffer implementation statically and evenly divides the available storage among the input ports. If the load among the input ports is unbalanced, the central-buffer-based scheme is likely to benefit because the ports have shared access to a larger buffer and can potentially use nearly the entire space available even when some or all of the intended output ports are busy with other traffic. In a bidi-MIN network, multidestination worms adaptively travel from the source to the least common ancestor and then turn back. They then replicate on their downward path, tending to cause more blocking on the “right” input ports than on the “left” input ports, and a centrally-buffered scheme can adjust better to this imbalance.

Another expected drawback of the input buffer based implementation is head-of-the-line blocking: a blocked packet at the head of the FIFO blocks all packets behind it. Multicast is likely to exacerbate this problem, as the packet at the head of the input FIFO will not be dislodged until all destination output ports have read the entire packet. In the next section, our simulation experiments attempt to quantify these comparisons.

7 Performance Evaluation

To assess the performance impact of input FIFO replication and central buffer replication schemes, we conducted simulations on a C++/CSIM-based model [27] incorporating these options. To judge hardware multicasting versus software multicasting methods, we also implemented the binomial tree-based U-Min algorithm [46] that eliminates link contention among the unicast messages of a software multicast.

7.1 Simulation parameters and methodology

The default performance parameters and the central buffer model were based roughly on a contemporary switch: the SP Switch of the IBM RS/6000 SP systems which is a successor to the SP2 High Performance Switch [42]. Each switch in these systems has 8 input ports and 8 output ports, where input and output ports are typically paired and connected to another switch's input/output port pair over a bidirectional link. Each output (input) port can send (receive) a 2-byte flit every 13.33 nsec cycle, for a maximum of 300 MB/s of bidirectional traffic over the link. For our simulations, we count latency in cycles and we assume 2-byte flits. Minimum latency through each switch is assumed to be 6 cycles, with 4 cycles required for reading data out of the input FIFO, determining the route, requesting output port(s), and granting the request. For input port FIFO replication or for traversing the switch via the central buffer, flits are deserialized into a chunk, requiring 7 extra cycles for the 8-flit chunks.

As in the SP Switch, we assume the central buffer accommodates 256 chunks, each of which contains eight 2-byte flits, for a total of 4 Kbytes of storage. In the central buffer model, each of the 8 input FIFOs contains 64 flits, for a total on-chip packet buffer space of 5 Kbytes. To equalize storage between the central buffer and input buffering models, we assume 640 bytes (320 flits) of storage for each input FIFO in the input buffering model.

We assume that packet flits are immediately pulled from the network upon arrival at a node. As with most network simulations, in our experiments we measure latency vs. applied load and received vs. applied load (under both switch architectures). We simulated random traffic for 128-byte and 512-byte messages, and we assumed that both of these message sizes fit within a single packet. For an m-way multicast simulation on an n-node system, for each message we randomly chose m uniformly distributed destinations among the $n-1$ non-source nodes. Message transmission times for each node were exponentially distributed. Latency results include queuing time at the source node, and latency curves are not plotted above the saturation bandwidth (latency is infinite in a stable but saturated system). To create a stable environment for statistics collection, we typically ran the simulations with 100,000–150,000 cycles (or more) of cold start time before commencing data collection.

7.2 Multicast simulation issues

Multicast traffic evaluation raises new issues beyond unicast traffic. The definition of multicast latency is problematic: latency can be defined as (a) the latency of the last received message of the multicast, or (b) as average of the latencies of each received message of the multicast. Nupairoj and Ni [25] argue that (a) is the more important term in assessing message-passing collective communication performance. In another example, shared-memory systems multicasts might be used for cache line invalidations to multiple destinations, and the source must wait for the last received (and acknowledged) invalidation before modifying the cache line. Except for graphs in which we directly compare definition (a) with (b), we choose definition (a) for recording latency.

A second issue: when the network is operating below the saturation bandwidth for m-way hardware multicasts, the packet injection bandwidth B_i does not equal the packet receive bandwidth B_o. Instead, $B_o = mB_i$. Thus mB_i is effectively the load being injected into the network. One is tempted to simply plot latency versus B_o, but we prefer to plot performance measures against a stimulus, not a response. Most of our graphs display "effective input load" (mB_i) on the x-axis. Let a load of 1 for B_o represent full bandwidth received at each node. Although it is possible that $mB_i > 1$, a stable system will already be in saturation for $mB_i = 1$ anyway. Thus we do not show results for $mB_i > 1$.

A final issue is message startup overhead. In network simulations, unicast messages are typically assumed to have negligible startup overhead. With this assumption the network can easily be stressed beyond saturation during simulation, yielding more insights into its behavior. In implementing software multicast, most real systems have a non-negligible overhead between receiving a multicast packet and forwarding the packet along the next stage of the binomial tree. In our simulations we "zero" this overhead—minimizing the latency of software multicasting—to give it the best chance of competing against hardware multicasting.

We conducted simulations for both 16-node and 64-node fat-trees constructed from switching elements with 4 parents and 4 children (8 ports). Our default message size is 128 bytes (64 flits), which could correspond roughly to message sizes of request and reply packets for shared-memory systems. To further stress the network, we also plot performance for 512-byte messages. In the rest of this section we discuss the effects of varying a number of input loading and network characteristics.

7.3 Impact of multicast degree

To understand the effect multicast on network performance, we begin by varying the degree of the multicast m for a 16-node system, for switches with and without central buffers.

The top-left graph in Figure 14 compares software versus hardware multicasting latency using centrally-buffered switches in both cases, and the top-right graph shows a similar comparison for input-buffered switches. As m increases, the software multicast latency increases and satu-

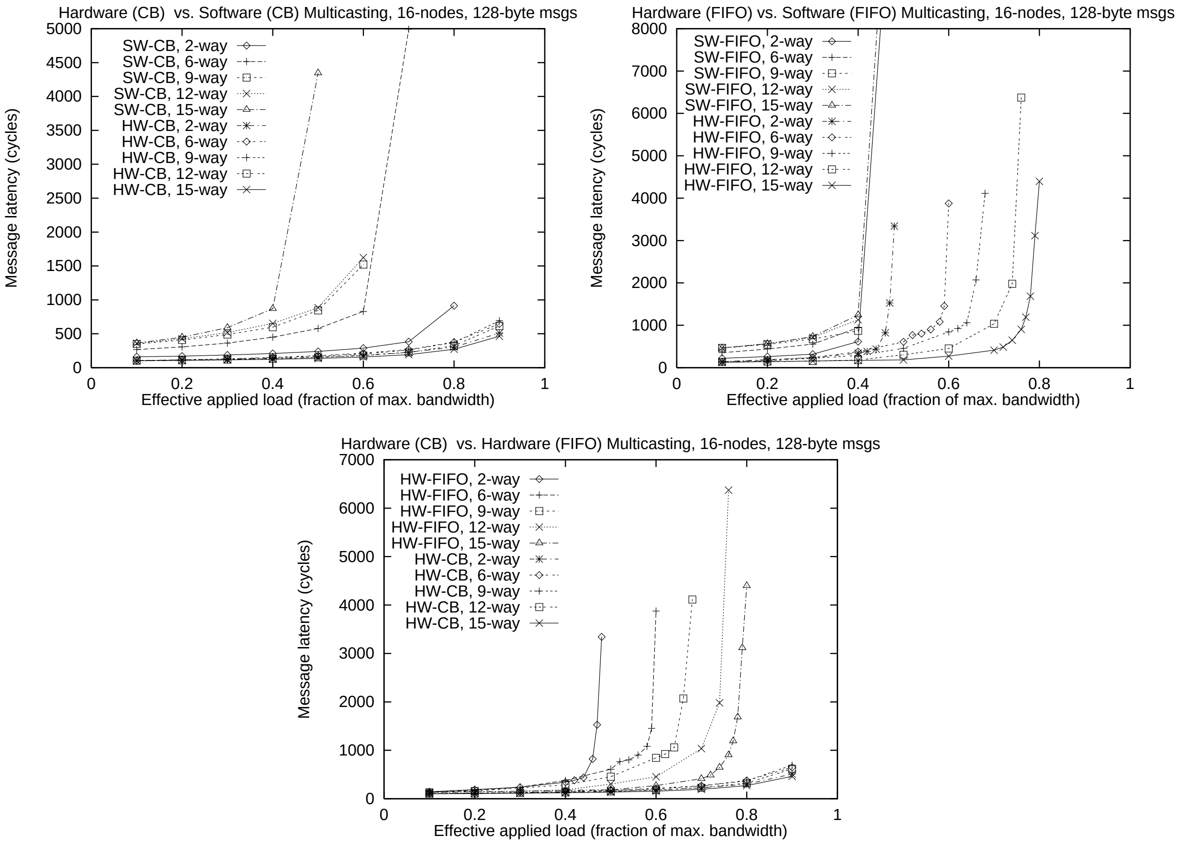

Figure 14: Latency vs. Effective applied load for software multicast and hardware multicast with central buffering and input buffering.

rates quickly. For a similar increase in m, hardware multicast latency actually *decreases*, since the same effective applied load results in less traffic under hardware multicast. Furthermore, hardware multicast saturates at a much higher applied load than the corresponding software multicast. Hardware multicast therefore performs significantly better than software multicast under both buffering schemes.

The bottom plot of Figure 14 compares our two proposed hardware replication methods. Comparing the top and bottom graphs, we can observe that for small m, input FIFO replication actually performs worse than software multicasting with a central buffer. This follows from the superiority of central buffer queuing over input queuing for unicast packets. However, in contrast to software multicasting, input FIFO replication latencies *decrease* with increasing m, and saturation bandwidth increases. As m increases, a larger percentage of the replication occurs late on the paths to the destination nodes. For example, for $m = 2$ in a 16-node system, most replication occurs at the 2nd-level switches, whereas for $m = 15$ most replication occurs at the 1st-level switches just before reaching the nodes. And as noted before, for $B_o = mB_i$, increasing m decreases B_i.

Figure 15 allows a better inspection of saturation bandwidth by plotting output bandwidth B_o versus effective input bandwidth mB_i. The bottom graph confirms that for input FIFO replication,

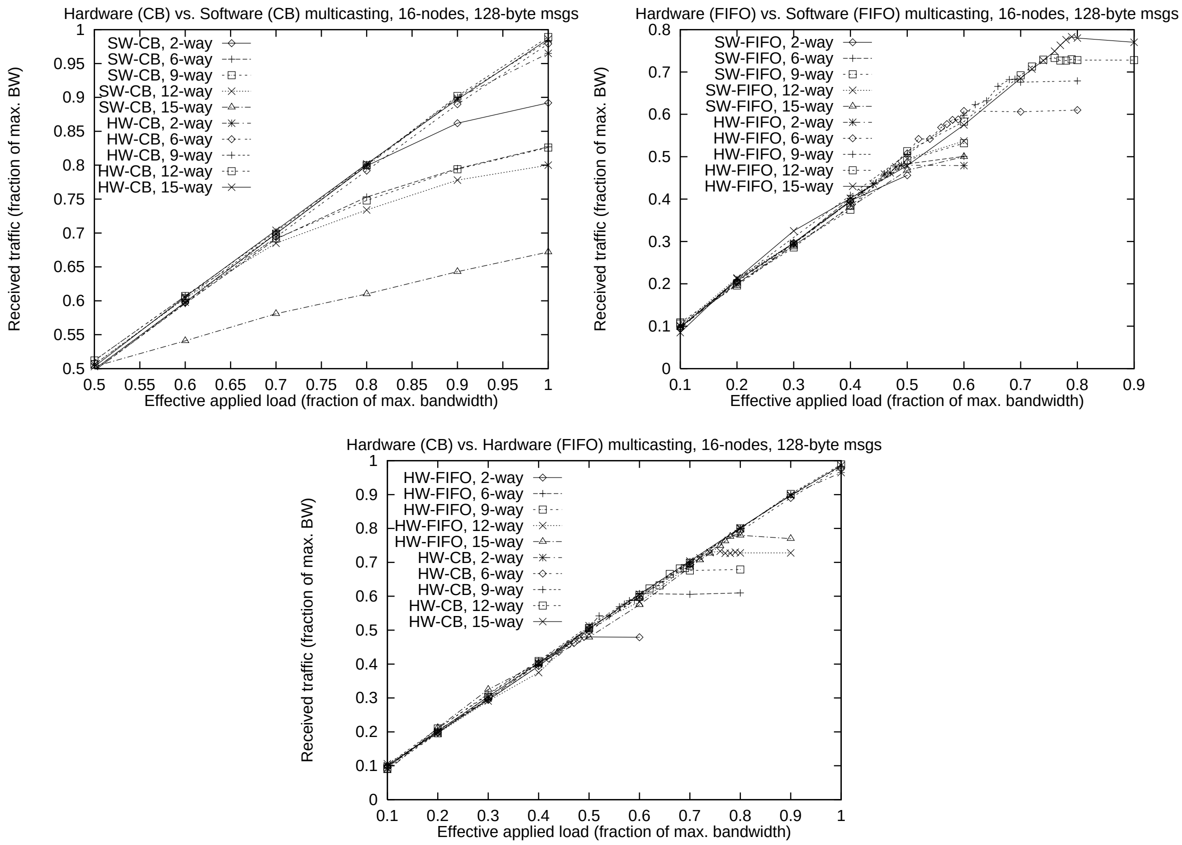

Figure 15: Received vs. Effective applied load for software multicast and hardware multicast with central buffering and input buffering.

saturation bandwidth increases with m. For the software multicasts shown in the top graph, the opposite is true. Unlike the characteristic flattened B_o lines for the bottom graph, the software multicast B_o does not flatten out after B_o becomes less than mB_i. This is because the software forwarding of messages is happening less frequently as latency increases, and this acts as a drag on the input bandwidth. Thus it may not be appropriate to characterize the software multicast as in saturation; the centrally-buffered switches continue to increase delivered bandwidth for this message size.

For the central buffer-based replication scheme, saturation bandwidth is near enough to 100% that it is hard to distinguish the curves, although the highest bandwidth is achieved for large m.

To better see the impact of m on saturation for central buffer replication, we increase the message size in the next section.

7.4 Impact of message size

The graphs we have shown thus far use 128-byte messages, which the large central buffers typically handle quite easily to over 90% of the maximum load. In this section, to stress the network further we examine the impact of a larger message size of 512 bytes.

Figure 16 is a latency graph for a 16-node system for small hardware and software multicasts. The 2-way and 4-way hardware central-buffer-based multicasts now saturate before 0.9 effective

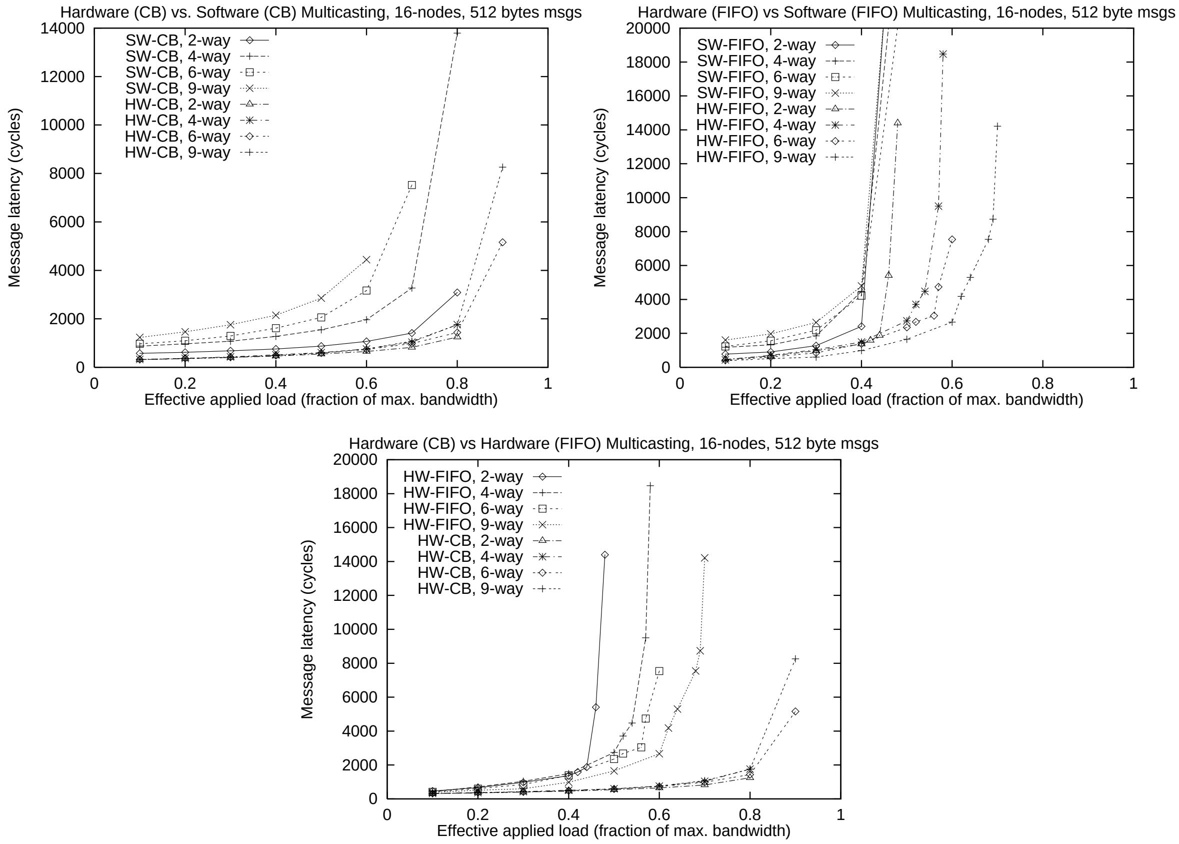

Figure 16: Latency vs. Effective applied load for a 16-node system, 512-byte messages

load, although the 6-way and 9-way multicasts still saturate at effective loads higher than 0.9. A similar decrease in the effective load for saturation is also observed for the input-buffer-based multicasts. Again this is because, for equivalent $B_o = mB_i$, larger m implies smaller B_i, resulting in less packets traversing the first stages of the network, although in the last stage of the network bandwidth is equivalent.

The software multicasts also saturate more quickly than for the 128-byte case. Just as for the 128-byte messages, saturation bandwidth decreases as m increases, because for software multicasts,

$B_o = B_i$.

The results for 128-byte and 512-byte simulations are qualitatively similar, and throughout the rest of the paper we shall choose message size arbitrarily.

7.5 Impact of system size

To this point we have relied on 16-node simulations to test various parameters. To assess whether lessons learned from 16-node simulations are applicable to larger systems, we now compare 16-node and 64-node hardware multicasting experiments for the central buffer model. We do not show software multicast results for 64-node systems, because they grow even worse as m increases.

Figure 17 shows latency and bandwidth for low, medium, and high values of m for each system size. The 64-node system exhibits higher latency and slightly lower saturation bandwidth than the

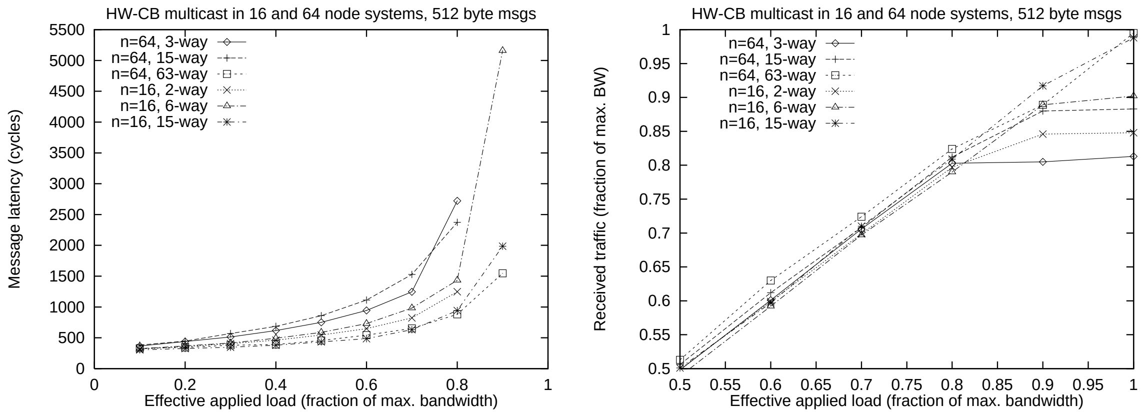

Figure 17: Performance of 16-node and 64-node systems, 512-byte messages

16-node system for comparable multicasts. This fits well with experience from unicast simulations on fat-trees: although fat-trees theoretically scale bandwidth linearly, in a larger network random traffic requires more hops on average and therefore encounters more contention, lowering achievable bandwidth. Just as for the 16-node system, the low m multicast has low latency but saturates earliest.

7.6 Interaction between multicast and unicast traffic

The experiments so far have stressed the network by applying a single mode of input traffic : random m-way multicast traffic. In a typical parallel application, however, message traffic will not be entirely multicast or entirely unicast at any moment.

To better understand the interaction of multicast and unicast traffic, we simulate random *bi-modal* traffic in which 20% of the received load is due to multicasts. Thus B_o consists of two components B_{ou} and B_{om} (where B_{ou} and B_{om} are the packet receive bandwidths of the unicast and multicast messages respectively), and $B_{om} = 0.2B_o$. For example, if $B_o = 0.7$, then $B_{om} = 0.14$ and $B_{ou} = 0.56$. Figure 18 shows the result of combining unicasts with 4-way multicasts.

For 80% unicast traffic, it is natural that the average latency should closely follow the unicast message latency unless multicast latency grows disproportionately large. Hardware multicasts perturb unicast traffic much less than do software multicasts, resulting in a lower combined latency. This is because hardware multicasts occupy network links for less time than software multicasts, due to both lower latency and a lower number of links traversed during the multicast.

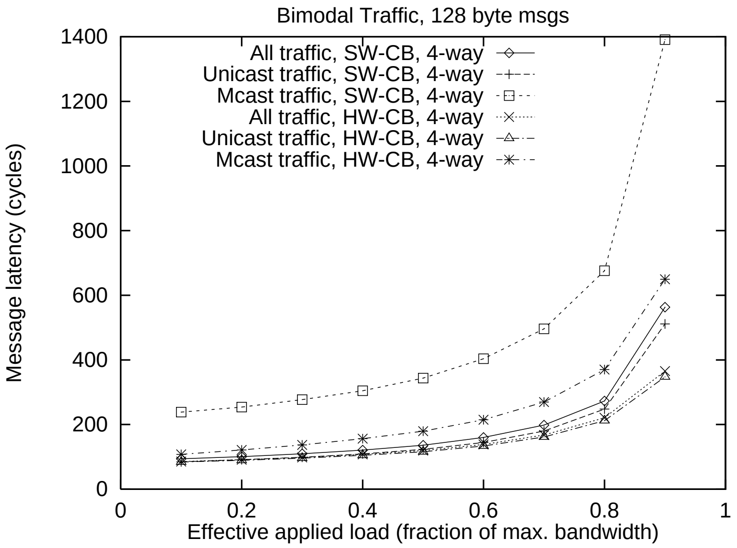

Figure 18: Bimodal latency for a 16-node system with central buffers, 128-byte messages

7.7 Measuring latency in an alternate way

We mentioned in Section 7.2 that there are two alternatives for calculating the average latency of multicasts. To this point, we have chosen definition (a): the latency of the last received message of the multicast. In this section we compare (a) to (b): the average of the latencies of each received message of the multicast.

Figure 19 displays this comparison for $m = 6$ on the top, and $m = 15$ on the bottom. For the $m = 6$ case, (a) and (b) are nearly equal for the hardware central buffer multicast at low loads, but for higher loads (a) exceeds (b) by almost 60%. For hardware input FIFO replication, the difference between (a) and (b) grows more rapidly, as does the magnitude of the latencies. For software multicasts with either of the buffering approaches, there is always a significant difference. As noted in Section 7.3, for small m software multicasting with central buffers actually saturates later than hardware multicasting with input FIFO replication, hence the crossovers in the graph.

For the $m = 15$ hardware multicast cases, the (a) and (b) definitions match closely throughout a large load range. Software multicast has no such property due to the $\lceil \log_2(m+1) \rceil$ steps required to complete it.

To summarize, when (a) and (b) definitions are nearly equal, it indicates that different branches of the multicast messages are arriving nearly synchronously. This is an attractive property for loosely synchronous algorithms and collective operations such as barrier synchronization that can be aided by multicasting. Hardware multicasting displays this nearly synchronous behavior for low loads or high degree of multicast (when all other traffic is multicast traffic of similar degree, and the network is not saturated).

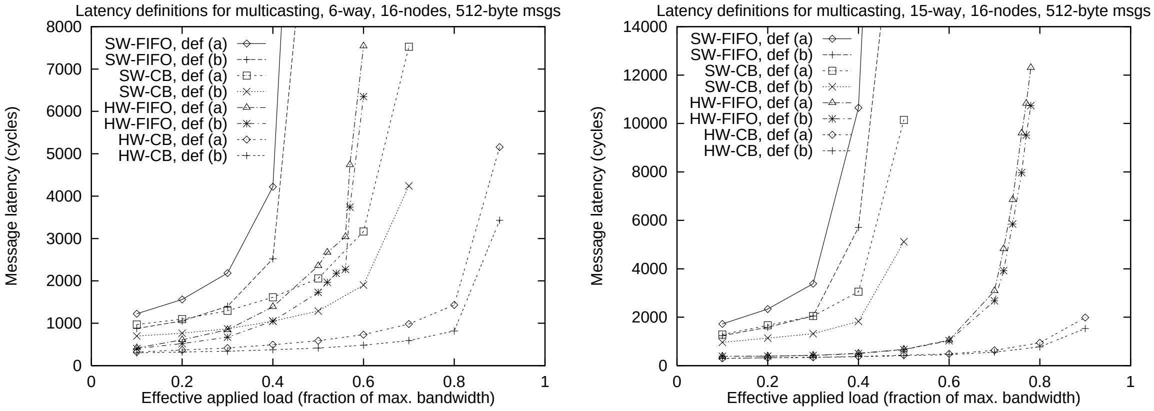

Figure 19: Comparing latency definitions for a 16-node system, 512-byte messages

7.8 Summary

We now summarize our simulated performance results. The hardware replication schemes using architectures with either buffering organization exhibited a number of desirable characteristics in comparison to software multicasting using the corresponding buffering organization. Latency was much lower and saturation bandwidth was higher for the entire range of multicast degree m, system size n, and message size. When random multicasts are part of bimodal traffic along with random unicast messages, the hardware multicasts perturb the unicasts less than do the software multicasts. Hardware multicast latency varies less over the range of destinations of a particular multicast, especially for light loads or large m.

The input-FIFO-based hardware replication scheme did not fare as well as the central-buffer-based schemes, especially at high loads, but for large m this scheme outperformed the software multicasting with a central buffer. This is significant considering that central buffers perform much better than input FIFOs for unicast traffic. The software multicasting scheme using input-buffered switches performs worse than the other three options that were simulated.

8 Related Work

Other recent research has focussed on hardware multicasting schemes. As mentioned before, a path based model was proposed for multicasting on two dimensional mesh networks by Lin and Ni [19, 18]. This work was extended to a Base Routing Conformed Path (BRCP) model by Panda et al. [28] applicable to any k-ary n-cube network. Kesavan and Panda [16] have studied the problem of multiple multicast for meshes using both software based and hardware path based multicasts.

In the context of switch based parallel systems, Chiang and Ni have proposed a method for synchronous replication of multidestination worms [6] adopting a deadlock avoidance scheme at the switches. A somewhat related multicasting technique proposed in the context of DSM systems based on direct networks is the tree based multicasting scheme proposed in [20]. Instead of guaranteeing freedom from deadlock, a deadlock recovery scheme is proposed that "prunes" blocked branches. This method is effective only for short messages, as is prevalent in a distributed shared memory system. The architectures proposed in the current paper allow multicast for packets as large as the buffer size at the switches and the technique works well for both long and short messages. Andrews et al. [1] have proposed a method for tree based multicast using bit-string encoding in the context of dance-hall architectures. However this work only focuses on store and forward networks and short message lengths.

Some parallel systems like the CM-5 [17], Meiko CS2 [3] etc. provide facilities for multicasting. However, the CM-5 uses a separate network for multicast operations and only one multicast is allowed in this network at any given time. Data-less multicasts or "eurekas" are supported in the Cray T3E [33] in the same network used for normal data communication. Using data-less multicast packets avoids the deadlock scenarios outlined in this paper and the solution does not extend to multicast of data packets. The approach presented in this paper allows arbitrary numbers of multicasts, each of size upto the maximum packet size allowed in the system, to proceed concurrently while using the same network as all other data to achieve multicast communication. Furthermore, unlike the approach proposed for the Meiko CS2, multicast to arbitrary destination sets is allowed.

9 Conclusion

In this paper we have presented two switch architectures with differing buffer/queue organizations for implementing multidestination worms in switch-based parallel systems. We have described in detail how a central-buffer-based switch architecture that supports only unicast message passing can be modified to support multidestination message passing at little additional cost. We also showed how an input-buffer-based switch architecture can be similarly extended to support multidestination message passing, and presented two alternatives for implementing such an input-buffer-based architecture that can reduce wiring complexity in a practical switch implementation. We then performed extensive simulations to evaluate the relative performance of our proposed switch architectures and to compare the achieved hardware multicast performance with the best software multicast algorithm. To make the comparisons more interesting, we factored out the high start-up overhead associated with the software schemes in our performance comparisons. The performance achieved by such a software multicast algorithm is the best that can be achieved using point to point communication primitives alone.

Our performance studies have shown that central-buffer-based switch architectures prove extremely beneficial in improving hardware multicast performance. Such architectures can deliver good performance across an entire range of applied loads, message lengths, multicast degrees, and system sizes. Although the alternative implementation using an input FIFO buffer delivers multicast performance similar to the central-buffer-based implementation for single multicasts and multiple multicasts at low loads, the degradation in performance is rapid and occurs at relatively light loads. *If the associated start-up overhead is neglected*, the central-buffer-based software multicast outperforms hardware multicast using the input-FIFO-based implementation for low multicast degrees on a 16 node system. This implies that software multicast, using efficient messaging layers and support at the network interfaces to reduce start-up overhead, has the potential to offer good performance when used in conjunction with central-buffer-based switches.

We are currently studying the impact of enhancements to the switch architecture to support reliable multicast, and efficient and reliable barrier synchronization [39, 34]. The relative performance of such a scheme compared to an efficient software-based scheme in terms of absolute barrier latency and in terms of their impact on other network traffic is being evaluated. We are also studying the impact of hot spot traffic and other traffic patterns. We are also studying the relative performance of multicast with switch support and multicast with network interface support [35]. Finally, we will also be working on measuring the impact of these enhancements to the communication subsystem on benchmark application performance in the DM and DSM [45] domains.

References

[1] J. B. Andrews, C. J. Beckmann, and D. K. Poulsen. Notification and multicast networks for synchronization and coherence. *Journal of Parallel and Distributed Computing*, 15:332–350, Aug. 1992.

[2] M. Barnett, D. G. Payne, and R. Van de Geijn. Optimal Broadcasting in Mesh-Connected Architectures. Technical Report TR91-38, Dept. of Computer Science, University of Texas at Austin, Dec 1991.

[3] J. Beecroft, M. Homewood, and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. *Parallel Computing*, 20:1627–1638, Nov 1994.

[4] J. Bruck, R. Cypher, P. Elustando, A. Ho, C.T. Ho, V. Bala, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. In *Proceedings of the International Parallel Processing Symposium*, 1994.

[5] C. M. Chiang and L. M. Ni. Multi-Address Encoding for Multicast. In *Proceedings of the Parallel Computer Routing and Communication Workshop*, pages 146–160, May 1994.

[6] C. M. Chiang and L. M. Ni. Deadlock-Free Multi-Head Wormhole Routing. In *Proceedings of the First High Performance Computing-Asia*, 1995.

[7] Cray Research, Inc. *Cray T3D System Architecture Overview*, 1993.

[8] D. Dai and D. K. Panda. Reducing Cache Invalidation Overheads in Wormhole DSMs Using Multidestination Message Passing. In *International Conference on Parallel Processing*, pages I:138–145, Chicago, IL, Aug 1996.

[9] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, pages 547–553, May 1987.

[10] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[11] B. Duzett and R. Buck. An Overview of the Ncube-3 Supercomputer. In *Proceedings of the Frontiers of Massively Parallel Computation*, pages 458–464, 1992.

[12] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[13] Manolis Katevenis, Panagiota Vatsolaki, and Aristides Efthymiou. Pipelined Memory Shared Buffer for VLSI Switches. In *Proceedings of ACM SIGCOMM*, pages 39–48, August 1995.

[14] P. Kermani and L. Kleinrock. Virtual Cut-Through: A New Computer Communications Switching Technique. *Computer Networks*, 3(4):267–286, Sept. 1979.

[15] R. Kesavan, K. Bondalapati, and D. K. Panda. Multicast on Irregular Switch-based Networks with Wormhole Routing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-3)*, pages 48–57, February 1997.

[16] R. Kesavan and D. K. Panda. Minimizing Node Contention in Multiple Multicast on Wormhole k-ary n-cube Networks. In *Proceedings of the International Conference on Parallel Processing*, pages I:188–195, Chicago, IL, Aug 1996.

[17] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[18] X. Lin, P. K.McKinley, and L. M. Ni. Deadlock free multicast wormhole routing in 2-D mesh multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):793–804, Aug. 1994.

[19] X. Lin and L. M. Ni. Deadlock-free Multicast Wormhole Routing in Multicomputer Networks. In *Proceedings of the International Symposium on Computer Architecture*, pages 116–124, 1991.

[20] M. P. Malumbres, J. Duato, and J. Torellas. An Efficient Implementation of Tree-Based Multicast Routing for Distributed Shared-Memory Multiprocessors. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 186–189, New Orleans, LA, October 1996.

[21] P. K. McKinley and D. F. Robinson. Collective Communication in Wormhole-Routed Massively Parallel Computers. *IEEE Computer*, pages 39–50, Dec 1995.

[22] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni. Unicast-based Multicast Communication in Wormhole-routed Networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1252–1265, Dec 1994.

[23] L. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, Feb. 1993.

[24] Lionel Ni. Should Scalable Parallel Computers Support Efficient Hardware Multicasting? In *ICPP Workshop on Challenges for Parallel Processing*, pages 2–7, 1995.

[25] N. Nupairoj and L. M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC '97), Lecture Notes in Computer Science 1199*, pages 212–226, February 1997.

[26] D. K. Panda. Issues in Designing Efficient and Practical Algorithms for Collective Communication in Wormhole-Routed Systems. In *ICPP Workshop on Challenges for Parallel Processing*, pages 8–15, 1995.

[27] D. K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi, and V. Moorthy. Simulation of Modern Parallel Systems: A CSIM-based approach. In *Proceedings of the 1997 Winter Simulation Conference (WSC'97)*, pages 1013–1020, December 1997.

[28] D. K. Panda, S. Singal, and R. Kesavan. Multidestination Message Passing in Wormhole k-ary n-cube Networks with Base Routing Conformed Paths. Technical Report OSU-CISRC-12/95-TR54, The Ohio State University, December 1995. *IEEE Transactions on Parallel and Distributed Systems*. In Press.

[29] D. K. Panda and R. Sivaram. Fast Broadcast and Multicast in Wormhole Multistage Networks with Multidestination Worms. Technical Report OSU-CISRC-4/95-TR21, Dept. of Computer and Information Science, The Ohio State University, April 1995.

[30] W. Qiao and L. M. Ni. Adaptive Routing in Irregular Networks Using Cut-Through Switches. In *Proceedings of the International Conference on Parallel Processing*, pages I:52–60, Chicago, IL, Aug 1996.

[31] I. D. Scherson and C.-H. Chien. Least common ancestor networks. In *Proc. 7th Int. Parallel Processing Symp.*, pages 507–513, 1993.

[32] M. D. Schroeder et al. Autonet: A High-speed, Self-configuring Local Area Network Using Point-to-point Links. Technical Report SRC research report 59, DEC, Apr 1990.

[33] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII*, Sept. 1996.

[34] R. Sivaram. *Architectural Support for Efficient Communication in Scalable Parallel Systems*. PhD thesis, The Ohio State University, 1998.

[35] R. Sivaram, R. Kesavan, D. K. Panda, and C. B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, pages 452–459, August 1998.

[36] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 36–45, Oct 1996.

[37] R. Sivaram, D. K. Panda, and C. B. Stunkel. Multicasting in Irregular Networks with Cut-Through Switches using Tree-Based Multidestination Worms. In *Proceedings of the 2nd Parallel Computer Routing and Communication Workshop (PCRCW '97), Lecture Notes in Computer Science # 1417*, pages 39–52, June 1997.

[38] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1004–1028, October 1998.

[39] R. Sivaram, C. B. Stunkel, and D. K. Panda. A Reliable Hardware Barrier Synchronization Scheme. In *Proceedings of the 11th IEEE International Parallel Processing Symposium*, pages 274–280, April 1997.

[40] R. Sivaram, C. B. Stunkel, and D. K. Panda. HIPIQS: A High Performance Switch Architecture using Input Queuing. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 134–143, April 1998.

[41] C. B. Stunkel, D. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 High Performance Switch. In *Scalable High Performance Computing Conference*, pages 150–157, 1994.

[42] C. B. Stunkel, D. G. Shea, B. Abali, et al. The SP2 High-Performance Switch. *IBM System Journal*, 34(2):185–204, 1995.

[43] C. B. Stunkel, R. Sivaram, and D. K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 50–61, June 1997.

[44] Y. Tamir and G. L. Frazier. High-performance multi-queue buffers for VLSI communication switches. In *Proc. 15st Ann. Int. Symp. on Computer Architecture*, pages 343–354, May 1988.

[45] N. F. Tzeng and A. Kongmunvattana. Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer. In *Proceedings of the 1997 ACM International Conference on Supercomputing (ICS '97)*, pages 148–155, July 1997.

[46] H. Xu, Y.-D. Gui, and L. M. Ni. Optimal Software Multicast in Wormhole-Routed Multistage Networks. In *Proceedings of the Supercomputing Conference*, pages 703–712, 1994.